

Optimierende Compiler

LVA 185.A04, VU 2.0, ECTS 3.0
WS 2019/2020
(Stand: 17.12.2019)

Jens Knoop



Technische Universität Wien
Information Systems Engineering
Compilers and Languages



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1/1857

Table of Contents

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

Table of Contents (1)

Part I: Introduction

▶ Chap. 1: Motivation

- 1.1 Setting the Scene
- 1.2 An Extensive Illustrating Example
- 1.3 The Impact of Optimization: A Case Study
- 1.4 Compilers, Optimizing Compilers, and their Structure
- 1.5 Optimizations: Objectives and Categorization
- 1.6 Tools for Compiler Construction and Optimization
- 1.7 Summary, Looking Ahead
- 1.8 References, Further Reading

Table of Contents (2)

- ▶ Chap. 2: Classical Gen/Kill Data Flow Analyses
 - 2.1 Programs, Flow Graphs
 - 2.2 Forward Analyses
 - 2.2.1 Reaching Definitions
 - 2.2.2 Available Expressions
 - 2.2.3 Summary: Forward Analyses
 - 2.2.4 References, Further Reading
 - 2.3 Backward Analyses
 - 2.3.1 Live Variables
 - 2.3.2 Very Busy Expressions
 - 2.3.3 Summary: Backward Analyses
 - 2.3.4 References, Further Reading
 - 2.4 Taxonomy of Gen/Kill Analyses
 - 2.5 Summary, Looking Ahead
 - 2.6 References, Further Reading

Table of Contents (3)

Part II: Intraprocedural Data Flow Analysis

- ▶ Chap. 3: The Intraprocedural DFA Framework
 - 3.1 Preliminaries
 - 3.2 DFA Specifications, DFA Problems
 - 3.3 The Meet Over All Paths Approach
 - 3.4 The Maximum Fixed Point Approach
 - 3.5 Safety and Coincidence
 - 3.6 Soundness and Completeness
 - 3.7 A Uniform Framework and Toolkit View to DFA
 - 3.8 Summary, Looking Ahead
 - 3.9 References, Further Reading

Table of Contents (4)

► Chap. 4: Gen/Kill Analyses Reconsidered

4.1 Reaching Definitions

4.1.1 Reaching Definitions for a Single Definition

4.1.2 Reaching Definitions for a Set of Definitions

4.1.3 Reaching Definitions for a Set of Definitions: Bitvector Implementation

4.1.4 Reaching Definitions for a Set of Definitions: Gen/Kill Implementation

4.1.5 Reaching Definitions Analysis: Soundness and Completeness

4.2 Very Busy Expressions

4.2.1 Very Busyness for a Single Term

4.2.2 Very Busyness for a Set of Terms

4.2.3 Very Busyness for a Set of Terms: Bitvector Implementation

4.2.4 Very Busyness for a Set of Terms: Gen/Kill Implementation

4.2.5 Very Busyness Analysis: Soundness and Completeness

Table of Contents (5)

- ▶ Chap. 4: Gen/Kill Analyses Reconsidered (cont'd)
 - 4.3 Summary, Looking Ahead
 - 4.4 References, Further Reading
- ▶ Chap. 5: Constant Propagation and Folding
 - 5.1 Motivation
 - 5.2 Preliminaries
 - 5.3 Non-deterministic Constant Propagation: Problem Definition
 - 5.3.1 The Non-Deterministic CP Problem: Set View
 - 5.3.2 The Non-Deterministic CP Problem: Valuation View
 - 5.3.3 Equivalence of Set and Valuation View
 - 5.3.4 The C^T Approximation induced by C^V
 - 5.3.5 CP Algorithms: Soundness and Completeness
 - 5.3.6 CP Algorithms: Conservativity and Optimality
 - 5.3.7 Classifying CP Algorithms
 - 5.3.8 The C^T , C^V Approximations of CP Algorithms

Table of Contents (6)

- ▶ Chap. 5: Constant Propagation (cont'd)
 - 5.4 Simple Constants
 - 5.4.1 DFA States, DFA Lattice
 - 5.4.2 Simple Constants: Specification
 - 5.4.3 Termination, Safety, and Coincidence
 - 5.4.4 Soundness and Completeness
 - 5.4.5 Illustrating Example
 - 5.5 Linear Constants
 - 5.5.1 DFA States, DFA Lattice
 - 5.5.2 Linear Constants: Specification
 - 5.5.3 Termination, Safety, and Coincidence
 - 5.5.4 Soundness and Completeness
 - 5.5.5 Illustrating Example
 - 5.6 Copy Constants
 - 5.6.1 DFA States, DFA Lattice
 - 5.6.2 Copy Constants: Specification
 - 5.6.3 Termination, Safety, and Coincidence
 - 5.6.4 Soundness and Completeness
 - 5.6.5 Illustrating Example

Table of Contents (7)

▶ Chap. 5: Constant Propagation (cont'd)

5.7 Q Constants

- 5.7.1 Background and Motivation
- 5.7.2 The Q-*MaxFP* Approach
- 5.7.3 Q Constants: The Specification
- 5.7.4 Termination, Safety, and Coincidence
- 5.7.5 Soundness and Completeness
- 5.7.6 Illustrating Example
- 5.7.7 Summary

5.8 Finite Constants

- 5.8.1 Background and Motivation
- 5.8.2 Finite Constants: The Very Idea
- 5.8.3 Finite Operational Constants
- 5.8.4 Finite Denotational Constants
- 5.8.5 Finite Constants
- 5.8.6 Deciding Finite Constants: Algorithm Sketch
- 5.8.7 Finite Constants: Specification
- 5.8.8 Termination, Safety, and Coincidence
- 5.8.9 Soundness and Completeness
- 5.8.10 Illustrating Example, Hierarchies of Classes of FCs

Table of Contents (8)

► Chap. 5: Constant Propagation (cont'd)

5.9 Conditional Constants

5.9.1 Motivation

5.9.2 Preliminaries

5.9.3 Deterministic Constant Propagation: Problem Definition

5.9.4 DFA States, DFA Lattice

5.9.5 Conditional Constants: Specification

5.9.6 Termination, Safety, and Coincidence

5.9.7 Soundness and Completeness

5.9.8 Illustrating Example

5.10 VG Constants

5.10.1 Motivation

5.10.2 VG_{sc} Constants: The Basic Approach

5.10.3 VG_{ϕ} Constants: The Full Approach

5.10.4 Main Results

5.10.5 Illustrating Example

5.11 Summary, Looking Ahead

5.12 References, Further Reading

Table of Contents (9)

- ▶ Chap. 6: Partial Redundancy Elimination
 - 6.1 Motivation
 - 6.2 PRE: Essence and Objectives
 - 6.3 The Groundbreaking PRE Algorithm of Morel/Renvoise
 - 6.4 References, Further Reading
- ▶ Chap. 7: Busy Code Motion
 - 7.1 Preliminaries, Problem Definition
 - 7.1.1 Code Motion
 - 7.1.2 Admissible Code Motion
 - 7.1.3 Computationally Optimal Code Motion
 - 7.2 The *BCM* Transformation
 - 7.3 Up-Safety and Down-Safety: The DFA Specifications
 - 7.4 Illustrating Example
 - 7.5 References, Further Reading

Table of Contents (10)

- ▶ Chap. 8: Lazy Code Motion
 - 8.1 Preliminaries, Problem Definition
 - 8.1.1 Lifetime Ranges
 - 8.1.2 Almost Lifetime Optimal Code Motion
 - 8.1.3 Lifetime Optimal Code Motion
 - 8.2 The *ALCM* Transformation
 - 8.3 The *LCM* Transformation
 - 8.4 Delayability and Isolation: The DFA Specifications
 - 8.5 Illustrating Example
 - 8.6 References, Further Reading
- ▶ Chap. 9: Sparse Code Motion
 - 9.1 Background and Motivation
 - 9.1.1 The Embedded Systems Market
 - 9.2 Running Example
 - 9.3 Code-size Sensitive Code Motion
 - 9.3.1 Graph-theoretical Preliminaries
 - 9.3.2 Modelling the Problem
 - 9.3.3 Main Results: Correctness and Optimality

Table of Contents (11)

- ▶ Chap. 9: Sparse Code Motion (cont'd)
 - 9.4 The *SpCM* Transformation
 - 9.5 The Cookbook: Recipes for Code Motion
 - 9.6 Illustrating Example
 - 9.7 References, Further Reading
- ▶ Chap. 10: Code Motion: Summary, Looking Ahead
 - 10.1 Summary: Roots and Relevance of Code Motion
 - 10.1.1 On the Roots and History of Code Motion
 - 10.1.2 On the Relevance of Code Motion
 - 10.2 Looking Ahead: Value Numbering
 - 10.2.1 (Local) Value Numbering
 - 10.2.2 Global Value Numbering: Semantic Code Motion
 - 10.3 Looking Ahead: Challenges and Pitfalls
 - 10.3.1 The Impact of Moving or Placing Code
 - 10.3.2 The Impact of Interacting Transformations
 - 10.3.3 The Impact of Paradigm Shifts
 - 10.4 References, Further Reading

Table of Contents (12)

Part III: Interprocedural Data Flow Analysis

- ▶ Chap. 11: Foundations, the Interproc. DFA Framework
 - 11.1 Preliminaries, the Setting
 - 11.2 IDFA Specifications, IDFA Problems
 - 11.3 Naive Interprocedural DFA
 - 11.4 The *IMOP* Approach
 - 11.5 The *IMaxFP* Approach
 - 11.5.1 In Preparation: The Functional Approach of DFA
 - 11.5.2 The *IMaxFP* Solution
 - 11.6 The Generic Fixed Point Algorithms
 - 11.6.1 Basic Algorithms: Plain Vanilla
 - 11.6.2 Enhanced Algorithms: Improving Performance
 - 11.6.3 Termination
 - 11.7 Safety and Coincidence
 - 11.8 Soundness and Completeness
 - 11.9 A Uniform Framework and Toolkit View
 - 11.10 Applications
 - 11.11 References, Further Reading

Table of Contents (13)

- ▶ Chap. 12: The Functional Approach
 - 12.1 Value Parameters, Local Variables
 - 12.1.1 IDFA_{Stk} Specifications, IDFA_{Stk} Problems
 - 12.1.2 The *IMOP*_{Stk} Approach
 - 12.1.3 The *IMaxFP*_{Stk} Approach
 - 12.1.4 Safety and Coincidence
 - 12.1.5 The Generic Fixed Point Algorithms
 - 12.1.6 Soundness and Completeness
 - 12.1.7 A Uniform Framework and Toolkit View
 - 12.2 Procedural Parameters
 - 12.3 Reference Parameters
 - 12.4 Static Procedure Nesting
 - 12.5 Applications
 - 12.5.1 Interprocedural Availability
 - 12.5.2 Interprocedural Copy Constants
 - 12.6 Summary, Looking Ahead
 - 12.7 References, Further Reading

Table of Contents (14)

- ▶ Chap. 13: The Context Information Approach
 - 13.1 Preliminaries, the Setting
 - 13.1.1 Naive Interprocedural DFA
 - 13.1.2 Interprocedurally Valid and Complete Paths
 - 13.2 *MVP* Approach and *MVP* Solution
 - 13.3 Call Strings, Assumption Sets
 - 13.3.1 Call Strings
 - 13.3.2 Assumption Sets
 - 13.3.3 Advanced Topics
 - 13.4 The Cloning-based Approach
 - 13.5 References, Further Reading

Part IV: Extensions, Other Settings

- ▶ Chap. 14: Alias Analysis
 - 14.1 Sources of Aliasing
 - 14.2 Relevance of Aliasing for Program Optimization
 - 14.3 Shape Analysis
 - 14.4 References, Further Reading

Table of Contents (15)

- ▶ Chap. 15: Optimizations for Object-Oriented Languages
 - 15.1 Object Layout and Method Invocation
 - 15.1.1 Single Inheritance
 - 15.1.2 Multiple Inheritance
 - 15.2 Devirtualization of Method Invocations
 - 15.2.1 Class Hierarchy Analysis
 - 15.2.2 Rapid Type Analysis
 - 15.2.3 Inlining
 - 15.3 Escape Analysis
 - 15.3.1 Connection Graphs
 - 15.3.2 Intraprocedural Setting
 - 15.3.3 Interprocedural Setting
 - 15.4 References, Further Reading

Table of Contents (16)

Part V: Conclusions, Perspectives

- ▶ Chap. 16: Conclusions, Emerging and Future Trends
 - 16.1 Reconsidering Optimization
 - 16.2 Summary, Looking Ahead
 - 16.3 References, Further Reading
- ▶ References

Appendices

- ▶ A Mathematical Foundations
- ▶ B Pragmatics: Variants of Flow Graphs
- ▶ C Implementing Busy and Lazy Code Motion
- ▶ D Lazy Strength Reduction

Table of Contents (17)

▶ A Mathematical Foundations

A.1 Relations

A.2 Ordered Sets

A.2.1 Pre-Order, Partial Orders, and More

A.2.2 Hasse Diagrams

A.2.3 Bounds and Extremal Elements

A.2.4 Noetherian Orders and Artinian Orders

A.2.5 Chains

A.2.6 Directed Sets

A.2.7 Maps on Partial Orders

A.2.8 Order Homomorphisms, Order Isomorphisms

A.3 Complete Partially Ordered Sets

A.3.1 Chain and Directly Complete Partial Orders

A.3.2 Maps on Complete Partial Orders

A.3.3 Mechanisms for Constructing Complete Partial Orders

Table of Contents (18)

- ▶ A Mathematical Foundations (cont'd)
 - A.4 Lattices
 - A.4.1 Lattices, Complete Lattices
 - A.4.2 Distributive, Additive Maps on Lattices
 - A.4.2 Lattice Homomorphisms, Lattice Isomorphisms
 - A.4.3 Modular, Distributive, and Boolean Lattices
 - A.4.4 Mechanisms for Constructing Lattices
 - A.4.5 Order-theoretic and Algebraic View of Lattices
 - A.5 Fixed Point Theorems
 - A.5.1 Fixed Points, Towers
 - A.5.2 Fixed Point Theorems for Complete Partial Orders
 - A.5.3 Fixed Point Theorems for Lattices
 - A.6 Fixed Point Induction
 - A.7 Completions, Embeddings
 - A.7.1 Downsets
 - A.7.2 Ideal Completion: Embedding of Lattices
 - A.7.3 Cut Completion: Embedding of Partial Orders and Latt.
 - A.7.4 Downset Completion: Embedding of Partial Orders
 - A.8 References, Further Reading

Table of Contents (19)

- ▶ B Pragmatics: Variants of Flow Graphs
 - B.1 Motivation
 - B.1.1 Flow Graph Variants
 - B.1.2 Flow Graph Variants: Which one to Select?
 - B.2 *MOP* and *MaxFP* Approach
 - B.2.1 Edge-labelled Instruction Graphs
 - B.2.2 Node-labelled Basic Block Graphs
 - B.3 Available Expressions
 - B.3.1 Node-labelled Basic Block Graphs
 - B.3.2 Node-labelled Instruction Graphs
 - B.3.3 Edge-labelled Instruction Graphs
 - B.3.4 Summary of Findings
 - B.4 Simple Constants
 - B.4.1 Edge-labelled Instruction Graphs
 - B.4.2 Node-labelled Basic Block Graphs
 - B.5 Faint Variables
 - B.6 Conclusions
 - B.7 References, Further Reading

Table of Contents (20)

- ▶ C Implementing Busy and Lazy Code Motion
 - C.1 Implementing BCM and LCM on SI Graphs
 - C.1.1 Preliminaries
 - C.1.2 Implementing BCM_{ι}
 - C.1.3 Implementing LCM_{ι}
 - C.2 Implementing BCM and LCM on BB Graphs
 - C.2.1 Preliminaries
 - C.2.2 Implementing BCM_{β}
 - C.2.3 Implementing LCM_{β}
 - C.3 Illustrating Example
 - C.4 References, Further Reading

Table of Contents (21)

► D Lazy Strength Reduction

D.1 Motivation

D.2 Running Example

D.3 Preliminaries

D.4 Extending *BCM* to Strength Reduction: The *BSR* Transformation

D.5 The 1st Refinement: Avoiding Multiplication-Addition Sequences – The BSR_{FstRef} Transformation

D.6 The 2nd Refinement: Avoiding Unnecessary Register Pressure – The BSR_{SndRef} Transformation

D.7 The 3rd Refinement: Avoiding Multiple-Addition Sequences – The $(BSR_{ThdRef} \equiv) LSR$ Transformation

D.8 References, Further Reading

Part I

Introduction

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

24/1857

Chapter 1

Motivation

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

25/1857

Chapter 1.1

Setting the Scene

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

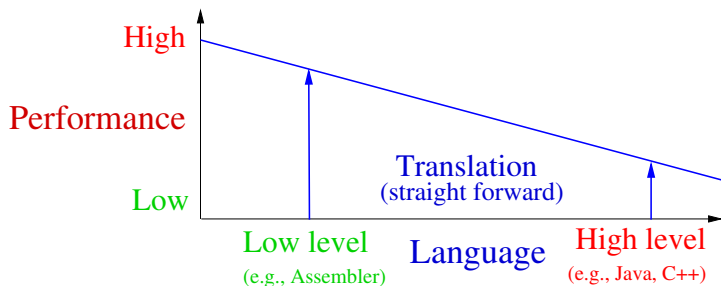
Chap. 9

Chap. 10

Part III

26/1857

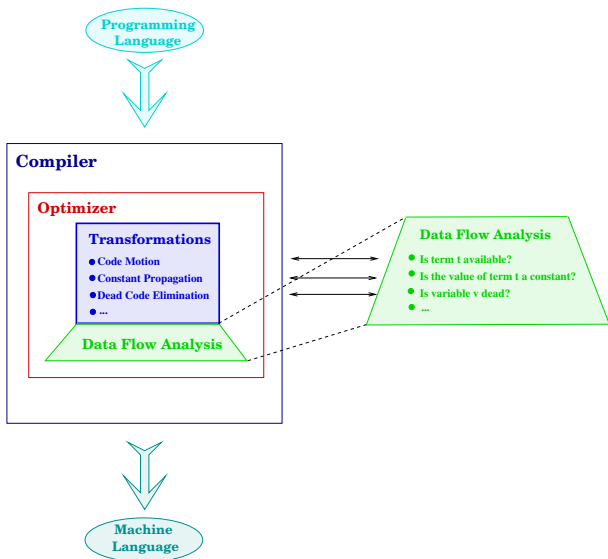
Languages and Their Perceived Performance



Common perception

- ▶ High level languages/abstractions give low level of performance.

The Optimizing Compiler to the Rescue



Translation vs. Optimizing Compilation

Translation (straight forward)

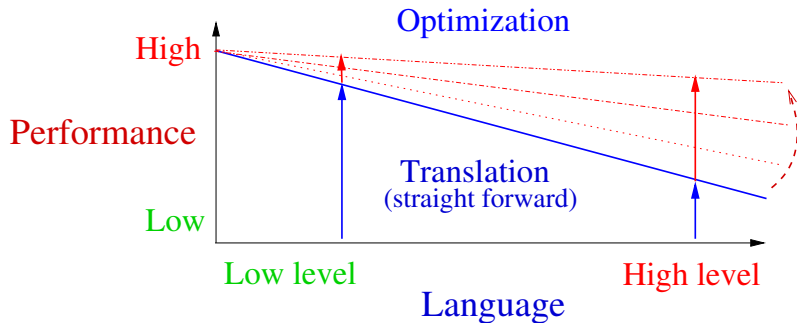
- ▶ preserves semantics but does not exploit specific opportunities of lower level languages with respect to performance.

Optimizing compilation (optimization)

- ▶ is performance-aware and strives to improve performance in the course of translation.

Optimizing Compilation (Optimization)

...aims at closing the performance gap.



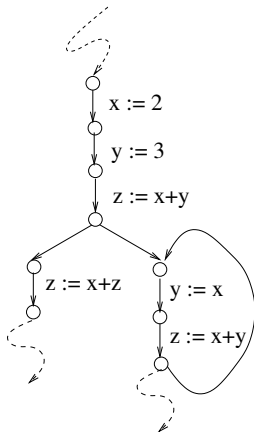
Selected Common Optimizations

...and optimization sequences for illustration and motivation.

Optimization	Before	After
Algebraic Simplification	x^0	1
Function Inlining	<code>y=inc(x);</code>	<code>y=x+1;</code>
Dead Code Elimination (DCE)	<code>b=0; if(b>0) x=x+1;</code> <code>y=f(x);</code>	<code>b=0; y=f(x);</code>
Constant Propagation (CP)	<code>x=21; y=2*x;</code>	<code>x=21; y=2*21;</code>
CP + DCE	<code>x=21; y=2*x;</code>	<code>y=2*21;</code>
Constant Folding (CF)	<code>y=2*21;</code>	<code>y=42;</code>
CP + DCE + CF	<code>x=21; y=2*x;</code>	<code>y=42;</code>
Copy Propagation (CpP)	<code>x=y; ...; z=x;</code>	<code>x=y; ...; z=y;</code>
CpP + DCE	<code>x=y; ...; z=x;</code>	<code>...; z=y;</code>
Code Motion	<code>if(b>0) {x=a+b;y=f(x)}</code> <code>else x=a+b;</code>	<code>x=a+b;</code> <code>if(b>0) y=f(x);</code>

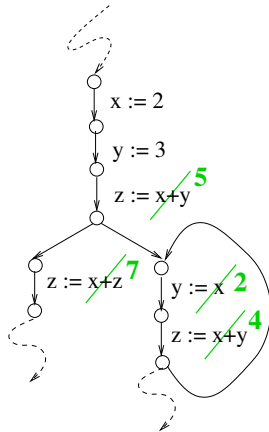
Constant Propagation: Orig. & Opt. Program

a)



Original program

b)



After simple constant propagation

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

32/1857

Typical Optimization Aspects

...avoid

- ▶ **redundant computations**
 - ▶ reuse available results
 - ▶ move loop invariant computations outside loops
 - ▶ ...
- ▶ **superfluous computations**
 - ▶ results known not to be needed
 - ▶ results known already at compile time
 - ▶ ...
- ▶ **costly computations**
 - ▶ results can be computed less costly
 - ▶ ...
- ▶ ...

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

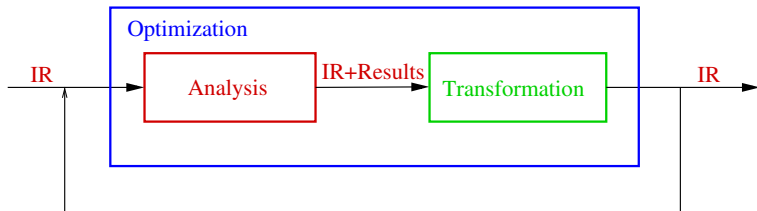
Chap. 9

Chap. 10

Part III

33/1857

Optimization Considered Schematically

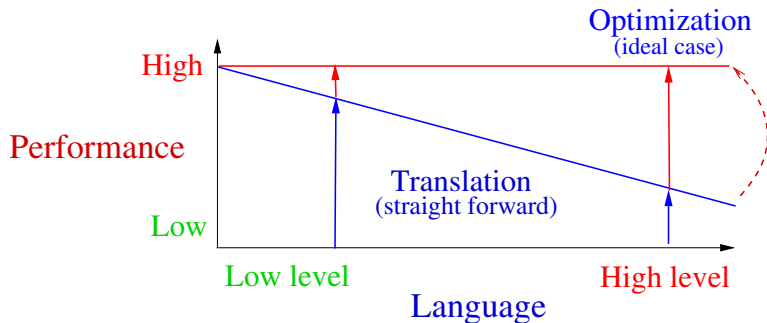


Optimization is a (repeatedly applied) two-stage process consisting of

- ▶ **Analysis**
 - ▶ determines properties of program
 - ▶ safe, pessimistic assumptions
- ▶ **Transformation**
 - ▶ based on analysis results

Optimization: The Ideal Case

...the performance gap is closed.



Note: The term **optimization** is a **misnomer**: usually we do not achieve an 'optimal' solution – but it is the ideal case).

Chapter 1.2

An Extensive Illustrating Example

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

36/1857

The Running Example

...adding two 2-dimensional matrices:

```
int a[m][n], b[m][n], c[m][n];
...
for(int i=0; i<m; ++i) {
    for(int j=0; j<n; ++j) {
        a[i][j]=b[i][j]+c[i][j];
    }
}
```

Note: There are no obvious optimizing/improving transformations recommending themselves for application.

Step 1: Lowering the High-level Code to IR

...revealing the [address computation](#):

```
i=0;
while(i<m) {
    j=0;
    while(j<n) {
        temp=Base(a)+i*n+j;
        *(temp)=*(Base(b)+i*n+j)+*(Base(c)+i*n+j);
        j=j+1;
    }
    i=i+1;
}
```

Note: Lowering the high-level code to intermediate-level code reveals the address computation for array accesses which enables several optimizing/improving program transformations.

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

38/1857

Step 2: Optimizations on IR

1. **Analysis:** Available expressions analysis
 \rightsquigarrow **Transformation:** Common subexpression elimination
2. **Analysis:** Loop invariants detection
 \rightsquigarrow **Transformation:** Loop invariant code motion
3. **Analysis:** Induction variables detection
 \rightsquigarrow **Transformation:** Strength reduction
4. **Analysis:** Copy analysis
 \rightsquigarrow **Transformation:** Copy propagation
5. **Analysis:** Dead variables analysis
 \rightsquigarrow **Transformation:** Dead code elimination
6. **Analysis:** LFTR candidates detection
 \rightsquigarrow **Transformation:** Linear function test replacement

1st Analysis: Available Expressions Analysis

...determines for each program point, which expression **must** have already been computed, and not later modified, on all paths to the program point.

```
i=0;
while(i<m) {
    j=0;
    while(j<n) {
        temp = (Base(a)+i*n+j);
        *temp = *(Base(b)+i*n+j) + *(Base(c)+i*n+j);
        j=j+1;
    }
    i=i+1;
}
```


1st Opt.: Common Subexpression Elimination

- ▶ **Analysis:** Available expressions analysis
- ▶ **Transformation:** Eliminate recomputations of $i*n+j$
 - ▶ Introduce $t1=i*n+j$
 - ▶ Use $t1$ instead of $i*n+j$

```
i=0;
while(i<m) {
    j=0;
    while(j<n) {
        temp = (Base(a)+ i*n+j);
        *temp = *(Base(b)+ i*n+j)
            + *(Base(c)+ i*n+j);
        j=j+1;
    }
    i=i+1;
}
```

```
i=0;
while(i<m) {
    j=0;
    while(j<n) {
        t1=i*n+j;
        temp = (Base(a)+ t1);
        *temp = *(Base(b)+ t1)
            + *(Base(c)+ t1);
        j=j+1;
    }
    i=i+1;
}
```

2nd Analysis: Loop Invariants Detection

...a **loop invariant** is an expression that is always computed to the same value in each iteration of the loop.

```
i=0;
while(i<m) {
    j=0;
    while(j<n) {
        t1=i*n+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1) + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
}
```

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

42/1857

2nd Opt.: Loop Invariant Code Motion

- ▶ **Analysis:** Loop invariants detection
- ▶ **Transformation:** Move loop invariant outside loop
 - ▶ Introduce `t2=i*n` and replace `i*n` by `t2`
 - ▶ Move `t2=i*n` outside loop

```
i=0;
while(i<m) {
    j=0;

    while(j<n) {
        t1=i*n+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
}
```

```
i=0;
while(i<m) {
    j=0;
    t2=i*n;
    while(j<n) {
        t1=t2+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
}
```

3rd Analysis: Induction Variables Detection

```
i=0;
while(i<m) {
    j=0;
    t2=i*n;
    while(j<n) {
        t1=t2+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
}
```

Basic Induction Variables

- ▶ Variables v whose only definitions within a loop are of the form $v = v + c$ or $v = v - c$ and c is a loop invariant.

Derived Induction Variables

- ▶ Variables w defined only once in a loop whose value is a linear function of some basic induction variable.

3rd Optimization: Strength Reduction (1)

...replaces a repeated series of **expensive ('strong') operations** with a series of **inexpensive ('weak') operations** that compute the same values.

Classical example:

- ▶ Replacing integer **multiplications** based on a loop index with equivalent **additions**.

Note: This particular case arises routinely from expansion of array and structure addresses in loops.

3rd Optimization: Strength Reduction (2)

- ▶ **Analysis:** Induction variables (IVs) detection
- ▶ **Transformation:** Move multiplications outside of loop
 - ▶ Introduce `t3=i*n` before the loop, replace `i*n` by `t3`
 - ▶ Add `t3=t3+i*c` at every update site of `i`

```
i=0;

while(i<m) {
    j=0;
    t2=i*n;
    while(j<n) {
        t1=t2+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
}
```

```
i=0;
t3=0;
while(i<m) {
    j=0;
    t2=t3;
    while(j<n) {
        t1=t2+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

4th Analysis: Copy Analysis

...determines for each program point the **copy statements** $x = y$ that still are relevant (i.e., neither x nor y have been redefined) when control reaches that point.

```
i=0;
t3=0;
while(i<m) {
    j=0;
    t2=t3;
    while(j<n) {
        t1=t2+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

4th Optimization: Copy Propagation

- ▶ **Analysis:** Copy analysis and def-use chains computation (ensure only one definition reaches the use of x)
- ▶ **Transformation:** Replace the use of x by y

```
i=0;
t3=0;
while(i<m) {
    j=0;
    t2=t3;
    while(j<n) {
        t1=t2+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

```
i=0;
t3=0;
while(i<m) {
    j=0;
    t2=t3;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```


5th Analysis: Dead Variables Analysis (1)

A **variable** is

- ▶ **live** at a program point if there is a path from this program point to a use of the variable that does not re-define the variable.
- ▶ **dead** at a program point, if it is not live at that point.

A **live (dead) variables analysis**

- ▶ determines for each program point, which variable **may be live (is dead)** at the exit from that point.

5th Analysis: Dead Variables Analysis (2)

```
i=0;
t3=0;
while(i<m) {
    j=0;
    t2=t3;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

► Only **dead** variables are marked.

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

50/1857

5th Optimization: Dead Code Elimination

- ▶ **Analysis:** Dead variables analysis
- ▶ **Transformation:** Remove all assignments to dead variables

```
i=0;
t3=0;
while(i<m) {
    j=0;
    t2=t3;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

```
i=0;
t3=0;
while(i<m) {
    j=0;
    ;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

6th Analysis: LFTR Candidates Detection

...determines so-called *LFTR candidates*: these are **IVs** that are only used in the loop-closing test, and can be replaced by other **IVs**, i.e., by *linear function expressions* on these **IVs**.

```
i=0;
t3=0;
while( i<m ) {
    j=0;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

52/1857

6th Opt.: Linear Function Test Replacement

- ▶ **Analysis:** Determine IVs that are only used in the loop-closing test, and can be replaced by other IVs
- ▶ **Transformation:** Remove all assignments to replaceable IVs and insert compensation code (LFTR)

```
i=0;
t3=0;
while( i<m ) {
    j=0;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    i=i+1;
    t3=t3+n;
}
```

```

t3=0;
t4=n*m;
while( t3<t4 ) {
    j=0;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    t3=t3+n;
}
```

Summary of Optimizations

...applied in this [example](#).

Analyses	Transformations
Available expr. analysis Loop invariants detection Induction variables detection Copy analysis Dead variables analysis LFTR candidates detection	Common subexpr. elimination Loop invariant code motion Strength reduction Copy propagation Dead code elimination Linear Function Test Repl.

The Application of Many Optimizations

...as in the preceding example is quite **typical in practice**:

“Compiler optimisations are like bullets.
Each bullet is ineffective for many programs;
but each gives a big payoff for a few programs
whose inner loop it strikes.
Good compilers simply deploy a hail of bullets,
so that few programs will survive unoptimised.”

Clement A. Baker-Finch, Kevin Glynn, Seymon Peyton Jones

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

55/1857

Challenging Problem

...the order in which to apply the various optimizations:

Some **optimizations**

- ▶ **are independent** of each other.
- ▶ **enable** another optimization.
- ▶ **prevent** another optimization.

Are there further Optimizations?

In fact, there is a plethora of further **optimizations**, i.e., pairs of **analyses** and **transformations**, including but not limited to:

- ▶ Optimizations for
 - ▶ imperative languages (procedural, object-oriented)
 - ▶ declarative languages (logical, functional)
 - ▶ parallel, distributed languages
 - ▶ ...
- ▶ Array analyses, optimizations
- ▶ Pointer/alias/shape analyses, optimizations
- ▶ Heap analyses, garbage collection
- ▶ ...

Note

In the preceding example

- ▶ all optimizations could have been done by the programmer, too.

This, however,

- ▶ would lead to the loss of all advantages and benefits of using programming abstractions offered by high-level languages, and effectively enforce programming on an intermediate code level.

Requiring and insisting on it

- ▶ would put an undue burden onto programmers, reduce their productivity, and be highly error-prone.

For Illustration

...compare the **initial** and the **final program** of the **running example** for adding two matrices:

```
int a[m][n], b[m][n], c[m][n];
...
for(int i=0; i<m; ++i) {
    for(int j=0; j<n; ++j) {
        a[i][j]=b[i][j]+c[i][j];
    }
}

t3=0;
t4=n*m;
while(t3<t4) {
    j=0;
    while(j<n) {
        t1=t3+j;
        temp = (Base(a)+t1);
        *temp = *(Base(b)+t1)
            + *(Base(c)+t1);
        j=j+1;
    }
    t3=t3+n;
}
```

Key Questions

Would you prefer programming matrix addition

- ▶ dealing with low-level details yourself as shown on the right-hand side

...or by

- ▶ taking advantage of the abstraction of 2-dimensional arrays offered by high-level languages as shown on the left-hand side?

Most likely

...you would even prefer programming matrix addition

- ▶ using an even higher language offering abstractions allowing you to write

```
int a[m][n], b[m][n], c[m][n];  
a=b+c;
```

As a matter of fact:

- ▶ Optimizing compilation is the key to render this possible!

Chapter 1.3

The Impact of Optimization: A Case Study

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

62/1857

Case Study: C++STL Code Optimization

...on the **impact** of **programming style** and **optimization** on **performance**:

- ▶ Different programming styles for iterating on a container and performing operation on each element.
- ▶ Use different levels of abstractions for iteration, container, and operation on elements.
- ▶ Optimization levels **O1-3** compared with **GNU 4.0** compiler.

Acknowledgement: This study is joint work of Markus Schordan and Rene Heinzl.

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

63/1857

Case Study Setting

We iterate on container 'mycontainer' and perform an operation on each element.

- ▶ Container is a `vector`.
- ▶ Elements are of type `numeric_type` (`double`).
- ▶ Operation of `adding 1` is applied to each element.
- ▶ Evaluation Cases `EC1` thru `EC6`.

Programming Styles 1, 2: EC1, EC2

EC1: Imperative Programming

```
for (unsigned int i = 0; i < mycontainer.size(); ++i)
{
    mycontainer[i] += 1.0;
}
```

EC2: Weakly Generic Programming

```
for (vector<numeric_type>::iterator
    it = mycontainer.begin();
    it != mycontainer.end();
    ++it)
{
    *it += 1.0;
}
```

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

65/1857

Programming Style 3: EC3

EC3: Generic Programming

```
for_each(mycontainer.begin(),
         mycontainer.end(),
         plus_n<numeric_type>(1.0) );
```

Functor

```
template<class datatype>
struct plus_n
{
    plus_n(datatype member):member(member) {}
    void operator()(datatype& value) {
        value += member;
    }
private:
    datatype member;
};
```

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

66/1857

Programming Style 4: EC4

EC4: Functional Programming with STL

```
transform(mycontainer.begin(),  
         mycontainer.end(),  
         mycontainer.begin(),  
         bind2nd(std::plus<numeric_type>(), 1.0));
```

- ▶ **plus**: binary function object that returns the result of adding its first and second arguments.
- ▶ **bind2nd**: Templated utility for binding values to function objects.

Programming Styles 5, 6: EC5, EC6

EC5: Functional Programming with Boost::lambda

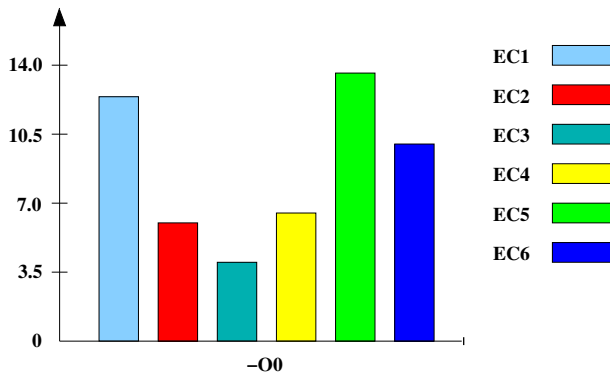
```
std::for_each(mycontainer.begin(),  
              mycontainer.end(),  
              boost::lambda::_1 += 1.0 );
```

EC6: Functional Programming with Boost::phoenix

```
std::for_each(mycontainer.begin(),  
              mycontainer.end(),  
              phoenix::arg1 += 1.0 );
```

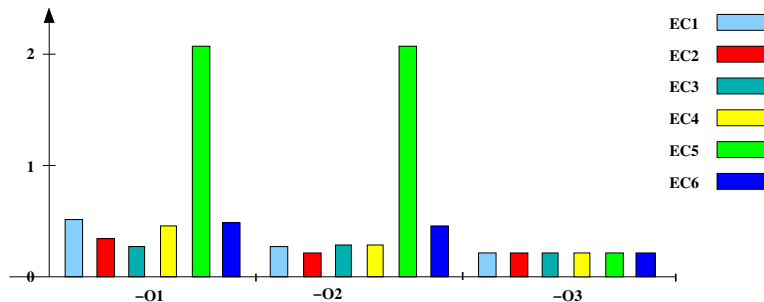
- ▶ Use of `unnamed function object`.

Evaluation: EC1-6 w/out optimization



- ▶ Compiler: GNU g++ 4.0
- ▶ Evaluation Cases: EC1 thru EC6.
- ▶ Container size: 1,000.
- ▶ Time measured in milliseconds.

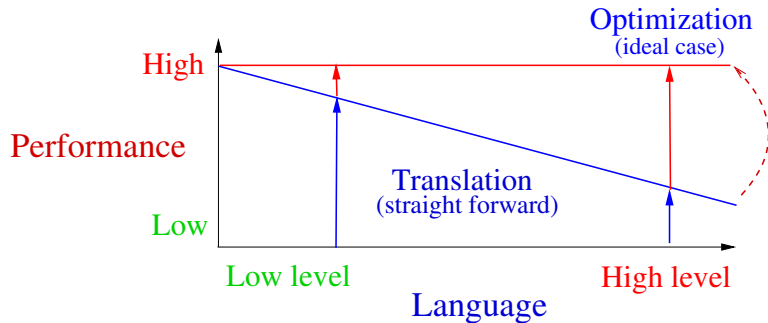
Evaluation: EC1-6 w/ optimization levels O1-3



- ▶ Compiler: GNU g++ 4.0
- ▶ The actual run-time with different optimization levels -01, -02, -03 for each programming style EC1-6.
- ▶ An almost identical run-time is achieved at level -03.

In this Case Study

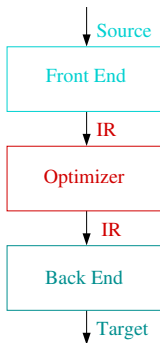
...the performance gap is closed!



Chapter 1.4

Compilers, Optimizing Compilers, and their Structure

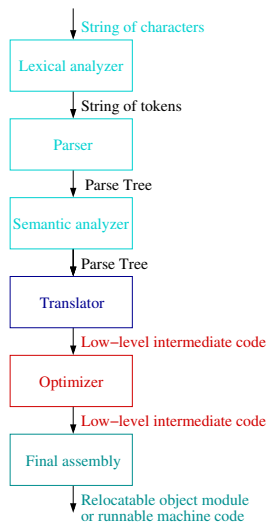
Generic Structure of an Optimizing Compiler



Goal of code optimization

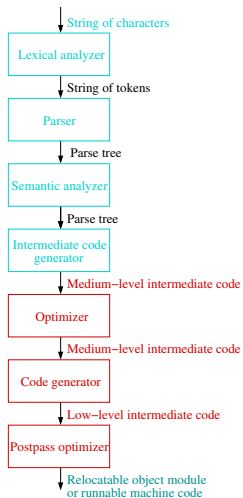
- ▶ Discover, **at compile-time**, information about the run-time behavior of the program and use that information **to improve the code generated by the compiler.**

Model of a Low Level Optimizer



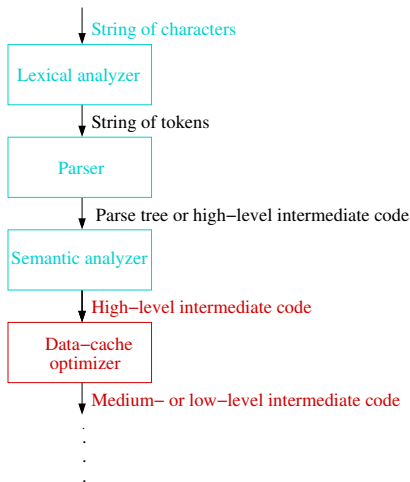
- All optimization is done on a **low level intermediate code**.

Model of a Mixed Level Optimizer



- Optimization is divided into two phases, one operating on a **medium level** and one on a **low level**.

Model of a High Level Cache Optimizer



Adding data-cache optimization to an optimizing compiler

- ▶ Data-cache optimizations are most effective when applied to a high-level intermediate form.

Examples

▶ High-Level optimizations

- ▶ IBM's PowerPC compiler: first translates to LL code (XIL) and then generates a HL representation (YIL) from it to do data-cache optimization.
- ▶ Source-To-Source Optimizer Tools: Sage++, LLNL-ROSE, JTransformer.

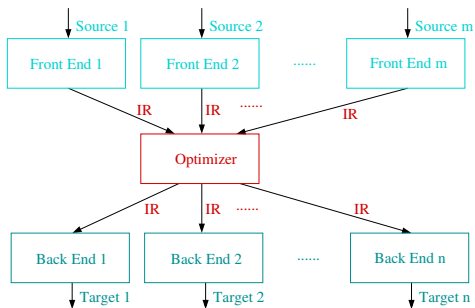
▶ Mixed model

- ▶ Sun Microsystem's compilers for SPARC.
- ▶ Intel's compilers for the 386 architecture family.
- ▶ Silicon Graphic's compilers for MIPS.

▶ Low level model

- ▶ IBM's compilers for PowerPC.
- ▶ Hewlett-Packard's compilers for PA-RISC.

Practice: m-2-n Compilers and Optimizers

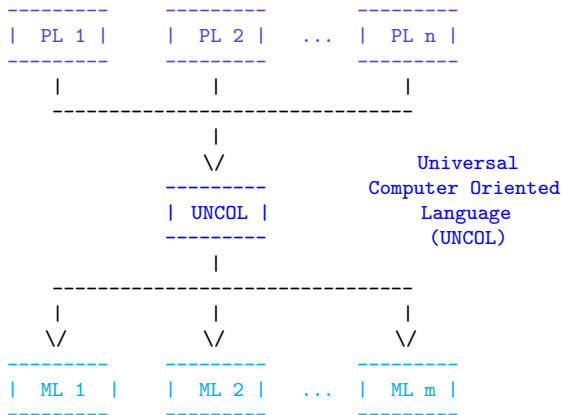


Idea: Decoupling of Compiler Front Ends from Back Ends

- ▶ **Without IR:** *m* source languages, *n* targets $\rightsquigarrow m \times n$ compilers
- ▶ **With IR:** *m* Front Ends, *n* Back Ends
- ▶ **Problem:** Appropriate choice of the level of IR (possible solution: multiple levels of IR)

IR-Decoupling of Compiler Front/Back Ends

...is an application of [Conway's](#) famous UNCOL concept:



- Melvin E. Conway. [Proposal for an UNCOL](#). Communications of the ACM 1(3):5, 1958.

Intermediate Representation (IR)

▶ High level

- ▶ quite close to source language, e.g., abstract syntax tree
- ▶ code generation issues are quite clumsy at high-level
- ▶ adequate for high-level optimizations (cache, loops)

▶ Medium level

- ▶ represent source variables, temporaries, (and registers)
- ▶ reduce control flow to conditional and unconditional branches
- ▶ adequate to perform machine independent optimizations

▶ Low level

- ▶ correspond to target-machine instructions
- ▶ adequate to perform machine dependent optimizations

Chapter 1.5

Optimizations: Objectives and Categorization

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

81/1857

Optimizations: Kinds and Objectives

...different kinds of optimizations for different purposes, e.g.:

- ▶ **Speed**
 - ▶ Speeding up execution of compiled code (awaiting the next generation of processors is not always a viable option)
- ▶ **Size**
 - ▶ of compiled code when committed to read-only memory where size is an economic constraint
 - ▶ or code is transmitted over a limited-bandwidth communications channel
- ▶ **Response**
 - ▶ to real-time events when dealing with (safety-critical) real-time systems: worst-case execution time (WCET) analysis and optimization
- ▶ **Energy consumption**
- ▶ **Parallelization**
- ▶ ...

Considerations for Optimization

- ▶ Safety
 - ▶ **correctness**: generated code must have the same meaning as the input code
 - ▶ **meaning**: is the observable behavior of the program
- ▶ Profitability
 - ▶ improvement of code
 - ▶ trade offs between different kinds of optimizations
- ▶ Problems
 - ▶ reading past array bounds, pointer arithmetics, etc.

Scope of Optimization (1)

Local

- ▶ Expressions
 - ▶ optimal code generation for expressions
- ▶ Basic blocks
 - ▶ statements are executed sequentially
 - ▶ if any statement is executed the entire block is executed
 - ▶ limited to improvements that involve operations that all occur in the same block

Scope of Optimization (2)

Global

- ▶ Intra-procedural (whole procedure)
 - ▶ entire procedure
 - ▶ procedure provides a natural boundary for both analysis and transformation
 - ▶ procedures are abstractions encapsulating and insulating run-time environments
 - ▶ opportunities for improvements that local optimizations do not have
- ▶ Inter-procedural (whole program)
 - ▶ entire program
 - ▶ exposes new opportunities but also new challenges
 - ▶ name-scoping
 - ▶ parameter binding
 - ▶ virtual methods
 - ▶ recursive methods (number of variables?)
 - ▶ **scalability to program size**

Optimization Taxonomy

Optimizations are categorized by the effect they have on the code.

▶ Machine independent

- ▶ largely ignore the details of the target machine
- ▶ in many cases profitability of a transformation depends on detailed machine-dependent issues, but those are ignored

▶ Machine dependent

- ▶ explicitly consider details of the target machine
- ▶ many of these transformations fall into the realm of code generation
- ▶ some are within the scope of the optimizer (some cache optimizations, some expose instruction level parallelism)

Machine Independent Optimizations (1)

▶ Dead code elimination

- ▶ eliminate useless or unreachable code
- ▶ algebraic identities
- ▶ e.g., dead variables elimination, faint variables elimination

▶ Code motion

- ▶ move operation to place where it executes less frequently
- ▶ e.g., loop invariant code motion, hoisting, constant propagation

▶ Specialize

- ▶ to specific context in which an operation will execute
- ▶ e.g., operator strength reduction, constant propagation, peep-hole optimization

Machine Independent Optimizations (2)

- ▶ Eliminate redundancy
 - ▶ replace redundant computation with a reference to previously computed value
 - ▶ e.g., common subexpression elimination, value numbering
- ▶ Enable other transformations
 - ▶ rearrange code to expose more opportunities for other transformations
 - ▶ e.g., inlining, cloning

Machine Dependent Optimizations

- ▶ Take advantage of special hardware features
 - ▶ Instruction selection
- ▶ Manage or hide latency
 - ▶ Arrange final code in a way that hides the latency of some operations
 - ▶ Instruction scheduling
- ▶ Manage bounded machine resources
 - ▶ Registers, functional units, cache memory, main memory

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

89/1857

Chapter 1.6

Tools for Compiler Construction and Optimization

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

90/1857

Compilers, Compiler Writing Tools

...dictionaries on [on-line](#) available [resources](#):



German National Research Center for Information Technology, Fraunhofer Institute for Computer Architecture and Software Technology. *The Catalog of Compiler Construction Tools*, 1996-2006.

<http://catalog.compilertools.net/>



Compilers.net Team. *Search Machine on Compilers and Programming Languages, Directory of Compiler and Language Resources*, 1997-2007.

<http://www.compilers.net>

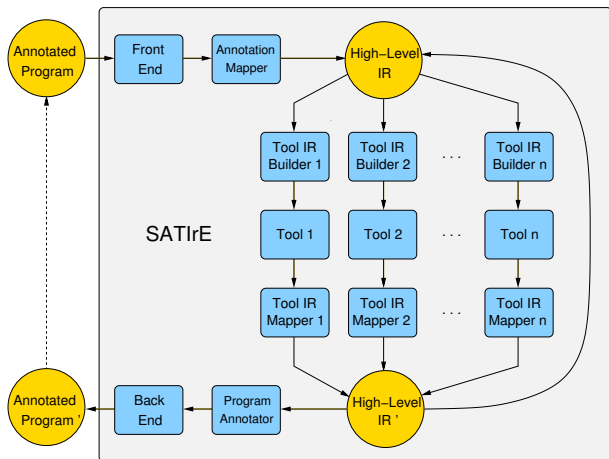
In this course

...we will **focus** on:

- ▶ **LLNL-ROSE: Source-to-Source C/C++ Optimization Framework**, Lawrence Livermore National Laboratory (LLNL), CA, USA, <http://rosecompiler.org/>
- ▶ **SATIrE: Static Analysis and Tool Integration Engine**, TU Vienna, Austria, <http://www.complang.tuwien.ac.at/satire/>
- ▶ **PAG: Program Analysis Generator**, AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany, <https://www.absint.com/pag/index.htm>

SATIrE: Abstract Architecture

Static Analysis and Tool Integration Engine (SATIrE)



Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

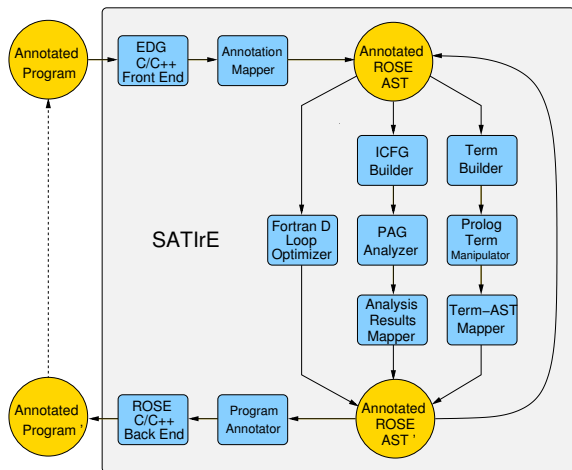
Chap. 9

Chap. 10

Part III

93/1857

SATIrE: Concrete Architecture



(Oct'07)

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

94/1857

SATIrE Components (1)

A set of **basic components**:

- ▶ **C/C++ Front End**: Edison Design Group.
- ▶ **Annotation Mapper**: maps source-code annotations to an accessible representation in the ROSE-AST.
- ▶ **Program Annotator**: annotates programs with analysis results; combined with the **Annotation Mapper** this allows to make analysis results persistent in source-code for subsequent analysis and optimization.
- ▶ **C/C++ Back End**: generates C++ code from ROSE-AST.

Anm.: **AST** reminds to **Abstract Syntax Tree**.

SATIrE Components (2)

A set of added, integrated components:

- ▶ **Integration 1: Loop Optimizer (Rice University, LLNL)**
 - ▶ **Loop Optimizer:** ported from the Fortran D compiler and integrated in LLNL-ROSE.
- ▶ **Integration 2: PAG (Saarland University, AbsInt GmbH, Saarbrücken)**
 - ▶ **ICFG Builder:** Interprocedural Control Flow Graph Generator, addresses full C++.
 - ▶ **PAG Analyzer:** a program analyzer, generated with AbsInt's Program Analysis Generator (PAG) from a user-specified program analysis.
 - ▶ **Analysis Results Mapper:** maps analysis results from ICFG back to ROSE-AST, makes them available as AST-attributes.

SATIrE Components (3)

- ▶ Integration 3: Termite (TU Vienna)
 - ▶ **Term Builder**: generates an external textual term representation of the ROSE-AST (Term is in Prolog syntax).
 - ▶ **Term-AST Mapper**: parses the external textual program representation and translates it into a ROSE-AST.

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

97/1857

Chapter 1.7

Summary, Looking Ahead

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

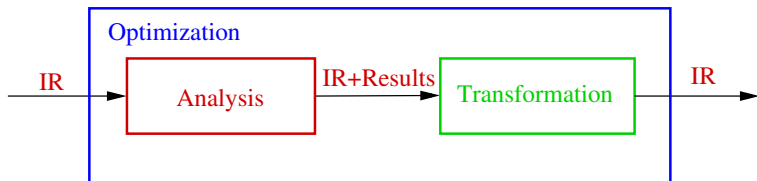
Chap. 9

Chap. 10

Part III

98/1857

Optimization: The General Schema



Optimization, a combination of

- ▶ **Analysis**

- ▶ determines properties of program.
- ▶ relies on safe, pessimistic assumptions.

- ▶ **Transformation**

- ▶ based on analysis results.
- ▶ must preserve the program semantics, i.e., the observable program behaviour.

Program Analysis: The Essence (1)

...offers techniques for predicting statically at compile-time **safe** and **efficient approximations** to the set of configurations or behaviors arising dynamically at run-time.

- ▶ **Safe:** Faithful to the semantics
- ▶ **Efficient:** Implementation with
 - ▶ good time performance
 - ▶ low space consumption

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

100/185

Program Analysis: The Essence (2)

Important approaches for **program analysis**:

- ▶ Data Flow Analysis
- ▶ Abstract Interpretation
- ▶ Model Checking
- ▶ Symbolic Analysis, Symbolic Execution
- ▶ Theorem Proving
- ▶ Integer Linear Programming
- ▶ Graph Theory, Graph Algorithms
- ▶ ...

...for many of these approaches we will see examples in the course of the lecture.

Assessing the Power/Success of Optimization

...via **validation** and/or **verification**.

Validation

- ▶ **Experimentally**: Benchmark suite(s)
 - ▶ **General purpose suites (ACET-focused)**: SPEC (Standard Performance Evaluation Corporation), Dhrystone, Whetstone,...
 - ▶ **Special purpose suites (WCET-focused)**: TACLe (EU FP7 COST Action 'Timing Analysis on Code Level'), Mälardalen University,...

Verification

- ▶ **Analytically**: Formal program and cost models
 - ▶ Rigorous mathematical proving

Key Issues in Optimization

...and in this lecture: The theory and practice of

Optimal

- ▶ Program Analysis
- ▶ Program Transformation

...and based thereon of:

Optimal Optimization

- ▶ Meaningful terms? If so, what do they mean?
- ▶ Achievable? If so, when and how?
- ▶ If not, how to proceed then?

Chapter 1.8

References, Further Reading

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9




Chap. 10

Part III





104/185

Further Reading for Chapter 1 (1)





Compilers

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 1, Introduction; Chapter 8, Code Generation; Chapter 9, Machine-Independent Optimizations)
-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2. aktualisierte Auflage, 2008. (Kapitel 1, Einleitung; Kapitel 8, Codeerzeugung; Kapitel 9, Maschinenunabhängige Optimierungen)
-  Dick Grune, Kees van Reeuwijk, Henri E. Bal, Criel J.H. Jacobs, Koen G. Langendoen. *Modern Compiler Design*. Springer-V., 2nd edition, 2012. (Chapter 1, Introduction)


Further Reading for Chapter 1 (2)

-  Helmut Seidl, Reinhard Wilhelm, Sebastian Hack. *Compiler Design: Analysis and Transformation*. Springer-V., 2012. (Chapter 1, Foundations and Intraprocedural Optimization)
-  Patrick D. Terry. *Compilers and Compiler Generators: An Introduction with C++*. International Thomson Computer Press, 1997.
-  Patrick D. Terry. *Compiling with C# and Java*. Addison-Wesley, 2005. (Chapter 1, Translators and Languages)
-  William M. Waite, Gerhard Goos. *Compiler Construction*. Springer-V., 1984. (Chapter 1, Introduction and Overview; Chapter 13, Optimization)


Further Reading for Chapter 1 (3)


-  William M. Waite, Lynn R. Carter. *An Introduction to Compiler Construction*. HarperCollins College Publishers, 1993. (Chapter 1, The Characteristics of a Compiler)
-  Reinhard Wilhelm, Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995. (Chapter 1, Introduction)
-  Reinhard Wilhelm, Dieter Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-V., 2. Auflage, 1997. (Kapitel 1, Einleitung)
-  Reinhard Wilhelm, Helmut Seidl. *Compiler Design: Virtual Machines*. Springer-V., 2010. (Chapter 1, Introduction; Chapter 2, Imperative Programming Languages)

Further Reading for Chapter 1 (4)


 Reinhard Wilhelm, Helmut Seidl, Sebastian Hack. *Compiler Design: Syntactic and Semantic Analysis*. Springer-V., 2013. (Chapter 1, The Structure of Compilers)

Optimizing Compilers



 Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 1, Compiler Challenges for High- Performance Architectures)

 Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 1, Overview; Chapter 2, Compiler Structure)




Further Reading for Chapter 1 (5)

-  Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2nd edition, 2008. (Chapter 1, Data Flow Analysis)

Compiler Implementation




-  Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1997. (Chapter 1, Introduction; Chapter 17, Dataflow Analysis; Chapter 18, Loop Optimizations)
-  Andrew W. Appel with Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998. (Chapter 1, Introduction; Chapter 17, Dataflow Analysis; Chapter 18, Loop Optimizations)

Further Reading for Chapter 1 (6)

-  Andrew W. Appel with Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002. (Chapter 1, Introduction; Chapter 17, Data-flow Analysis; Chapter 18, Loop Optimizations)
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 1, Overview of Compilation; Chapter 8, Introduction to Code Optimization; Chapter 10, Scalar Optimizations)
-  Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, William M. Waite. *Eli: A Complete, Flexible Compiler Construction System*. Communications of the ACM 35(2):121-131, 1992.




Further Reading for Chapter 1 (7)

Program Analysis


-  Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977. (Chapter 1, Introduction)
-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 1, An Introduction to Data Flow Analysis; Chapter 10, Implementing Data Flow Analysis in GCC; Appendix A, An Introduction to GCC)
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 1, Introduction)


Further Reading for Chapter 1 (8)

Optimization

-  Donald E. Knuth. *An Empirical Study of Fortran Programs*. Software – Practice and Experience 1:105-133, 1971.
-  Stephen S. Muchnick, Neil D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981. (Chapter 1, A Survey of Data Flow Analysis Techniques)
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 1, Introduction to Advanced Topics; Chapter 8, Data-Flow Analysis; Chapter 11, Introduction to Optimization)

Further Reading for Chapter 1 (9)


 Chris Lattner, Vikram S. Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*. In Proceedings of the 3rd Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004), 75-88, 2004.

 Chris Lattner, Vikram S. Adve. *The LLVM Compiler Framework and Infrastructure Tutorial*. In Proceedings of the 17th International Workshop on Languages and Compilers for High Performance Computing (LCPC 2004), Revised Selected Papers, 15-16, 2004.


Miscellaneous

 Clement A. Baker-Finch, Kevin Glynn, Simon L. Peyton Jones. *Constructed Product Result Analysis for Haskell*. Journal of Functional Programming 14(2):211-245, 2004.


Further Reading for Chapter 1 (10)

 Melvin E. Conway. *Proposal for an UNCOL*. Communications of the ACM 1(3):5, 1958.




On-line Textbooks

 Jack Crenshaw. *Let's build a Compiler*. A set of tutorial articles, on-line published, 1988-1995.
<http://www.iecc.com/compilers/crenshaw>

On-line Resources for Compilers and Compiler Writing Tools

 German National Research Center for Information Technology, Fraunhofer Institute for Computer Architecture and Software Technology. *The Catalog of Compiler Construction Tools*, 1996-2006.
<http://catalog.compilertools.net/>

Further Reading for Chapter 1 (11)

-  Compilers.net Team. *Search Machine on Compilers and Programming Languages, Directory of Compiler and Language Resources*, 1997-2007.
<http://www.compilers.net>
-  Nullstone Corporation. *The Compiler Connection: A Resource for Compiler Developers and Those who use Their Products and Services (Books, Tools, Techniques, Conferences, Jobs and more)*, 2011-2012.
<http://www.compilerconnection.com>
-  Olaf Langmack. *Catalog of Compiler Construction Products 01-98*. 13th Issue, 1998.
<http://compilers.iecc.com/tools.html>

Further Reading for Chapter 1 (12)

-  William M. Waite, Uwe Kastens et al. *Eli: Translator Construction made Easy*, 1989-today.
<http://eli-project.sourceforge.net/>
-  Free Software Foundation (FSF). *GCC, the GNU Compiler Collection*. <https://gcc.gnu.org/>
-  LLVM Foundation. *The LLVM Compiler Infrastructure*.
<http://llvm.org/>
-  The SUIF Group (Monica S. Lam et al.), Stanford University. *The SUIF (Stanford University Intermediate Format) Compiler System*. <http://suif.stanford.edu/>

Further Reading for Chapter 1 (13)



Sable Research Group (Laurie Hendren et al.), McGill University, Secure Software Engineering Group (Eric Bodden et al.), TU Darmstadt/U. Paderborn. *Soot: A Framework for Analyzing and Transforming Java and Android Applications*. <https://sable.github.io/soot/>

Chapter 2

Classical Gen/Kill Data Flow Analyses

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

118/185

Classical Gen/Kill Data Flow Analyses (1)

Gen/Kill Data Flow Analyses are ubiquitous in data flow analysis and there is a huge number of them.

Next, we focus on a canonical collection of four analyses:

- ▶ Reaching Definitions
- ▶ Available Expressions
- ▶ Live Variables
- ▶ Very Busy Expressions

Classical Gen/Kill Data Flow Analyses (2)

...are **classifiable** according to the **direction of the information flow**:

- ▶ **Forward Problems**
 - ▶ Reaching Definitions
 - ▶ Available Expressions
- ▶ **Backward Problems**
 - ▶ Live Variables
 - ▶ Very Busy Expressions

Classical Gen/Kill Data Flow Analyses (3)

...and their dependency on a quantification of program paths:

Forward Problems

- ▶ **Existential/may:** Reaching Definitions
 - ▶ A definition d of a variable v reaches a program point u , if d occurs on some path from the beginning of the program to u and is not followed by any other definition of v on this path.
- ▶ **Universal/must:** Available Expressions
 - ▶ An expression e is available at a program point u if all paths from the beginning of the program to u contain a computation of e which is not followed by an assignment to any of its operands.

Classical Gen/Kill Data Flow Analyses (4)

Backward Problems

- ▶ **Existential/may:** Live Variables
 - ▶ A variable v is **live** at a program point u if **some path from u to the end of the program** contains a use of v which is not preceded by its definition.
- ▶ **Universal/must:** Very Busy Expressions
 - ▶ An expression e is **very busy** at a program point u if **all paths from u to the end of the program** contain a computation of e which is not preceded by an assignment to any of its operands.

In the following sections

...we will consider these [information flow problems](#) (each together with a [typical application](#)) in more detail following the approach of Nielson, Nielson, and Hankin:

- ▶ Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005.

Chapter 2.1

Programs, Flow Graphs

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

124/185

Formalising the Development

- ▶ the programming language of interest
 - ▶ abstract syntax
 - ▶ labelled program fragments
- ▶ abstract flow graphs
 - ▶ control and data flow between labelled program fragments
- ▶ extract equations from the program
 - ▶ specify the information to be computed at entry and exit of labeled fragments
- ▶ compute the solution to the equations
 - ▶ work list algorithms
 - ▶ compute entry and exit information at entry and exit of labelled fragments

WHILE Language

Syntactic categories

$a \in \text{AExp}$ arithmetic expressions

$b \in \text{BExp}$ boolean expressions

$S \in \text{Stmt}$ statements

$x, y \in \text{Var}$ variables

$n \in \text{Num}$ numerals

$l \in \text{Lab}$ labels

$op_a \in \text{Op}_a$ arithmetic operators

$op_b \in \text{Op}_b$ boolean operators

$op_r \in \text{Op}_r$ relational operators

Abstract Syntax

$$\begin{aligned} a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\ b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\ S & ::= [x:=a]^\ell \mid [\text{skip}]^\ell \\ & \quad \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \\ & \quad \mid \text{while } [b]^\ell \text{ do } S \text{ od} \\ & \quad \mid S_1; S_2 \end{aligned}$$

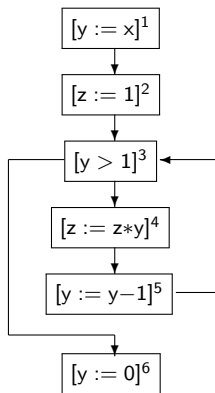
Assignments and **tests** are (uniquely) labelled to allow analyses to refer to these program fragments – the labels correspond to pointers into the syntax tree. We use abstract syntax and insert parentheses to disambiguate syntax.

We will often refer to labelled fragments as **elementary blocks**.

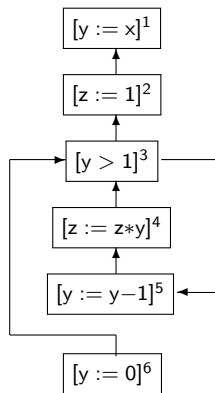
A Program and its Flow Graph

Example:

$[y := x]^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } [z := z * y]^4; [y := y - 1]^5 \text{ od}; [y := 0]^6$



$\text{flow}(S_*) = \{(1, 2), (2, 3), (3, 4),$
 $(4, 5), (5, 3), (3, 6)\}$



$\text{flow}^R(S_*) = \{(6, 3), (3, 5), (5, 4),$
 $(4, 3), (3, 2), (2, 1)\}$

Auxiliary Functions for Flow Graphs

- labels(S)** set of nodes of flow graphs of S .
- init(S)** initial node of flow graph of S ; the unique node where execution of program starts.
- final(S)** final nodes of flow graph for S ; set of nodes where program execution may terminate.
- flow(S)** edges of flow graphs for S (used for forward analyses).
- flow^R(S)** reverse edges of flow graphs for S (used for backward analyses).
- blocks(S)** set of elementary blocks in a flow graph.

Computing the Auxiliary Information (1)

S	$\text{labels}(S)$	$\text{init}(S)$	$\text{final}(S)$
$[x := a]^\ell$	$\{l\}$	l	$\{l\}$
$[\text{skip}]^\ell$	$\{l\}$	l	$\{l\}$
$S_1; S_2$	$\text{labels}(S_1) \cup \text{labels}(S_2)$	$\text{init}(S_1)$	$\text{final}(S_2)$
$\text{if } [b]^\ell \text{ then } (S_1) \text{ else } (S_2)$	$\{l\} \cup \text{labels}(S_1) \cup \text{labels}(S_2)$	l	$\text{final}(S_1) \cup \text{final}(S_2)$
$\text{while } [b]^\ell \text{ do } S \text{ od}$	$\{l\} \cup \text{labels}(S)$	l	$\{l\}$

Computing the Auxiliary Information (2)

S	$\text{flow}(S)$	$\text{blocks}(S)$
$[x := a]^\ell$	\emptyset	$\{[x := a]^\ell\}$
$[\text{skip}]^\ell$	\emptyset	$\{[\text{skip}]^\ell\}$
$S_1; S_2$	$\text{flow}(S_1) \cup \text{flow}(S_2) \cup \{(\ell, \text{init}(S_2)) \mid \ell \in \text{final}(S_1)\}$	$\text{blocks}(S_1) \cup \text{blocks}(S_2)$
$\text{if } [b]^\ell \text{ then } (S_1) \text{ else } (S_2)$	$\text{flow}(S_1) \cup \text{flow}(S_2) \cup \{(\ell, \text{init}(S_1)), (\ell, \text{init}(S_2))\}$	$\{[b]^\ell\} \cup \text{blocks}(S_1) \cup \text{blocks}(S_2)$
$\text{while } [b]^\ell \text{ do } S \text{ od}$	$\{(\ell, \text{init}(S))\} \cup \text{flow}(S) \cup \{(\ell', \ell) \mid \ell' \in \text{final}(S)\}$	$\{[b]^\ell\} \cup \text{blocks}(S)$

$$\text{flow}^R(S) = \{(\ell, \ell') \mid (\ell', \ell) \in \text{flow}(S)\}$$

Further Notations (1)

We shall use the following notation for a program of interest:

- ▶ S_* to represent the program being analyzed (the 'top level' statement).
- ▶ Lab_* to represent the labels ($labels(S_*)$) appearing in S_* .
- ▶ Var_* to represent the variables ($FV(S_*)$) appearing in S_* .
- ▶ $Blocks_*$ to represent the elementary blocks ($blocks(S_*)$) occurring in S_* .
- ▶ $AExp_*$ to represent the set of *non-trivial arithmetic subexpressions* in S_* ; an expression is *trivial* if it is a single variable or constant.
- ▶ $AExp(a)$, $AExp(b)$ to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression.

Further Notations (2)

Free Variables

The set of **free variables** $FV(a)$ of an arithmetic expression, $a \in AExp$, is defined to be the set of variables occurring in it.

Compositional definition of subset $FV(a)$ of Var :

$$FV(n) = \emptyset$$

$$FV(x) = \{x\}$$

$$FV(a_1 + a_2) = FV(a_1) \cup FV(a_2)$$

$$FV(a_1 * a_2) = FV(a_1) \cup FV(a_2)$$

$$FV(a_1 - a_2) = FV(a_1) \cup FV(a_2)$$

Similarly for boolean expressions, $b \in BExp$, and statements, $S \in Stmt$, such that $Var_* = FV(S_*)$.

Illustration

Example:

$[y := x]^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } [z := z * y]^4; [y := y - 1]^5 \text{ od}; [y := 0]^6$

$$\text{labels}(S_*) = \{1, 2, 3, 4, 5, 6\}$$

$$\text{init}(S_*) = 1$$

$$\text{final}(S_*) = \{6\}$$

$$\text{flow}(S_*) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 3), (3, 6)\}$$

$$\text{flow}^R(S_*) = \{(6, 3), (3, 5), (5, 4), (4, 3), (3, 2), (2, 1)\}$$

$$\text{blocks}(S_*) = \{[y := x]^1, [z := 1]^2, [y > 1]^3, \\ [z := z * y]^4, [y := y - 1]^5, [y := 0]^6\}$$

Simplifying Assumptions

The program of interest S_* is often assumed to satisfy:

- ▶ S_* has isolated entries if there are no edges leading into $\text{init}(S_*)$:

$$\forall l : (l, \text{init}(S_*)) \notin \text{flow}(S_*)$$

- ▶ S_* has isolated exits if there are no edges leading out of labels in $\text{final}(S_*)$:

$$\forall l \in \text{final}(S_*), \forall l' : (l, l') \notin \text{flow}(S_*)$$

- ▶ S_* is label consistent if

$$\forall B_1^{l_1}, B_2^{l_2} \in \text{blocks}(S_*) : l_1 = l_2 \rightarrow B_1 = B_2$$

This holds if S_* is uniquely labelled.

Chapter 2.2

Forward Analyses

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chapter 2.2.1

Reaching Definitions

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Reaching Definitions Analysis

Definition 2.2.1.1 (Reaching Definitions)

A definition of variable v at label l reaches the entry from a label l' if there is a path from l to l' that does not re-define v .

Reaching Definitions Analysis

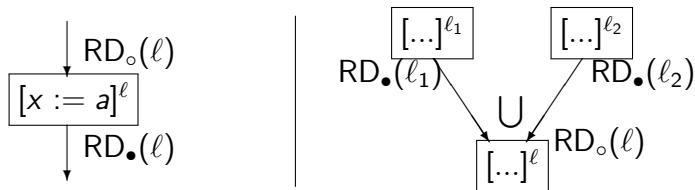
...determines for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.

Example:

$[y := x]^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } [z := z * y]^4; [y := y - 1]^5 \text{ od}; [y := 0]^6$

- ▶ The assignments labelled 1,2,4,5 reach the entry at 4.
- ▶ Only the assignments labelled 1,4,5 reach the entry at 5.

RD Analysis Information and Characteristics



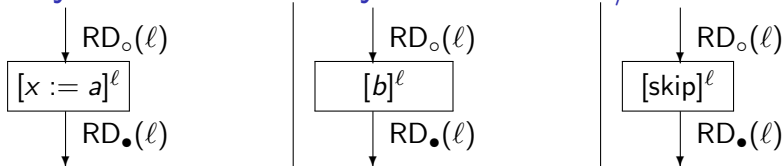
Analysis information: $RD_o(l), RD_\bullet(l) : \text{Lab}_* \rightarrow \mathcal{P}(\text{Var}_* \times \text{Lab}_*)$

- ▶ $RD_o(l)$: the definitions that reach **entry** of block l .
- ▶ $RD_\bullet(l)$: the definitions that reach **exit** of block l .

Analysis characteristics:

- ▶ Direction: forward
- ▶ May analysis with combination operator \cup

Analysis of Elementary Blocks: Gen/Kill-Defs.



$$\text{gen}_{RD}([x := a]^\ell) = \{(x, \ell)\}$$

$$\text{gen}_{RD}([b]^\ell) = \emptyset$$

$$\text{gen}_{RD}([\text{skip}]^\ell) = \emptyset$$

$$\text{kill}_{RD}([x := a]^\ell) = \{(x, ?)\} \cup \{(x, \ell') \mid B^{\ell'} \text{ is assignment to } x\}$$

$$\text{kill}_{RD}([b]^\ell) = \emptyset$$

$$\text{kill}_{RD}([\text{skip}]^\ell) = \emptyset$$

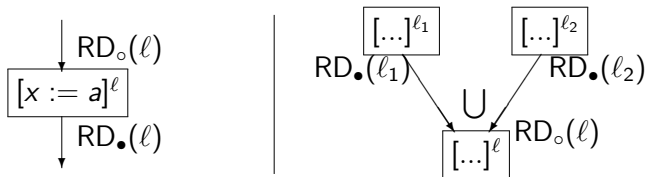
Example:

$[x := y]^1; [x := x + 3]^2;$

► $\text{gen}_{RD}([x := y]^1) = \{(x, 1)\}$

► $\text{kill}_{RD}([x := y]^1) = \{(x, ?)\} \cup \{(x, 1), (x, 2)\}$

Analysis of the Program: The RD Equations



$$\begin{aligned}
 RD_o(l) &= \begin{cases} \{(x, ?) \mid x \in FV(S_*)\} & : \text{ if } l = \text{init}(S_*) \\ \bigcup \{RD_•(l') \mid (l', l) \in \text{flow}(S_*)\} & : \text{ otherwise} \end{cases} \\
 RD_•(l) &= (RD_o(l) \setminus \text{kill}_{RD}(B^l)) \cup \text{gen}_{RD}(B^l) \quad \text{where } B^l \in \text{blocks}(S_*)
 \end{aligned}$$

Illustration

Example:

$[y := x]^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } [z := z * y]^4; [y := y - 1]^5 \text{ od}; [y := 0]^6$

Equations: Let

$$S_1 = \{(y, ?), (y, 1), (y, 5), (y, 6)\}, S_2 = \{(z, ?), (z, 2), (z, 4)\}$$

$$RD_o(1) = \{(x, ?), (y, ?), (z, ?)\} \quad RD_\bullet(1) = RD_o(1) \setminus S_1 \cup \{(y, 1)\}$$

$$RD_o(2) = RD_\bullet(1) \quad RD_\bullet(2) = RD_o(2) \setminus S_2 \cup \{(z, 2)\}$$

$$RD_o(3) = RD_\bullet(2) \cup RD_\bullet(5) \quad RD_\bullet(3) = RD_o(3)$$

$$RD_o(4) = RD_\bullet(3) \quad RD_\bullet(4) = RD_o(4) \setminus S_2 \cup \{(z, 4)\}$$

$$RD_o(5) = RD_\bullet(4) \quad RD_\bullet(5) = RD_o(5) \setminus S_1 \cup \{(y, 5)\}$$

$$RD_o(6) = RD_\bullet(5) \quad RD_\bullet(6) = RD_o(6) \setminus S_1 \cup \{(y, 6)\}$$

ℓ	$RD_o(\ell)$	$RD_\bullet(\ell)$
1	$\{(x, ?), (y, ?), (z, ?)\}$	$\{(x, ?), (y, 1), (z, ?)\}$
2	$\{(x, ?), (y, 1), (z, ?)\}$	$\{(x, ?), (z, 2), (y, 1)\}$
3	$\{(x, ?), (z, 4), (z, 2), (y, 5), (y, 1)\}$	$\{(x, ?), (z, 4), (z, 2), (y, 5), (y, 1)\}$
4	$\{(x, ?), (z, 4), (z, 2), (y, 5), (y, 1)\}$	$\{(z, 4), (x, ?), (y, 5), (y, 1)\}$
5	$\{(z, 4), (x, ?), (y, 5), (y, 1)\}$	$\{(z, 4), (x, ?), (y, 5)\}$
6	$\{(x, ?), (z, 4), (z, 2), (y, 5), (y, 1)\}$	$\{(z, 4), (x, ?), (z, 2), (y, 6)\}$

Solving the RD Equations: The Algorithm (1)

Input

- ▶ A set of reaching definitions equations.

Output

- ▶ The **least solution** to the equations: RD_{\circ} .

Data structures

- ▶ The current analysis result for block entries: RD_{\circ} .
- ▶ The worklist W : a list of pairs (ℓ, ℓ') indicating that the current analysis result has changed at the entry to the block ℓ and hence the information must be recomputed for ℓ' .

Solving the RD Equations: The Algorithm (2)

```
W:=nil;
foreach  $(\ell, \ell') \in \text{flow}(S_*)$  do  $W := \text{cons}((\ell, \ell'), W)$ ; od;
foreach  $\ell \in \text{labels}(S_*)$  do
  if  $\ell \in \text{init}(S_*)$  then
     $\text{RD}_o(\ell) := \{(x, ?) \mid x \in \text{FV}(S_*)\}$ 
  else
     $\text{RD}_o(\ell) := \emptyset$ 
  fi
od
while  $W \neq \text{nil}$  do
   $(\ell, \ell') := \text{head}(W)$ ;
   $W := \text{tail}(W)$ ;
  if  $(\text{RD}_o(\ell) \setminus \text{kill}_{\text{RD}}(B^\ell)) \cup \text{gen}_{\text{RD}}(B^\ell) \not\subseteq \text{RD}_o(\ell')$  then
     $\text{RD}_o(\ell') := \text{RD}_o(\ell') \cup (\text{RD}_o(\ell) \setminus \text{kill}_{\text{RD}}(B^\ell)) \cup \text{gen}_{\text{RD}}(B^\ell)$ ;
    foreach  $\ell''$  with  $(\ell', \ell'')$  in  $\text{flow}(S_*)$  do
       $W := \text{cons}((\ell', \ell''), W)$ ;
    od
  fi
od
```

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

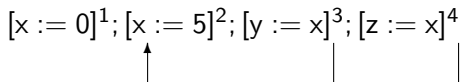
Chap. 10

Application/Usage of RD Information

...for constructing **Use-Definition** and **Definition-Use Chains**:

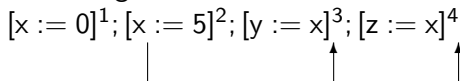
► Use-Definition chains or *ud* chains

each use of a variable is linked to all assignments that *reach* it



► Definition-Use chains or *du* chains

each assignment of a variable is linked to all uses of it



UD/DU Chains: Defined via RDs

$$\text{UD, DU} : \text{Var}_* \times \text{Lab}_* \rightarrow \mathcal{P}(\text{Lab}_*)$$

are defined by

$$\text{UD}(x, \ell) = \begin{cases} \{\ell' \mid (x, \ell') \in \text{RD}_o(\ell)\} & : \text{ if } x \in \text{used}(B^\ell) \\ \emptyset & : \text{ otherwise} \end{cases}$$

where $\text{used}([x := a]^\ell) = \text{FV}(a)$, $\text{used}([b]^\ell) = \text{FV}(b)$,
 $\text{used}([\text{skip}]^\ell) = \emptyset$

and

$$\text{DU}(x, \ell) = \{\ell' \mid \ell \in \text{UD}(x, \ell')\}$$

Chapter 2.2.2

Available Expressions

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Available Expressions Analysis

Definition 2.2.2.1 (Available Expressions)

An expression is **available at the entry from a label** if, no matter what path is taken from the entry of the program to that label, the expression is computed without that any of the variables occurring in it is redefined afterwards.

Available Expression Analysis

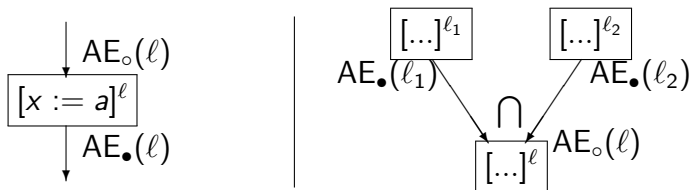
...determines for each program point, which expressions **must** have already been computed, and not later modified, on all paths to the program point.

Example:

$[x := a+b]^1; [y := a*x]^2; \text{while } [y > a+b]^3 \text{ do } [a := a + 1]^4; [x := a + b]^5 \text{ od}$

- ▶ No expression is available at the start of the program.
- ▶ The expression $a+b$ is available every time execution reaches the test in the loop at 3.

AE Analysis Information and Characteristics



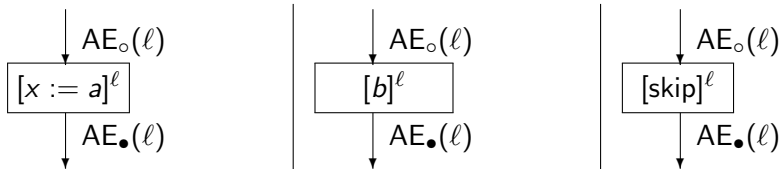
Analysis information: $AE_o(l), AE_\bullet(l) : \text{Lab}_* \rightarrow \mathcal{P}(\text{AExp}_*)$

- ▶ $AE_o(l)$: the expressions that have been comp. at **entry** of block l .
- ▶ $AE_\bullet(l)$: the expressions that have been comp. at **exit** of block l .

Analysis characteristics:

- ▶ Direction: forward
- ▶ Must analysis with combination operator \cap

Analysis of Elementary Blocks: Gen/Kill-Defs.



$$\text{gen}_{AE}([x := a]^\ell) = \{a' \in AExp(a) \mid x \notin FV(a')\}$$

$$\text{gen}_{AE}([b]^\ell) = AExp(b)$$

$$\text{gen}_{AE}([\text{skip}]^\ell) = \emptyset$$

$$\text{kill}_{AE}([x := a]^\ell) = \{a' \in AExp_\star \mid x \in FV(a')\}$$

$$\text{kill}_{AE}([b]^\ell) = \emptyset$$

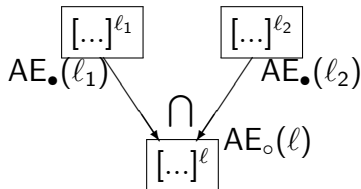
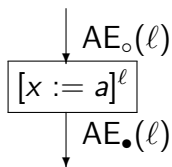
$$\text{kill}_{AE}([\text{skip}]^\ell) = \emptyset$$

Example: $[x := a+b]^1; [y := a*x]^2;$

► $\text{gen}_{AE}([x := a+b]^1) = \{a+b\}$

► $\text{kill}_{AE}([x := a+b]^1) = \{a*x\}$

Analysis of the Program: The AE Equations



$$\begin{aligned}
 AE_o(\ell) &= \begin{cases} \emptyset & : \text{ if } \ell = \text{init}(S_\star) \\ \bigcap \{AE_\bullet(\ell') \mid (\ell', \ell) \in \text{flow}(S_\star)\} & : \text{ otherwise} \end{cases} \\
 AE_\bullet(\ell) &= (AE_o(\ell) \setminus \text{kill}_{AE}(B^\ell)) \cup \text{gen}_{AE}(B^\ell) \quad \text{where } B^\ell \in \text{blocks}(S_\star)
 \end{aligned}$$

Illustration

Example:

$[x := a+b]^1; [y := a*x]^2; \text{while } [y > a+b]^3 \text{ do } [a := a + 1]^4; [x := a + b]^5 \text{ od}$

Equations:

$$\begin{array}{ll} \text{AE}_o(1) = \emptyset & \text{AE}_\bullet(1) = \text{AE}_o(1) \setminus \{a * x\} \cup \{a + b\} \\ \text{AE}_o(2) = \text{AE}_\bullet(1) & \text{AE}_\bullet(2) = \text{AE}_o(2) \setminus \emptyset \cup \{a * x\} \\ \text{AE}_o(3) = \text{AE}_\bullet(2) \cap \text{AE}_\bullet(5) & \text{AE}_\bullet(3) = \text{AE}_o(3) \setminus \emptyset \cup \{a + b\} \\ \text{AE}_o(4) = \text{AE}_\bullet(3) & \text{AE}_\bullet(4) = \text{AE}_o(4) \setminus \{a + b, a * x, a + 1\} \cup \emptyset \\ \text{AE}_o(5) = \text{AE}_\bullet(4) & \text{AE}_\bullet(5) = \text{AE}_o(5) \setminus \{a * x\} \cup \{a + b\} \end{array}$$

ℓ	$\text{AE}_o(\ell)$	$\text{AE}_\bullet(\ell)$
1	\emptyset	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*x\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	\emptyset
5	\emptyset	$\{a+b\}$

Remark: The predefined AE Analysis in PAG/WWW includes boolean expressions.

Solving AE Equations: The Algorithm (1)

Input

- ▶ A set of available expressions equations.

Output

- ▶ The **largest solution** to the equations: AE_o .

Data structures

- ▶ The current analysis result for block entries: AE_o .
- ▶ The worklist W : a list of pairs (l, l') indicating that the current analysis result has changed at the entry to the block l and hence the information must be recomputed for l' .

Solving AE Equations: The Algorithm (2)

```
W:=nil;
foreach  $(\ell, \ell') \in \text{flow}(S_*)$  do  $W := \text{cons}((\ell, \ell'), W)$ ; od;
foreach  $\ell \in \text{labels}(S_*)$  do
  if  $\ell \in \text{init}(S_*)$  then
     $\text{AE}_o(\ell) := \emptyset$ 
  else
     $\text{AE}_o(\ell) := \text{AExp}_*$ 
  fi
od
while  $W \neq \text{nil}$  do
   $(\ell, \ell') := \text{head}(W)$ ;
   $W := \text{tail}(W)$ ;
  if  $(\text{AE}_o(\ell) \setminus \text{kill}_{\text{AE}}(B^\ell)) \cup \text{gen}_{\text{AE}}(B^\ell) \not\subseteq \text{AE}_o(\ell')$  then
     $\text{AE}_o(\ell') := \text{AE}_o(\ell') \cap (\text{AE}_o(\ell) \setminus \text{kill}_{\text{AE}}(B^\ell)) \cup \text{gen}_{\text{AE}}(B^\ell)$ ;
    foreach  $\ell''$  with  $(\ell', \ell'')$  in  $\text{flow}(S_*)$  do
       $W := \text{cons}((\ell', \ell''), W)$ ;
    od
  fi
od
```

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

154/185

1st Application/Usage of AE-Information

Common Subexpression Elimination (CSE)

...aims at finding computations that are always performed at least twice on a given execution path and to eliminate the second and later occurrences; it uses [Available Expressions Analysis](#) to determine the redundant computations.

Example:

$[x := a+b]^1; [y := a*x]^2; \text{while } [y > a+b]^3 \text{ do } [a := a + 1]^4; [x := a + b]^5 \text{ od}$

- ▶ Expression $a+b$ is computed at 1 and 5 and recomputation can be eliminated at 3.

The Optimization: CSE

Let S_\star^N be the normalized form of S_\star such that there is at most one operator on the right hand side of an assignment.

For each $[...a...]^\ell$ in S_\star^N with $a \in AE_o(\ell)$ do

- ▶ determine the set $\{[y_1 := a]^{\ell_1}, \dots, [y_k := a]^{\ell_k}\}$ of elementary blocks in S_\star^N 'defining' a that **reaches** $[...a...]^\ell$
- ▶ create a fresh variable u and
 - ▶ replace each occurrence of $[y_i := a]^{\ell_i}$ with $[u := a]^{\ell_i}; [y_i := u]^{\ell_i}$ for $1 \leq i \leq k$
 - ▶ replace $[...a...]^\ell$ with $[...u...]^\ell$

$[x := a]^{\ell'}$ **reaches** $[...a...]^\ell$ if there is a path in $\text{flow}(S_\star^N)$ from ℓ' to ℓ that does not contain *any* assignments with expression a on the right hand side and no variable of a is modified.

Computing the 'reaches' Information

$[x := a]^{\ell'}$ **reaches** $[...a...]^{\ell}$ if there is a path in $\text{flow}(S_{\star}^N)$ from ℓ' to ℓ that does not contain **any** assignments with expression a on the right hand side and no variable of a is modified.

The set of elementary blocks that **reaches** $[...a...]^{\ell}$ can be computed as $\text{reaches}_{\circ}(a, \ell)$ where

$$\begin{aligned} \text{reaches}_{\circ}(a, \ell) &= \begin{cases} \emptyset & : \text{ if } \ell = \text{init}(S_{\star}) \\ \cup \text{reaches}_{\bullet}(a, \ell') & : \text{ otherwise} \end{cases} \\ \text{reaches}_{\bullet}(a, \ell) &= \begin{cases} \{B^{\ell}\} & : \text{ if } B^{\ell} \text{ has the form } [x := a]^{\ell} \text{ and } x \notin \text{FV}(a) \\ \emptyset & : \text{ if } B^{\ell} \text{ has the form } [x := \dots]^{\ell} \text{ and } x \in \text{FV}(a) \\ \text{reaches}_{\circ}(a, \ell) & : \text{ otherwise} \end{cases} \end{aligned}$$

Illustration: CSE

Example:

$[x := a+b]^1; [y := a*x]^2; \text{while } [y > a+b]^3 \text{ do } [a := a + 1]^4; [x := a + b]^5 \text{ od}$

ℓ	$AE_o(\ell)$
1	\emptyset
2	$\{a+b\}$
3	$\{a+b\}$
4	$\{a+b\}$
5	\emptyset

$$\text{reaches}(a+b,3) = \{[x := a + b]^1, [x := a + b]^5\}$$

Result of CSE optimization wrt $\text{reaches}(a+b,3)$:

$[u := a+b]^1; [x := u]^1; [y := a*x]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^5; [x := u]^5 \text{ od}$

2nd Application/Usage of AE-Information (1)

Copy Analysis

...aims at determining for each program point ℓ' , which copy statements $[x := y]^\ell$ that still are relevant (i.e., neither x nor y have been redefined) when control reaches point ℓ' .

Example:

$[a := b]^1$; if $[x > b]^2$ then $([y := a]^3)$ else $([b := b + 1]^4$; $[y := a]^5)$; $[\text{skip}]^6$

ℓ	$C_o(\ell)$	$C_\bullet(\ell)$
1	\emptyset	$\{(a,b)\}$
2	$\{(a,b)\}$	$\{(a,b)\}$
3	$\{(a,b)\}$	$\{(y,a), (a,b)\}$
4	$\{(a,b)\}$	\emptyset
5	\emptyset	$\{(y,a)\}$
6	$\{(y,a)\}$	$\{(y,a)\}$

2nd Application/Usage of AE-Information (2)

Copy Propagation

...aims at finding copy statements $[x := y]^{\ell_j}$ and eliminating them if possible.

If x is used in $B^{\ell'}$ then x can be replaced by y in $B^{\ell'}$ provided that

- ▶ $[x := y]^{\ell_j}$ is the only kind of definition of x that reaches $B^{\ell'}$: this information can be obtained from the **def-use chain**.
- ▶ on every path from ℓ_j to ℓ' (including paths going through ℓ' several times but only once through ℓ_j) there are no redefinitions of y : this can be detected by **Copy Analysis**.

The Optimization: Copy Propagation

For each copy statement $[x := y]^{\ell_j}$ in S_* do

- ▶ determine the set $\{[\dots x \dots]^{\ell_1}, \dots, [\dots x \dots]^{\ell_i}\}, 1 \leq i \leq k$, of elementary blocks in S_* that uses $[x := y]^{\ell_j}$ – this can be computed from $DU(x, \ell_j)$
- ▶ for each $[\dots x \dots]^{\ell_i}$ in this set determine whether $\{(x', y') \in C_o(\ell_i) \mid x' = x\} = \{(x, y)\}$; if so then $[x := y]$ is the only kind of definition of x that reaches ℓ_i from all ℓ_j .
- ▶ if this holds for all i ($1 \leq i \leq k$) then
 - ▶ remove $[x := y]^{\ell_j}$
 - ▶ replace $[\dots x \dots]^{\ell_i}$ with $[\dots y \dots]^{\ell_i}$ for $1 \leq i \leq k$.

Illustration: Copy Propagation (1)

Example 1

$[u := a+b]^{1'}; [x := u]^1; [y := a*x]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^{5'}; [x := u]^5 \text{ od}$

becomes after Copy Propagation

$[u := a+b]^{1'}; [y := a*u]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^{5'}; [x := u]^5 \text{ od}$

Illustration: Copy Propagation (2)

Example 2

$[a := 2]^1; \text{if } [y > u]^2 \text{ then } ([a := a + 1]^3; [x := a]^4); \text{ else } ([a := a * 2]^5; [x := a]^6); [y := y * x]^7;$

becomes after Copy Propagation

$[a := 2]^1; \text{if } [y > u]^2 \text{ then } ([a := a + 1]^3; \quad ;) \text{ else } ([a := a * 2]^5; \quad); [y := y * a]^7;$

Example 3

$[a := 10]^1; [b := a]^2; \text{while } [a > 1]^3 \text{ do } [a := a - 1]^4; [b := a]^5; \text{od } [y := y * b]^6;$

becomes after Copy Propagation

$[a := 10]^1; \quad ; \text{while } [a > 1]^3 \text{ do } [a := a - 1]^4; \quad ; \text{od } [y := y * a]^6;$

Chapter 2.2.3

Summary: Forward Analyses

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

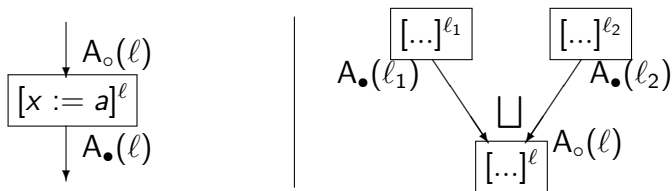
Chap. 7

Chap. 8

Chap. 9

Chap. 10

Summary: Forward Analyses, RD and AE (1)



$$\begin{aligned}
 A_o(l) &= \begin{cases} \iota_A & : \text{if } l = \text{init}(S_*) \\ \bigsqcup_A \{A_•(l') \mid (l', l) \in \text{flow}(S_*)\} & : \text{otherwise} \end{cases} \\
 A_•(l) &= (A_o(l) \setminus \text{kill}_A(B^l)) \cup \text{gen}_A(B^l) \quad \text{where } B^l \in \text{blocks}(S_*)
 \end{aligned}$$

where

Analysis	RD	AE
ι_A	$\{(x, ?) \mid x \in FV(S_*)\}$	\emptyset
\bigsqcup_A	\cup	\cap

Summary: Forward Analyses, RD and AE (2)

This means **effect functions** of blocks are of the form

$$f_\ell = (A_o(\ell) \setminus \text{kill}_A(B^\ell)) \cup \text{gen}_A(B^\ell) \quad \text{where } B^\ell \in \text{blocks}(S_\star)$$

where kill_ℓ and gen_ℓ are auxiliary functions for **invalidating** and **generating** information for an elementary block:

- ▶ $\text{kill}_A(B^\ell)$: information that is **invalidated** by an elementary block.
- ▶ $\text{gen}_A(B^\ell)$: information that is **generated** by an elementary block.

Chapter 2.2.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6




Chap. 7

Chap. 8

Chap. 9

Chap. 10

Further Reading for Chapter 2.2

-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 2, Classical Bit Vector Data Flow Analysis)
-  Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 4.12, Global Available Temporary Information)
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 2, Data Flow Analysis)

Chapter 2.3

Backward Analyses

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chapter 2.3.1

Live Variables

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Live Variables Analysis

Definition 2.3.1.1 (Live Variables)

A variable is **live at the exit from a label** if there is a path from the label to a use of the variable that does not re-define the variable.

Live Variables Analysis

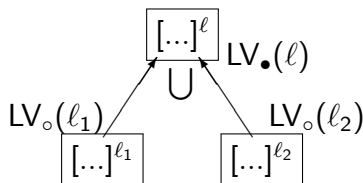
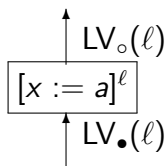
...determines for each program point, which variables **may** be live at the exit from the point.

Example

$[y := 0]^0; [u := a+b]^1; [y := a*u]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^5; [x := u]^6 \text{ od}$

- ▶ y is dead (i.e., not live) at the exit from label 0
- ▶ x is dead (i.e., not live) at the exit from label 6

LV Analysis Information and Characteristics



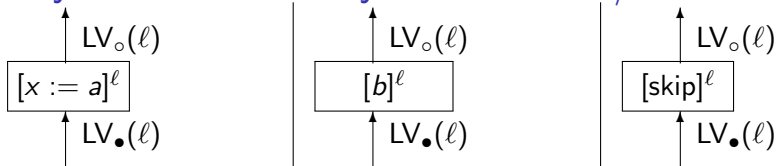
Analysis information: $LV_o(\ell), LV_\bullet(\ell) : \text{Lab}_* \rightarrow \mathcal{P}(\text{Var}_*)$

- ▶ $LV_o(\ell)$: the variables that are live at **entry** of block ℓ .
- ▶ $LV_\bullet(\ell)$: the variables that are live at **exit** of block ℓ .

Analysis characteristics:

- ▶ Direction: backward
- ▶ May analysis with combination operator \cup

Analysis of Elementary Blocks: Gen/Kill-Defs.



$$\text{gen}_{LV}([x := a]^{\ell}) = FV(a)$$

$$\text{gen}_{LV}([b]^{\ell}) = FV(b)$$

$$\text{gen}_{LV}([\text{skip}]^{\ell}) = \emptyset$$

$$\text{kill}_{LV}([x := a]^{\ell}) = \{x\}$$

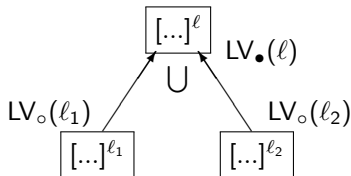
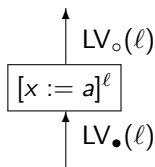
$$\text{kill}_{LV}([b]^{\ell}) = \emptyset$$

$$\text{kill}_{LV}([\text{skip}]^{\ell}) = \emptyset$$

Example: $[u := a+b]^1$;

- ▶ $\text{gen}_{LV}([u := a+b]^1) = \{a, b\}$
- ▶ $\text{kill}_{LV}([u := a+b]^1) = \{u\}$

Analysis of the Program: The LV Equations



$$\begin{aligned} LV_\circ(\ell) &= (LV_\bullet(\ell) \setminus \text{kill}_{LV}(B^\ell)) \cup \text{gen}_{LV}(B^\ell) && \text{where } B^\ell \in \text{blocks}(S_\star) \\ LV_\bullet(\ell) &= \begin{cases} \emptyset & : \text{ if } \ell = \text{final}(S_\star) \\ \bigcup \{LV_\circ(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_\star)\} & : \text{ otherwise} \end{cases} \end{aligned}$$

Illustration

Example

Program	$LV_{\bullet}(\ell)$	$LV_{\circ}(\ell)$	ℓ	$kill_{LV}(\ell)$	$gen_{LV}(\ell)$
$[y := 0]^0;$	$\{a, b\}$	$\{a, b\}$	0	$\{y\}$	\emptyset
$[u := a+b]^1;$	$\{u, a, b\}$	$\{a, b\}$	1	$\{u\}$	$\{a, b\}$
$[y := a*u]^2;$	$\{u, a, b, y\}$	$\{u, a, b\}$	2	$\{y\}$	$\{a, u\}$
$while[y > u]^3 do$	$\{a, b, y\}$	$\{u, a, b, y\}$	3	\emptyset	$\{y, u\}$
$[a := a + 1]^4;$	$\{a, b, y\}$	$\{a, b, y\}$	4	$\{a\}$	$\{a\}$
$[u := a + b]^5;$	$\{u, a, b, y\}$	$\{a, b, y\}$	5	$\{u\}$	$\{a, b\}$
$[x := u]^6 od$	$\{u, a, b, y\}$	$\{u, a, b, y\}$	6	$\{x\}$	$\{u\}$
$[skip]^7$	\emptyset	\emptyset	7	\emptyset	\emptyset

Application/Usage of LV Information

Dead Code Elimination (DCE):

An assignment $[x := a]^\ell$ is **dead** if the value of x is not used before it is redefined. Dead assignments can be eliminated.

- ▶ **Analysis:** Live Variables Analysis
- ▶ **Transformation:** For each $[x := a]^\ell$ in S_\star with $x \notin \text{LV}_\bullet(\ell)$ (i.e., dead) eliminate $[x := a]^\ell$ from the program.

Example:

Before DCE:

$[y := 0]^0; [u := a+b]^1; [y := a*u]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^5; [x := u]^6 \text{ od}$

After DCE:

$[u := a+b]^1; [y := a*u]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^5; \text{od}$

Combining Optimizations

...usually strengthens the overall impact.

Example:

$[x := a+b]^1; [y := a*x]^2; \text{while } [y > a+b]^3 \text{ do } [a := a + 1]^4; [x := a + b]^5 \text{ od}$

1. Common Subexpression Elimination gives

$[u := a+b]^{1'}; [x := u]^1; [y := a*x]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^{5'}; [x := u]^5 \text{ od}$

2. Copy Propagation gives

$[u := a+b]^{1'}; [y := a*u]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^{5'}; [x := u]^5 \text{ od}$

3. Dead Code Elimination gives

$[u := a+b]^1; [y := a*u]^2; \text{while } [y > u]^3 \text{ do } [a := a + 1]^4; [u := a + b]^5; \text{od}$

What are the results for other optimization sequences?

Faint Variables

...generalize the notion of **dead variables**.

Consider the following program consisting of three statements:

$$[x := 1]^1; [x := 2]^2; [y := x]^3;$$

Clearly x is dead at the exit from 1 and y is dead at the exit of 3. But x is live at the exit of 2 although it is only used to calculate a new value for y that turns out to be dead.

We shall say that a variable is a **faint variable** if it is dead or if it is only used to calculate new values for faint variables; otherwise it is **strongly live**.

Chapter 2.3.2

Very Busy Expressions

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Very Busy Expressions Analysis

Definition 2.3.2.1 (Very Busy Expressions)

An expression is **very busy at the exit from a label** if, no matter what path is taken from the label, the expression is always used before any of the variables occurring in it is redefined.

Very Busy Expression Analysis

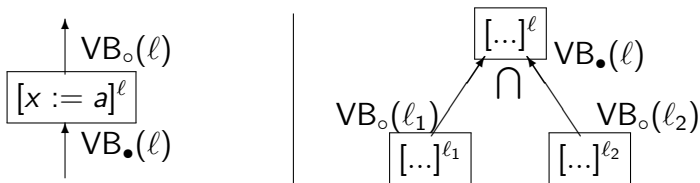
...determines for each program point, which expressions **must** be very busy at the exit from the point.

Example

if $[a > b]^1$ then $([x := b-a]^2; [y := a-b]^3)$ else $([y := b-a]^4; [x := a-b]^5)$

- ▶ $b-a$ and $a-b$ are very busy at the exit from label 1.

VB Analysis Information and Characteristics



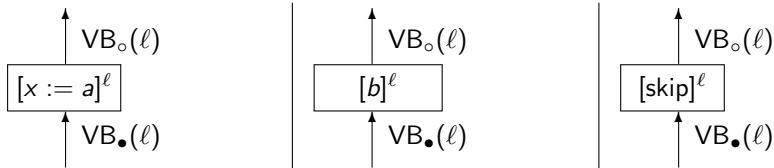
Analysis information: $VB_o(\ell), VB_\bullet(\ell) : \text{Lab}_* \rightarrow \mathcal{P}(\text{AExp}_*)$

- ▶ $VB_o(\ell)$: the expressions that are very busy at **entry** of block ℓ .
- ▶ $VB_\bullet(\ell)$: the expressions that are very busy at **exit** of block ℓ .

Analysis characteristics:

- ▶ Direction: backward
- ▶ Must analysis with combination operator \cap

Analysis of Elementary Blocks: Gen/Kill-Defs.



$$\text{gen}_{\text{VB}}([x := a]^\ell) = \text{AExp}(a)$$

$$\text{gen}_{\text{VB}}([b]^\ell) = \text{AExp}(b)$$

$$\text{gen}_{\text{VB}}([\text{skip}]^\ell) = \emptyset$$

$$\text{kill}_{\text{VB}}([x := a]^\ell) = \{a' \in \text{AExp}_* \mid x \in \text{FV}(a')\}$$

$$\text{kill}_{\text{VB}}([b]^\ell) = \emptyset$$

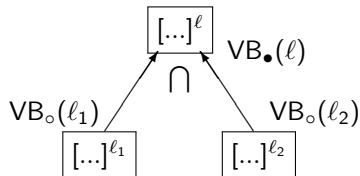
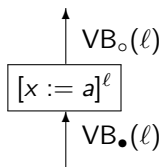
$$\text{kill}_{\text{VB}}([\text{skip}]^\ell) = \emptyset$$

Example: $[x := a+b]^1; [y := a*x]^2; [z := x*b]^3;$

► $\text{gen}_{\text{VB}}([x := a+b]^1) = \{a+b\}$

► $\text{kill}_{\text{VB}}([x := a+b]^1) = \{a*x, x*b\}$

Analysis of the Program: The VB Equations



$$\begin{aligned}
 VB_o(\ell) &= (VB_{\bullet}(\ell) \setminus \text{kill}_{VB}(B^\ell)) \cup \text{gen}_{VB}(B^\ell) && \text{where } B^\ell \in \text{blocks}(S_*) \\
 VB_{\bullet}(\ell) &= \begin{cases} \emptyset & : \text{ if } \ell = \text{final}(S_*) \\ \bigcap \{VB_o(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_*)\} & : \text{ otherwise} \end{cases}
 \end{aligned}$$

Illustration

Example

if $[a > b]^1$ then $([x := b-a]^2; [y := a-b]^3)$ else $([y := b-a]^4; [x := a-b]^5)$

ℓ	$VB_{\bullet}(\ell)$	$VB_{\circ}(\ell)$	ℓ	$kill_{VB}(\ell)$	$gen_{VB}(\ell)$
1	$\{a-b, b-a\}$	$\{a-b, b-a\}$	1	\emptyset	\emptyset
2	$\{a-b\}$	$\{a-b, b-a\}$	2	\emptyset	$\{b-a\}$
3	\emptyset	$\{a-b\}$	3	\emptyset	$\{a-b\}$
4	$\{a-b\}$	$\{a-b, b-a\}$	4	\emptyset	$\{b-a\}$
5	\emptyset	$\{a-b\}$	5	\emptyset	$\{a-b\}$

Application/Usage of VB Information

Code Hoisting

...finds expressions that are always evaluated following some point in the program regardless of the execution path – and moves them to the earliest point (in execution order) beyond which they would always be executed.

Example

Before Code Hoisting:

if $[a > b]^1$ then $([x := b-a]^2; [y := a-b]^3)$ else $([y := b-a]^4; [x := a-b]^5)$

After Code Hoisting:

$[t1 := a-b]^0; [t2 := b-a]^0;$
if $[a > b]^1$ then $([x := t2]^2; [y := t1]^3)$ else $([y := t2]^4; [x := t1]^5)$

Chapter 2.3.3

Summary: Backward Analyses

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

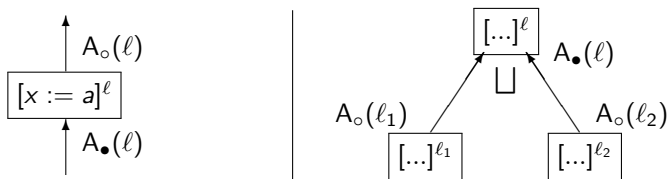
Chap. 7

Chap. 8

Chap. 9

Chap. 10

Summary: Backward Analyses, LV and VB (1)



$$\begin{aligned}
 A_o(l) &= (A_\bullet(l) \setminus \text{kill}_A(B^\ell)) \cup \text{gen}_A(B^\ell) && \text{where } B^\ell \in \text{blocks}(S_\star) \\
 A_\bullet(l) &= \begin{cases} \iota_A & : \text{ if } l = \text{final}(S_\star) \\ \sqcup \{A_o(l') \mid (l', l) \in \text{flow}^R(S_\star)\} & : \text{ otherwise} \end{cases}
 \end{aligned}$$

where

Analysis	LV	VB
ι_A	\emptyset	\emptyset
\sqcup_A	\cup	\cap

Summary: Backward Analyses, LV and VB (2)

This means **effect functions** of blocks are of the form

$$f_\ell = (A_\bullet(\ell) \setminus \text{kill}_A(B^\ell)) \cup \text{gen}_A(B^\ell) \quad \text{where } B^\ell \in \text{blocks}(S_\star)$$

where kill_ℓ and gen_ℓ are auxiliary functions for **invalidating** and **generating** information for an elementary block:

- ▶ $\text{kill}_A(B^\ell)$: information that is **invalidated** by an elementary block.
- ▶ $\text{gen}_A(B^\ell)$: information that is **generated** by an elementary block.

Chapter 2.3.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6




Chap. 7

Chap. 8

Chap. 9

Chap. 10

Further Reading for Chapter 2.3

-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 2, Classical Bit Vector Data Flow Analysis)
-  Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 4.10, Global Anticipated Information)
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 2, Data Flow Analysis)

Chapter 2.4

Taxonomy of Gen/Kill Analyses

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Taxonomy of Gen/Kill Analyses

Analysis	may (existential)	must (universal)
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions
Analysis	may (existential)	must (universal)
Combination Op.	\cup	\cap
Solution of equ.	smallest	largest
Analysis	Extremal labels set	Abstract flow graph
Forward	$\{\text{init}(S_*)\}$	$\text{flow}(S_*)$
Backward	$\text{final}(S_*)$	$\text{flow}^R(S_*)$

Chapter 2.5

Summary, Looking Ahead

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Gen/Kill Data Flow Analyses

...are also known as

- ▶ Bitvector Data Flow Analyses.

This notion refers to a common **implementation strategy** for

- ▶ Gen/Kill Data Flow Analyses.

Bit Vectors and Bit Vector Analyses

The **classical Gen/Kill analyses** operate over elements of $\mathcal{P}(D)$ where D is a finite set.

The elements can be represented as **bit vectors**. Each element of D can be assigned a unique bit position i ($1 \leq i \leq n$). A subset S of D is then represented by a **vector of n bits**:

- ▶ if the i 'th element of D is in S then the i 'th bit is 1.
- ▶ if the i 'th element of D is not in S then the i 'th bit is 0.

Then we have **efficient implementations** of

- ▶ **set union** as logical 'or'.
- ▶ **set intersection** as logical 'and'.

More Bit Vector Framework Examples

- ▶ **Dual available expressions** determines for each program point which expressions may not be available when execution reaches that point (**forward may analysis**)
- ▶ **Copy analysis** determines whether there on every execution path from a copy statement $x := y$ to a use of x there are no assignments to y (**forward must analysis**).
- ▶ **Dominators** determines for each program point which program points are guaranteed to have been executed before the current one is reached (**forward must analysis**).
- ▶ **Upwards exposed uses** determines for a program point, what uses of a variable are reached by a particular definition (assignment) (**backward may analysis**).

Some Non-Bit Vector Framework Examples (1)

- ▶ **Constant propagation** determines for each program point whether or not a variable has a constant value whenever execution reaches that point (**forward must analysis**, cf. Chapter 5).
- ▶ **Detection of signs analysis** determines for each program point the possible signs that the values of the variables may have whenever execution reaches that point (**forward must analysis**).
- ▶ **Faint variables** determines for each program point which variables are faint: a variable is faint if it is dead or it is only used to compute new values of faint variables (**backward must analysis**, cf. Chapter B.4).

Some Non-Bit Vector Framework Examples (2)

- ▶ **May be uninitialized** determines for each program point which variables have dubious values: a variable has a dubious value if either it is not initialized or its value depends on variables with dubious values (**forward may analysis**).

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

198/185

Flow-Sensitive/Flow-Insensitive DFA Problems

...another categorization of DFA problems and analyses:

- ▶ **Flow-sensitive** problems and analyses
 - ▶ The validity of a property at some program point depends on the control flow path(s) involving it.
E.g., Gen/Kill Problems (RD, AE, LV, VB, etc.), constant propagation and folding, partial redundancy elimination, etc.
- ▶ **Flow-insensitive** problems and analyses
 - ▶ The validity of a property at some program point is independent of the control flow path(s) involving it.
E.g., type analysis (for many programming languages, e.g., C but not Ruby), Procedure_X_Can_Modify_Variable_V, Procedure_X_Can_Have_Side_Effects, etc.

Note: Flow insensitivity is often used for trading precision for efficiency and scalability.

Outlook

Gen/Kill Data Flow Analyses

- ▶ are most important in practice,
- ▶ will be reconsidered in detail and from various angles (**soundness**, **completeness**, **optimality**, **implementation**, etc.) in [Chapter 4](#),
- ▶ will be considered in the context of practically relevant optimizations in [Chapter 7](#) and [Chapter 8](#).

Chapter 2.6

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9




Chap. 10

Part III




Chap. 11

201/185

Further Reading for Chapter 2 (1)

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 9.2, Introduction to Data-Flow Analysis; Chapter 9.3, Foundations of Data-Flow Analysis)
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 10.2, A Taxonomy for Transformations — Machine-Independent Transformations, Machine-Dependent Transformations)
-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 2, Classical Bit Vector Data Flow Analysis)

Further Reading for Chapter 2 (2)

-  Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 4, Flow Graph)
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 8.3, Taxonomy of Data-Flow Problems and Solution Methods)
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)

Part II

Intraprocedural Data Flow Analysis

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

204/185

Chapter 3

The Intraprocedural DFA Framework

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

205/185

Chapter 3.1

Preliminaries

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

206/185

Outlook

...before going into details, we first (re-) consider:

- ▶ **Flow graphs** and notions on flow graphs.
- ▶ **Lattices** and properties of functions on lattices.
- ▶ **DFA specifications** and problems.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

207/185

Flow Graphs

Definition 3.1.1 (Flow Graph)

A (non-deterministic) **flow graph** is a quadruple $G = (N, E, s, e)$ with

- ▶ node set N .
- ▶ edge set $E \subseteq N \times N$.
- ▶ distinguished **start node** s w/out any predecessors.
- ▶ distinguished **end node** e w/out any successors.

Nodes represent the **program points**, **edges** the **branching structure** of G . Every node of G is assumed to lie on a path from s to e .

Node-labelled vs. Edge-labelled Flow Graphs

Program instructions (i.e., assignments, tests) can be represented by

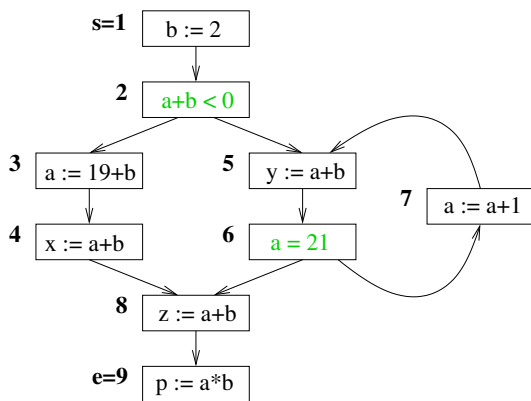
- ▶ nodes
- ▶ edges

Depending on the choice this leads to

- ▶ node-labelled flow graphs
- ▶ edge-labelled flow graphs

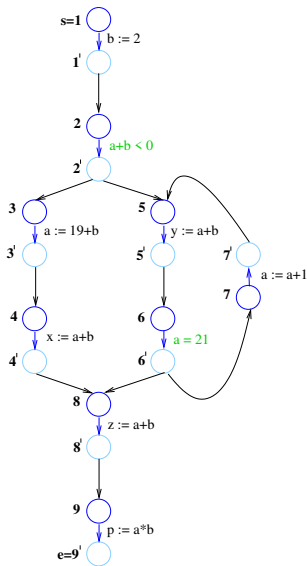
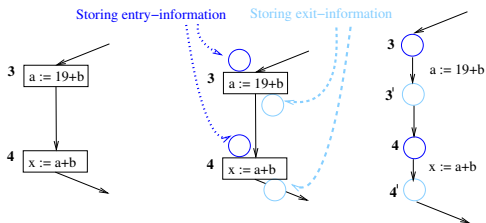
respectively.

Example: A Node-Labelled Flow Graph



Example: An Edge-Labelled Flow Graph

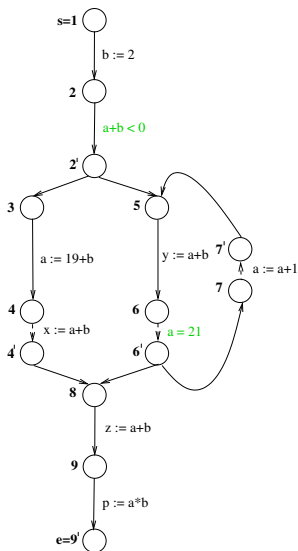
...schematically constructed from its node-labelled counterpart.



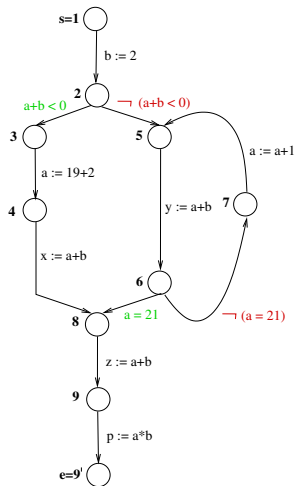
Edge-Labelled Flow Graphs after Cleaning Up

...differing in the handling of tests.

a)



b)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

1212/185

Reverse Flow Graph

Definition 3.1.2 (Reverse Flow Graph)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph. The **reverse flow graph** G_{rev} of G is a quadruple with $G_{rev} = (N', E', \mathbf{s}', \mathbf{e}')$ with

- ▶ node set $N' =_{df} N$.
- ▶ edge set $E' =_{df} \{ (n, m) \mid (m, n) \in E \}$.
- ▶ distinguished **start node** $\mathbf{s}' =_{df} \mathbf{e}$.
- ▶ distinguished **end node** $\mathbf{e}' =_{df} \mathbf{s}$.

Note

- ▶ Like \mathbf{s} and \mathbf{e} , \mathbf{s}' and \mathbf{e}' do not have any predecessors and successors, respectively.
- ▶ Every node in G_{rev} lies on a path from \mathbf{s}' to \mathbf{e}' .

In the following

...we consider for pragmatic reasons

- ▶ edge-labelled flow graphs.

Pragmatics, i.e., advantages and disadvantages of choosing a specific flow graph variant, are discussed in

- ▶ [Appendix B: Pragmatics of Flow Graph Representations](#)

Notions, Notations for Flow Graphs (1)

Let $G = (N, E, s, e)$ be a flow graph, let m, n be two nodes of N .

Predecessor and Successor Nodes

- ▶ $pred_G(n) =_{df} \{ m \mid (m, n) \in E \}$ denotes the set of **predecessor nodes** of n .
- ▶ $succ_G(n) =_{df} \{ m \mid (n, m) \in E \}$ denotes the set of **successor nodes** of n .

Notions, Notations for Flow Graphs (2)

Paths

- ▶ $\mathbf{P}_G[m, n]$ denotes the set of all paths from m to n (including m and n).
- ▶ $\mathbf{P}_G[m, n[$ denotes the set of all paths from m to a predecessor of n .
- ▶ $\mathbf{P}_G]m, n]$ denotes the set of all paths from a successor of m to n .
- ▶ $\mathbf{P}_G]m, n[$ denotes the set of all paths from a successor of m to a predecessor of n .

Note: If G is obvious from the context, we drop G as index and write *pred*, *succ*, and \mathbf{P} instead of pred_G , succ_G , and \mathbf{P}_G , respectively.

Partially Ordered Sets, Complete Lattices

Definition 3.1.3 (Partially Ordered Set)

Let S be a set and $R \subseteq S \times S$ be a relation on S . Then (S, R) is called a **partially ordered set** iff R is reflexive, transitive, and anti-symmetric.

Definition 3.1.4 (Lattice, Complete Lattice)

Let (P, \sqsubseteq) be a partially ordered set, $P \neq \emptyset$.

Then (P, \sqsubseteq) is a

- ▶ **lattice**, if every finite nonempty subset P' of P has a least upper bound and a greatest lower bound in P .
- ▶ **complete lattice**, if every subset P' of P has a least upper bound and a greatest lower bound in P .

Examples: Partially Ordered Sets and Lattices

a)

\vdots
|
3
|
2
|
1
|
0
|
-1
|
-2
|
-3
|
 \vdots

b)

\top
 \vdots
 \vdots
|
3
|
2
|
1
|
0
|
-1
|
-2
|
-3
|
 \vdots
 \perp

c)

\top
 \vdots
 \vdots
|
3
|
2
|
1
|
0

d)

\vdots
|
3
|
2
|
1
|
0

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

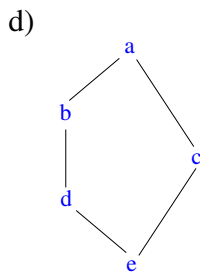
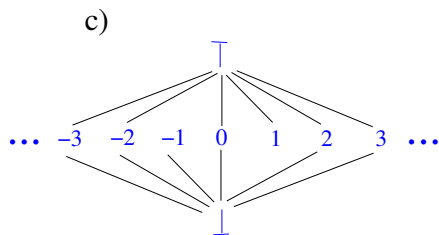
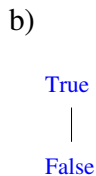
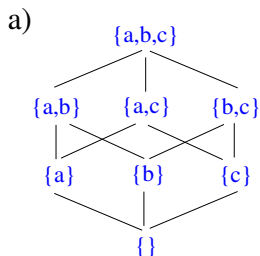
Chap. 8

Chap. 9

Chap. 10

218/185

Examples: Complete Lattices



Notions, Notations for Lattices

Let (C, \sqsubseteq) be a complete lattice, and let $C' \subseteq C$ be a subset of C . Then

- ▶ $\sqcap C'$ denotes the greatest lower bound of C' .
- ▶ $\sqcup C'$ denotes the least upper bound of C' .
- ▶ $\perp =_{df} \sqcap C = \sqcup \emptyset$ denotes the least element of C .
- ▶ $\top =_{df} \sqcup C = \sqcap \emptyset$ denotes the greatest element of C .

This gives rise to write a complete lattice as a quintuple

$$\text{▶ } \hat{C} = (C, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$$

where \sqcap , \sqcup , \perp , and \top are read as meet, join, bottom, and top, respectively.

Descending, Ascending Chain Condition

Definition 3.1.5 (Chain Condition)

A lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ satisfies the

1. **descending chain condition**, if every descending chain gets stationary, i.e., for every chain $c_1 \supseteq c_2 \supseteq \dots \supseteq c_n \supseteq \dots$ there is an index $m \geq 1$ with $c_m = c_{m+j}$ for all $j \in \mathbb{N}$.
2. **ascending chain condition**, if every ascending chain gets stationary, i.e., for every chain $c_1 \sqsubseteq c_2 \sqsubseteq \dots \sqsubseteq c_n \sqsubseteq \dots$ there is an index $m \geq 1$ with $c_m = c_{m+j}$ for all $j \in \mathbb{N}$.

Monotonicity, Distributivity, and Additivity

...are important properties of functions on lattices:

Definition 3.1.6 (Monotonicity)

Let $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ be a complete lattice and $f : \mathcal{C} \rightarrow \mathcal{C}$ be a function on \mathcal{C} . Then f is

- ▶ **monotonic** iff $\forall c, c' \in \mathcal{C}. c \sqsubseteq c' \Rightarrow f(c) \sqsubseteq f(c')$
(Preservation of the order of elements)

Definition 3.1.7 (Distributivity, Additivity)

Let $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ be a complete lattice and $f : \mathcal{C} \rightarrow \mathcal{C}$ be a function on \mathcal{C} . Then f is

- ▶ **distributive** iff $\forall \emptyset \neq C' \subseteq \mathcal{C}. f(\sqcap C') = \sqcap \{f(c) \mid c \in C'\}$
(Preservation of greatest lower bounds)
- ▶ **additive** iff $\forall \emptyset \neq C' \subseteq \mathcal{C}. f(\sqcup C') = \sqcup \{f(c) \mid c \in C'\}$
(Preservation of least upper bounds)

Characterizing Monotonicity

...in terms of the preservation of greatest lower and least upper bounds:

Lemma 3.1.8

Let $f : \mathcal{C} \rightarrow \mathcal{C}$ be a function on a complete lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$. Then:

$$\begin{aligned} f \text{ is monotonic} &\iff \forall \emptyset \neq C' \subseteq \mathcal{C}. f(\bigsqcap C') \sqsubseteq \bigsqcap \{f(c) \mid c \in C'\} \\ &\iff \forall \emptyset \neq C' \subseteq \mathcal{C}. f(\bigsqcup C') \sqsupseteq \bigsqcup \{f(c) \mid c \in C'\} \end{aligned}$$

Useful Results

Let $f : \mathcal{C} \rightarrow \mathcal{C}$ be a function on a complete lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$.

Lemma 3.1.9 (Equivalence)

f is distributive iff f is additive.

Lemma 3.1.10

f is monotonic if f is distributive (additive).

Chapter 3.2

DFA Specification, DFA Problem

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

225/185

DFA Specification

Let $G = (N, E, s, e)$ be an edge-labelled flow graph.

Definition 3.2.1 (DFA Specification)

A DFA specification for G is a quadruple $\mathcal{S}_G = (\widehat{\mathcal{C}}, \llbracket \cdot \rrbracket, c_s, d)$ with

- ▶ $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ a complete lattice.
- ▶ $\llbracket \cdot \rrbracket : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ a local abstract semantics.
- ▶ $c_s \in \mathcal{C}$ an initial information/assertion.
- ▶ $d \in \{fw, bw\}$ a direction of information flow.

Note:

- ▶ *fw* and *bw* stand for *forward* and *backward*, respectively.
- ▶ The validity of $c_s \in \mathcal{C}$ at s needs to be ensured by the calling context of G .

Notations for DFA Specifications

Let $\mathcal{S}_G = (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket, c_s, d)$ be a DFA specification for G .

Then

- ▶ The elements of \mathcal{C} represent the **data flow information** of interest.
- ▶ The functions $\llbracket e \rrbracket, e \in E$, abstract the concrete semantics of instructions to the level of the analysis.

Thus

- ▶ $\hat{\mathcal{C}}$ is called a **DFA lattice**.
- ▶ $\llbracket \cdot \rrbracket$ is called a **DFA functional**.
- ▶ $\llbracket e \rrbracket, e \in E$, is called a **(local) DFA function**.

DFA Problem

Definition 3.2.2 (DFA Problem)

A DFA specification $\mathcal{S}_G = (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket, c_s, d)$ defines a **DFA problem** for G .

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

228/185

Example: Availability of a Term t (1)

DFA Specification for the Availability of a Term t

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\text{IB}, \wedge, \vee, \leq, \text{false}, \text{true}) = \widehat{\text{IB}}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{av}^t : E \rightarrow (\text{IB} \rightarrow \text{IB}) \text{ where}$$

$$\forall e \in E \forall b \in \text{IB}. \llbracket e \rrbracket_{av}^t(b) =_{df} (b \vee \text{Comp}_e^t) \wedge \text{Transp}_e^t$$

- ▶ Initial information: $b_s \in \text{IB}$
- ▶ Direction of information flow: forward

In total

- ▶ Availability Specification: $\mathcal{S}_G^{av,t} = (\widehat{\text{IB}}, \llbracket \cdot \rrbracket_{av}^t, b_s, fw)$

Example: Availability of a Term t (2)

...where \widehat{IB} denotes the data-flow lattice and $Comp_e^t$, Mod_e^t , and $Transp_e^t$ three local predicates associated with edges.

▶ $\widehat{IB} =_{df} (IB, \wedge, \vee, \leq, false, true)$

...lattice of Boolean truth values: least element *false*, greatest element *true*, $false \leq true$, logical \wedge and logical \vee as meet and join operation, respectively.

- ▶ $Comp_e^t$...is *true*, if t is *computed* by the instruction at edge e , otherwise *false*.
- ▶ Mod_e^t ...is *true*, if t is *modified* by the instruction at edge e , otherwise *false*.
- ▶ $Transp_e^t =_{df} \neg Mod_e^t$...is *true*, if e is *transparent* for t (i.e., no operand of t is assigned a new value by the instruction at edge e), otherwise *false*.

Practically Relevant DFA Problems

...DFA problems are (only) practically relevant, if they are

- ▶ monotonic
- ▶ distributive (additive)

and satisfy the

- ▶ descending (ascending) chain condition.

Example: Availability of a Term t (1)

Lemma 3.2.3 (Data Flow Functions)

$$\forall e \in E. \llbracket e \rrbracket_{av}^t = \begin{cases} Cst_{true} & \text{if } Comp_e^t \wedge Transp_e^t \\ Id_{IB} & \text{if } \neg Comp_e^t \wedge Transp_e^t \\ Cst_{false} & \text{otherwise} \end{cases}$$

where

- ▶ $Cst_{true}, Cst_{false} : IB \rightarrow IB$ (constant functions on IB)

$$Cst_{true} =_{df} \lambda b. true$$

$$Cst_{false} =_{df} \lambda b. false$$

- ▶ $Id_{IB} : IB \rightarrow IB$ (identity on IB)

$$Id_{IB} =_{df} \lambda b. b$$

Example: Availability of a Term t (2)

Lemma 3.2.4 (Descending Chain Condition)

\widehat{B} satisfies the descending chain condition.

Lemma 3.2.5 (Distributivity)

$\llbracket e \rrbracket_{av}^t$, $e \in E$, is distributive.

Proof. Immediately with Lemma 3.2.3.

Corollary 3.2.6 (Monotonicity)

$\llbracket e \rrbracket_{av}^t$, $e \in E$, is monotonic.

Properties of DFA Functionals

Definition 3.2.7 (Properties of DFA Functionals)

Let $\mathcal{S}_G =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket, c_s, d)$ be a DFA specification for G .

The DFA functional $\llbracket \cdot \rrbracket : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ of \mathcal{S}_G is

- ▶ **monotonic/distributive/additive**

iff for every $e \in E$ the local DFA function $\llbracket e \rrbracket$ is

- ▶ **monotonic/distributive/additive**

respectively.

Properties of DFA Problems

Definition 3.2.8 (Properties of DFA Problems)

Let $\mathcal{S}_G =_{df} (\hat{C}, \llbracket \cdot \rrbracket, c_s, d)$ be a DFA specification for G .

The DFA problem induced by \mathcal{S}_G

- ▶ is **monotonic/distributive/additive** iff the DFA functional $\llbracket \cdot \rrbracket$ of \mathcal{S}_G is monotonic/distributive/additive, respectively.
- ▶ **satisfies the descending (ascending) chain condition** iff the DFA lattice \hat{C} of \mathcal{S}_G satisfies the descending (ascending) chain condition.

Example: Availability of a Term t

Corollary 3.2.9 (Availability is distributive)

The DFA problem induced by the DFA specification $\mathcal{S}_G^{av,t} = (\widehat{B}, \llbracket \rrbracket_{av}^t, b_s, fw)$ for the availability of a term t is

- ▶ distributive
- ▶ satisfies the descending chain condition.

Towards a Global Abstract Semantics

...globalizing a local abstract semantics for instructions to a global abstract semantics for flow graphs.

Actually, we introduce two globalization approaches:

- ▶ Meet over all Paths (*MOP*) Approach
 \rightsquigarrow defines the specifying solution of a DFA problem
- ▶ Maximum Fixed Point (*MaxFP*) Approach
 \rightsquigarrow induces a computable solution of a DFA problem

Chapter 3.3

The Meet Over All Paths Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

238/185

The Meet Over All Paths (MOP) Approach

Let $\mathcal{S}_G =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket, c_s, fw)$ be a DFA specification.

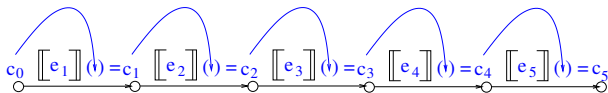
Definition 3.3.1 (Extending $\llbracket \cdot \rrbracket$ to Paths)

The DFA functions $\llbracket e \rrbracket$, $e \in E$, are extended onto paths $p = \langle e_1, e_2, \dots, e_q \rangle$ in G by defining:

$$\llbracket p \rrbracket =_{df} \begin{cases} Id_{\mathcal{C}} & \text{if } q < 1 \\ \llbracket \langle e_2, \dots, e_q \rangle \rrbracket \circ \llbracket e_1 \rrbracket & \text{otherwise} \end{cases}$$

where $Id_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$ denotes the identical mapping on \mathcal{C} , i.e., $Id_{\mathcal{C}} = \lambda c. c$.

Illustrating the extension of $\llbracket \cdot \rrbracket$ from edges to paths:



The Meet Over All Paths (*MOP*) Solution

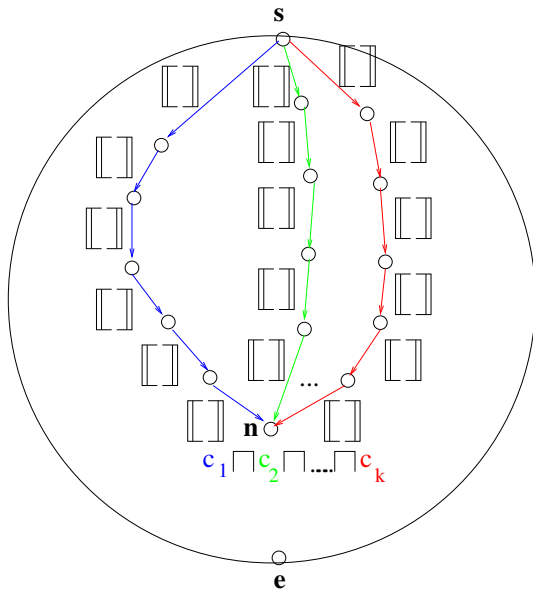
Definition 3.3.2 (The *MOP* Solution)

The *MOP* solution of \mathcal{S}_G is defined by:

$$MOP_{\mathcal{S}_G} : N \rightarrow \mathcal{C}$$

$$\forall n \in N. MOP_{\mathcal{S}_G}(n) =_{df} \bigsqcap \{ \llbracket p \rrbracket(c_s) \mid p \in \mathbf{P}[s, n] \}$$

Illustrating *MOP* Approach and *MOP* Solution



The Specifying Solution of a DFA Problem

...as illustrated by the previous figure:

- ▶ The *MOP solution* is for every program point n the
 - ▶ *strongest* DFA information valid at n (wrt \mathcal{S}_G).

This gives rise to consider the *MOP solution* the

- ▶ *specifying solution* of a DFA problem.

Conservative and Optimal DFA Algorithms

Definition 3.3.4 (Conservative DFA Algorithm)

A DFA algorithm A is *MOP* conservative for \mathcal{S}_G , if A terminates with a lower approximation of the *MOP* solution of \mathcal{S}_G .

Definition 3.3.5 (Optimal DFA Algorithm)

A DFA algorithm A is *MOP* optimal for \mathcal{S}_G , if A terminates with the *MOP* solution of \mathcal{S}_G .

Unfortunately

...the *MOP* approach itself does not induce an

- ▶ effective computation procedure

for computing the *MOP* solution (think of loops in a flow graph).

Even worse, the *MOP* solution is

- ▶ not even decidable!

Undecidability of the *MOP* Solution

Theorem 3.3.3 (Undecidability, Kam&Ullman 1977)

There is no algorithm A satisfying:

- ▶ The input of A are
 - ▶ a DFA specification $\mathcal{S}_G = (\hat{C}, \llbracket \ \rrbracket, c_s, fw)$
 - ▶ algorithms for the computation of the meet, the equality test, and the application of monotonic functions on the elements of a complete lattice
- ▶ The output of A is the *MOP* solution of \mathcal{S}_G .

(John B. Kam, Jeffrey D. Ullman. [Monotone Data Flow Analysis Frameworks](#). Acta Informatica 7, 305-317, 1977)

Towards a Conservative and Optimal DFA Alg.

Because of the preceding negative result(s) we introduce in addition to the *MOP* approach an orthogonal *second globalization approach* of a local abstract semantics, the

- ▶ Maximum Fixed Point (*MaxFP*) Approach.

The *MaxFP* approach leads to the

- ▶ Maximum Fixed Point (*MaxFP*) Solution

of a DFA problem and an

- ▶ effective computation procedure

computing the *MaxFP* solution (under certain conditions).

Chapter 3.4

The Maximum Fixed Point Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

247/185

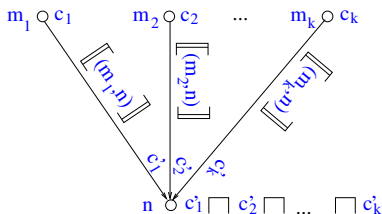
The Maximum Fixed Point (*MaxFP*) Approach

Let $\mathcal{S}_G =_{df} (\hat{C}, \llbracket \cdot \rrbracket, c_s, fw)$ be a DFA specification.

Equation System 3.4.1 (*MaxFP* Equation System)

$$inf(n) = \begin{cases} c_s & \text{if } n = s \\ \bigcap \{ \llbracket (m, n) \rrbracket (inf(m)) \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Illustrating the *MaxFP* Approach ($n \neq s$):



The Maximum Fixed Point (*MaxFP*) Solution

Let

$$\blacktriangleright \nu\text{-inf}_{c_s}(n), n \in N$$

denote the greatest solution of Equation System 3.4.1.

Definition 3.4.2 (The *MaxFP* Solution)

The *MaxFP* solution of S_G is defined by:

$$\text{MaxFP}_{S_G} : N \rightarrow \mathcal{C}$$

$$\forall n \in N. \text{MaxFP}_{S_G}(n) =_{df} \nu\text{-inf}_{c_s}(n)$$

The *MaxFP* Approach

...is practically relevant because the *MaxFP* Equation System 3.4.1 induces a generic

- ▶ *iterative computation procedure* (Algorithm 3.4.3)

approximating its greatest solution, i.e., the *MaxFP* solution.

The Generic Fixed Point Algorithm 3.4.3 (1)

Input: A DFA specification $\mathcal{S}_G =_{df} (\hat{C}, \llbracket \cdot \rrbracket, c_s, d)$. If $d = bw$, G_{rev} is used by the algorithm instead of G .

Output: On termination of the algorithm (cf. Termination Theorem 3.4.4), the variables $inf[n]$ store the *MaxFP* solution of \mathcal{S}_G at node n .

Additionally, we have (cf. Safety Theorem 3.5.1 and Coincidence Theorem 3.5.2): If

- ▶ $\llbracket \cdot \rrbracket$ distributive: $inf[n]$ stores
- ▶ $\llbracket \cdot \rrbracket$ monotonic: $inf[n]$ stores a lower approximation of the *MOP* solution of \mathcal{S}_G at node n .

Remark: The variable *workset* controls the iterative process. It temporarily stores a set of nodes of G , whose annotations have recently been changed and thus can impact the annotations of the nodes of their neighbourhood.

The Generic Fixed Point Algorithm 3.4.3 (2)

(Prologue: Initializing *inf* and *workset*)

FORALL $n \in N \setminus \{s\}$ DO $inf[n] := \top$ OD;

$inf[s] := c_s$;

$workset := N$;

(Main loop: The iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

 CHOOSE $m \in workset$;

$workset := workset \setminus \{m\}$;

 (Updating the annotations of all successors of node m)

 FORALL $n \in succ(m)$ DO

$meet := \llbracket (m, n) \rrbracket (inf[m]) \sqcap inf[n]$;

 IF $inf[n] \sqsupset meet$

 THEN

$inf[n] := meet$;

$workset := workset \cup \{n\}$

 FI

 OD ESOOHC OD.

Termination

Theorem 3.4.4 (Termination)

The Generic Fixed Point Algorithm 3.4.3 terminates with the *MaxFP* solution of \mathcal{S}_G , if

1. $\llbracket \cdot \rrbracket$ is monotonic
2. $\hat{\mathcal{C}}$ satisfies the descending chain condition.

The Computable Solution of a DFA Problem

...together the [Generic Fixed Point Algorithm 3.4.3](#) and the [Termination Theorem 3.4.4](#) give rise to consider the *MaxFP* solution a/the

- ▶ [computable solution](#) of a DFA problem.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

254/185

Flow Sensitivity and May/Must Problems

For flow-sensitive DFA problems we must distinguish (cf. Chapter 2)

- ▶ may/must forward problems (e.g., RD, AE)
- ▶ may/must backward problems (e.g., LV, VB)

Obviously, the Generic Fixed Point Algorithm 3.4.3 is formulated for

- ▶ must forward problems.

This raises the question:

- ▶ How can we handle instances of the other three kinds?

Uniform Handling of All Four Problem Kinds

The [Generic Fixed Point Algorithm 3.4.3](#) allows us to handle instances of all four problem kinds [uniformly](#):

- ▶ [must/forward](#): Directly.
- ▶ [may/forward](#): Defining \sqsubseteq in terms of \sqsupseteq .
- ▶ [must/backward](#): Using G_{rev} instead of G .
- ▶ [may/backward](#): Using G_{rev} instead of G and defining \sqsubseteq in terms of \sqsupseteq .

...this will be illustrated in detail in [Chapter 4](#).

Chapter 3.5

Safety and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

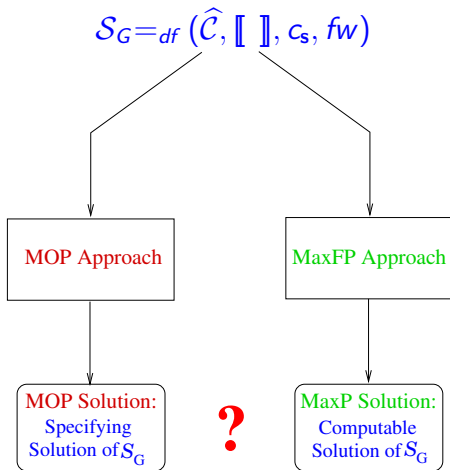
Chap. 9

Chap. 10

257/185

MOP / MaxFP Solution of a DFA Specification

...how are they related?



Safety

Theorem 3.5.1 (Safety)

The *MaxFP* solution of \mathcal{S}_G is a *safe* (i.e., lower) approximation of the *MOP* solution of \mathcal{S}_G , i.e.,

$$\forall n \in N. \text{MaxFP}_{\mathcal{S}_G}(n) \sqsubseteq \text{MOP}_{\mathcal{S}_G}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket$ is monotonic.

Coincidence

Theorem 3.5.2 (Coincidence)

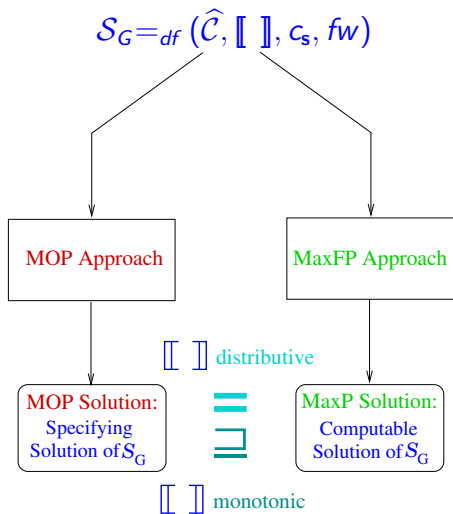
The *MaxFP* solution of \mathcal{S}_G and the *MOP* solution of \mathcal{S}_G coincide, i.e.,

$$\forall n \in N. \text{MaxFP}_{\mathcal{S}_G}(n) = \text{MOP}_{\mathcal{S}_G}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket$ is distributive.

MOP / MaxFP Solution of a DFA Specification

...and their relationship:



Conservativity, Optimality of Algorithm 3.4.3

Corollary 3.5.3 (*MOP* Conservativity)

Algorithm 3.4.3 is *MOP* conservative for \mathcal{S}_G (i.e., it terminates with a lower approximation of the *MOP* solution of \mathcal{S}_G), if $\llbracket \cdot \rrbracket$ is monotonic and $\hat{\mathcal{C}}$ satisfies the descending chain condition.

Corollary 3.5.4 (*MOP* Optimality)

Algorithm 3.4.3 is *MOP* optimal for \mathcal{S}_G (i.e., it terminates with the *MOP* solution of \mathcal{S}_G), if $\llbracket \cdot \rrbracket$ is distributive and $\hat{\mathcal{C}}$ satisfies the descending chain condition.

Chapter 3.6

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

263/185

Soundness and Completeness (1)

Analysis Scenario:

- ▶ Let ϕ be a program property of interest (e.g., **availability** of an expression, **liveness** of a variable, etc.).
- ▶ Let \mathcal{S}_G^ϕ be a DFA specification designed for ϕ .

Definition 3.6.1 (Soundness)

\mathcal{S}_G^ϕ is **sound** for ϕ , if, whenever the *MOP* solution of \mathcal{S}_G^ϕ indicates that ϕ is valid, then ϕ is valid.

Definition 3.6.2 (Completeness)

\mathcal{S}_G^ϕ is **complete** for ϕ , if, whenever ϕ is valid, then the *MOP* solution of \mathcal{S}_G^ϕ indicates that ϕ is valid.

Soundness and Completeness (2)

Intuitively

- ▶ Soundness means: $MOP_{S_G^\phi}$ implies ϕ .
- ▶ Completeness means: ϕ implies $MOP_{S_G^\phi}$.

Soundness and Completeness (3)

If \mathcal{S}_G^ϕ is sound and complete for ϕ , this intuitively means:

We compute

- ▶ the property of interest,
- ▶ the whole property of interest,
- ▶ and only the property of interest.

In other words

- ▶ We compute the program property of interest accurately!

Chapter 3.7

A Uniform Framework and Toolkit View to DFA

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

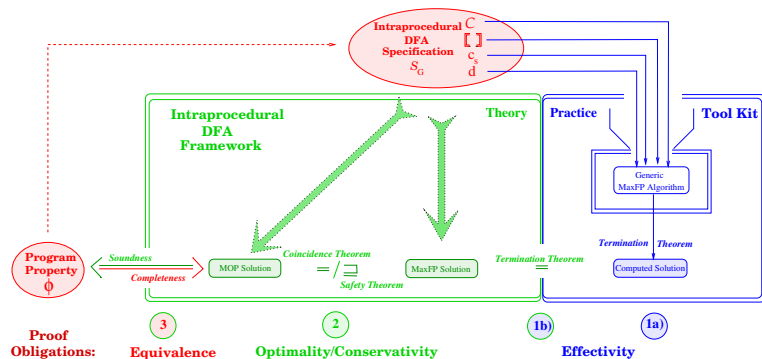
Chap. 10

267/185

Intraprocedural DFA: A Holistic Uniform View

...considering intraprocedural DFA from a holistic perspective:

► A Uniform Framework and Toolkit View



Intraprocedural DFA in Practice

...working with **framework** and **toolkit**, a **three-stage process**:

The Three-Stage Process

1. Identifying a Program Property of Interest

Identify a program property of interest (e.g., **availability of an expression**, **liveness of a variable**, etc.), say ϕ , and define ϕ formally.

2. Designing a DFA Specification

Design a DFA specification $\mathcal{S}_G^\phi = (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket, c_s, d)$ for ϕ .

3. Accomplishing Proof Obligations, Obtaining Guarantees

Verify a fixed set of proof obligations about the components of \mathcal{S}_G^ϕ and the relation of its *MOP* solution and ϕ to obtain guarantees that its *MaxFP* solution is **sound** or even **sound and complete** for ϕ .

Proof Obligations and Guarantees (1)

Proof obligations and guarantees in detail:

- ▶ Proof Obligations 1a), 1b): Descending Chain Condition for $\widehat{\mathcal{C}}$, Monotonicity of $\llbracket \cdot \rrbracket$

Guarantees:

- ▶ **Effectivity:** Termination of Algorithm 3.4.3 with the *MaxFP* solution of \mathcal{S}_G^ϕ .
- ▶ **Conservativity:** The *MaxFP* solution of \mathcal{S}_G^ϕ is *MOP* conservative.
- ▶ Proof Obligation 2): Distributivity of $\llbracket \cdot \rrbracket$

Guarantee:

- ▶ **Optimality:** The *MaxFP* solution of \mathcal{S}_G^ϕ is *MOP* optimal.

Proof Obligations and Guarantees (2)

- ▶ Proof Obligation 3): Equivalence of $MOP_{S_G^\phi}$ and ϕ

Guarantees:

- ▶ Whenever the *MOP* solution of S_G^ϕ indicates the validity of ϕ , then it is valid: **Soundness**.
 - ↪ We compute the property of interest, and only the property of interest.
- ▶ Whenever ϕ is valid, this is indicated by the *MOP* solution of S_G^ϕ : **Completeness**.
 - ↪ We compute the whole property of interest.

Guarantee of combined Soundness and Completeness:

- ▶ We compute program property ϕ accurately!

Chapter 3.8

Summary, Looking Ahead

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

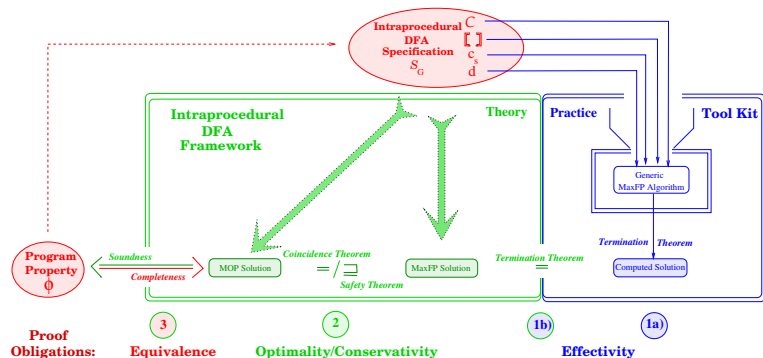
Chap. 8

Chap. 9

Chap. 10

272/185

The Holistic View of Intraprocedural DFA



...reconsidered from the angle of correctness and precision.

Reconsidering Correctness and Precision

Essentially, there are two sites where **correctness** and **precision issues** are handled in the **framework/toolkit** view of **DFA**:

Framework/Toolkit **internally**: Captured by

- ▶ **Safety** \rightsquigarrow Correctness
- ▶ **Coincidence** \rightsquigarrow Precision

...relating *MaxFP* and *MOP* solution.

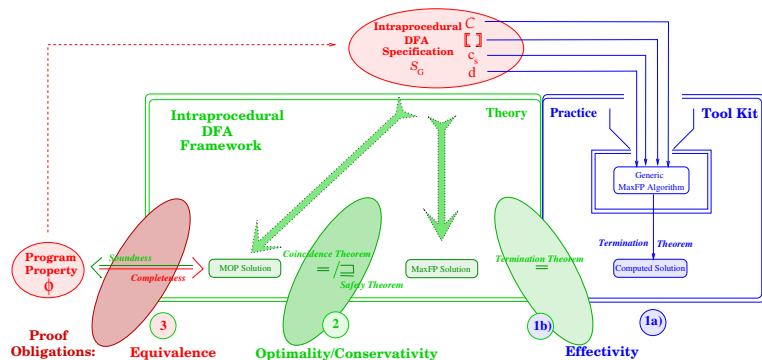
Framework/Toolkit **externally**: Captured by

- ▶ **Soundness** \rightsquigarrow Correctness
- ▶ **Completeness** \rightsquigarrow Precision

...relating *MOP* solution and ϕ .

Illustrating

...the sites of **internal** and **external** correctness and **precision** handling:

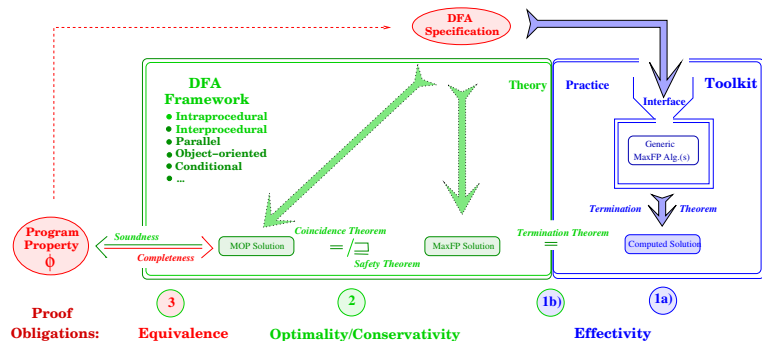


Outlook: The Holistic Uniform View to DFA

In the course of this lecture, we will see

- The Uniform Framework and Toolkit View of DFA

...is achievable beyond the base case of intraprocedural DFA.



Next

...we will consider **applications** of the intraproductural DFA framework for

- ▶ Gen/Kill DFA problems (cf. Chapter 4)
- ▶ Constant Propagation (cf. Chapter 5)

Chapter 3.9

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

3.9

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8




Chap. 9

Chap. 10




278/185

Further Reading for Chapter 3 (1)



Textbook Representations

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 1.2, The Structure of a Compiler; Chapter 1.4, The Science of Building a Compiler; Chapter 1.4.2, The Science of Code Optimization; Chapter 9.1, The Principal Sources of Program Optimization)
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Appendix B.3.1, Graphical Intermediate Representations)
-  Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.

Further Reading for Chapter 3 (2)





-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 3, Theoretical Abstractions in Data Flow Analysis; Chapter 4, General Data Flow Frameworks; Chapter 5, Complexity of Iterative Data Flow Analysis)
-  Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 2.3, Building the Flow Graph; Chapter 4.7, Structure of Program Flow Graph)
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 7, Control-Flow Analysis)

Further Reading for Chapter 3 (3)

-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 5, Static Program Analysis)
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 7, Program Analysis; Chapter 8, More on Program Analysis; Appendix B, Implementation of Program Analysis)

Further Reading for Chapter 3 (4)


Pioneering, Groundbreaking Articles

-  Frances E. Allen, John A. Cocke. *A Program Data Flow Analysis Procedure*. Communications of the ACM 19(3):137-147, 1976.
-  Susan Horwitz, Alan J. Demers, Tim Teitelbaum. *An Efficient General Iterative Algorithm for Dataflow Analysis*. Acta Informatica 24(6):679-694, 1987.
-  Gary A. Kildall. *A Unified Approach to Global Program Optimization*. In Conference Record of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73), 194-206, 1973.
-  John B. Kam, Jeffrey D. Ullman. *Global Data Flow Analysis and Iterative Algorithms*. Journal of the ACM 23:158-171, 1976.

Further Reading for Chapter 3 (5)

 John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. Acta Informatica 7:305-317, 1977.

Frameworks and Toolkits

 Marion Klein, Jens Knoop, Dirk Koschützki, Bernhard Steffen. *DFA&OPT-METAFrame: A Toolkit for Program Analysis and Optimization*. In Proceedings of the 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), Springer-V., LNCS 1055, 422-426, 1996.




 Jens Knoop. *From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Springer-V., LNCS 1579, 360-374, 1999.

Further Reading for Chapter 3 (6)


-  Thomas J. Marlowe, Barbara G. Ryder. *Properties of Data Flow Frameworks*. Acta Informatica 28(2):121-163, 1990.
-  Stephen P. Masticola, Thomas J. Marlowe, Barbara G. Ryder. *Lattice Frameworks for Multisource and Bidirectional Data Flow Problems*. ACM Transactions on Programming Languages and Systems (TOPLAS) 17(5):777-803, 1995.
-  Florian Martin. *PAG - An Efficient Program Analyzer Generator*. Journal of Software Tools for Technology Transfer 2(1):46-67, 1998.
-  Flemming Nielson. *Semantics-directed Program Analysis: A Tool-maker's Perspective*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 2-21, 1996.

Further Reading for Chapter 3 (7)



Solving Equation Systems, Computing Fixed Points

-  Christian Fecht, Helmut Seidl. *An Even Faster Solver for General Systems of Equations*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 189-204, 1996.
-  Christian Fecht, Helmut Seidl. *Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems*. In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 90-104, 1998.
-  Christian Fecht, Helmut Seidl. *A Faster Solver for General Systems of Equations*. *Science of Computer Programming* 35(2):137-161, 1999.

Further Reading for Chapter 3 (8)

-  Bernhard Steffen, Andreas Claßen, Marion Klein, Jens Knoop, Tiziana Margaria. *The Fixpoint Analysis Machine*. In Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95), Springer-V., LNCS 962, 72-87, 1995.

Flow Graph Pragmatics

-  Larry Carter, Jeanne Ferrante, Clark Thomborson. *Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger*. In Conference Record of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 106-114, 2003.
-  Jens Knoop, Dirk Koschützki, Bernhard Steffen. *Basic-block Graphs: Living Dinosaurs?* In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 65-79, 1998.



Further Reading for Chapter 3 (9)

Miscellaneous



Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeny, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Nathan Keynes, Nathaniel Nystrom, Andreas Zeller. *The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations*. ACM Transactions on Programming Languages and Systems 38(4), Article 15:1-20, 2016.

Further Reading for Chapter 3 (10)

-  William Landi. *Undecidability of Static Analysis*. ACM Letters on Programming Languages and Systems 1(4):323-337, 1992.
-  Janusz Laski, William Stanley. *Software Verification and Analysis*. Springer-V., 2009. (Chapter 7, What can one tell about a Program without its Execution: Static Analysis)

Chapter 4

Gen/Kill Analyses Reconsidered

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

289/185

Gen/Kill Analyses

...are common examples of

- ▶ distributive DFA problems.

In this chapter, we **reconsider Gen/Kill analyses** under the perspective of the

- ▶ Intraprocedural DFA Framework

of **Chapter 3** using

- ▶ **reaching definitions** (forward/may DFA problem)
- ▶ **very busy expressions** (backward/must DFA problem)

for illustration.

Remarks

Note that the DFA-problems of

- ▶ available expressions (forward/must DFA problem)
- ▶ live variables (backward/may DFA problem)

can be dealt with analogously.

Throughout Chapter 4, let

- ▶ $G = (N, E, s, e)$

be an edge-labelled flow graph.

Chapter 4.1

Reaching Definitions

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

292/185

Chapter 4.1.1

Reaching Definitions for a Single Definition

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

293/185

Scenario 1: The Setting (1)

...reaching definitions for a single definition $Def(v@ê) \equiv d_{ê}^V$.

Lattice

- ▶ $\widehat{IB}_V =_{df} (IB, \vee, \wedge, \geq, true, false)$

...lattice of Boolean truth values: least element *true*, greatest element *false*, $true \geq false$, logical \vee and logical \wedge as meet and join operation, respectively.

Special Functions

- ▶ Constant Functions $Cst_{true}, Cst_{false} : IB \rightarrow IB$

$$Cst_{true} =_{df} \lambda b. true$$

$$Cst_{false} =_{df} \lambda b. false$$

- ▶ Identity $Id_{IB} : IB \rightarrow IB$

$$Id_{IB} =_{df} \lambda b. b$$

Scenario 1: The Setting (2)

Let $\iota_e \equiv x := \text{exp}$ be the instruction at edge e .

Local Predicates

- ▶ $At_e^{\hat{e}}$
...*true*, if $e = \hat{e}$, otherwise *false*.
- ▶ Mod_e^v
...*true*, if v is **modified** by ι_e (i.e., ι_e assigns a new value to v), otherwise *false*.

Scenario 1: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\mathbb{B}, \vee, \wedge, \geq, \text{true}, \text{false}) = \widehat{\mathbb{B}}_{\vee}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{rd}^{d_e^v} : E \rightarrow (\mathbb{B} \rightarrow \mathbb{B}) \text{ where}$$

$$\forall e \in E \forall b \in \mathbb{B}. \llbracket e \rrbracket_{rd}^{d_e^v}(b) =_{df} (b \wedge \neg \text{Mod}_e^v) \vee \text{At}_e^{\hat{e}}$$

- ▶ Initial information: $b_s \in \mathbb{B}$
- ▶ Direction of information flow: forward

Reaching Definitions Specification for d_e^v

- ▶ Specification: $\mathcal{S}_G^{rd, d_e^v} = (\widehat{\mathbb{B}}_{\vee}, \llbracket \cdot \rrbracket_{rd}^{d_e^v}, b_s, fw)$

Towards Termination and Optimality

Lemma 4.1.1.1 (Data Flow Functions)

$$\forall e \in E. \llbracket e \rrbracket_{rd}^{d_e^V} = \begin{cases} Cst_{true} & \text{if } At_e^{\hat{e}} \\ Id_{IB} & \text{if } \neg At_e^{\hat{e}} \wedge \neg Mod_e^V \\ Cst_{false} & \text{otherwise} \end{cases}$$

Lemma 4.1.1.2 (Descending Chain Condition)

\widehat{IB}_V satisfies the descending chain condition (wrt $\sqsupseteq =_{df} \leq$).

Lemma 4.1.1.3 (Distributivity)

$\llbracket \rrbracket_{rd}^{d_e^V}$ is distributive (wrt $\sqcap =_{df} \vee$).

Proof. Immediately with Lemma 4.1.1.1.

Corollary 4.1.1.4 (Monotonicity)

$\llbracket \rrbracket_{rd}^{d_e^V}$ is monotonic.

Termination and Optimality

Theorem 4.1.1.5 (Termination)

Applied to $\mathcal{S}_G^{rd, d_e^v} = (\widehat{\text{IB}}_{\vee}, \llbracket \rrbracket_{rd}^{d_e^v}, b_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{rd, d_e^v}$.

Proof. Immediately with Lemma 4.1.1.2, Corollary 4.1.1.4, and Termination Theorem 3.4.4.

Theorem 4.1.1.6 (Optimality)

Applied to $\mathcal{S}_G^{rd, d_e^v} = (\widehat{\text{IB}}_{\vee}, \llbracket \rrbracket_{rd}^{d_e^v}, b_s, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{rd, d_e^v}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{rd, d_e^v}$).

Proof. Immediately with Lemma 4.1.1.3, Coincidence Theorem 3.5.2, and Termination Theorem 4.3.1.5.

Chapter 4.1.2

Reaching Definitions for a Set of Definitions

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

299/185

Scenario 2: The Setting

...reaching definitions for a set of definitions

$$\{d_{\hat{e}_1}^{V_1}, \dots, d_{\hat{e}_k}^{V_k}\} \equiv \mathcal{D}_{\hat{E}}^V, k \in \mathbb{N}.$$

Lattice

► $\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup} =_{df} (\mathcal{P}(\mathcal{D}_{\hat{E}}^V), \cup, \cap, \supseteq, \mathcal{D}_{\hat{E}}^V, \emptyset)$

...power set lattice over $\mathcal{D}_{\hat{E}}^V$: least element $\mathcal{D}_{\hat{E}}^V$, greatest element \emptyset , superset relation \supseteq as ordering relation, set union \cup and set intersection \cap as meet and join operation, respectively.

Scenario 2: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathcal{P}(\mathcal{D}_{\hat{E}}^V), \cup, \cap, \supseteq, \mathcal{D}_{\hat{E}}^V, \emptyset) = \widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V} : E \rightarrow (\mathcal{P}(\mathcal{D}_{\hat{E}}^V) \rightarrow \mathcal{P}(\mathcal{D}_{\hat{E}}^V)) \text{ where}$$

$$\forall e \in E \forall \mathcal{D} \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V). \llbracket e \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V}(\Delta) =_{df} \{d_e^V \in \mathcal{D}_{\hat{E}}^V \mid (d_e^V \in \Delta \wedge \neg \text{Mod}_e^V) \vee \text{At}_e^{\hat{E}}\}$$

- ▶ Initial information: $\mathcal{D}_s \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V)$
- ▶ Direction of information flow: forward

Reaching Definitons Specification for $\mathcal{D}_{\hat{E}}^V$

- ▶ Specification: $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V} = (\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}, \llbracket \cdot \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V}, \mathcal{D}_s, fw)$

Towards Termination and Optimality

Lemma 4.1.2.1 (Descending Chain Condition)

$\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}$ satisfies the descending chain condition (wrt $\sqsubseteq =_{df} \supseteq$).

Lemma 4.1.2.2 (Distributivity)

$\llbracket \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V}$ is distributive (wrt $\sqcap =_{df} \cup$).

Corollary 4.1.2.3 (Monotonicity)

$\llbracket \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V}$ is monotonic.

Termination and Optimality

Theorem 4.1.2.4 (Termination)

Applied to $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V} = (\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}, \llbracket \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V}, \mathcal{D}_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V}$.

Proof. Immediately with Lemma 4.1.2.1, Corollary 4.1.2.3, and Termination Theorem 3.4.4.

Theorem 4.1.2.5 (Optimality)

Applied to $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V} = (\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}, \llbracket \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V}, \mathcal{D}_s, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V}$).

Proof. Immediately with Lemma 4.1.2.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.1.2.4.

Chapter 4.1.3

Reaching Definitions for a Set of Definitions: Bitvector Implementation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

304/185

Scenario 3: The Setting (1)

...reaching definitions for a set of definitions

$$\{d_{\hat{e}_1}^{v_1}, \dots, d_{\hat{e}_k}^{v_k}\} \equiv \mathcal{D}_{\hat{E}}^V, k \in \mathbb{N}.$$

Lattice

► $\widehat{\mathbb{B}}_{\vee}^n =_{df} (\mathbb{B}^n, \vee_{pw}, \wedge_{pw}, \geq_{pw}, \overline{true}, \overline{false})$

...*n*-ary cross-product lattice over \mathbb{B} : least element

$\overline{true} =_{df} (true, \dots, true) \in \mathbb{B}^n$, greatest element

$\overline{false} =_{df} (false, \dots, false) \in \mathbb{B}^n$, ordering relation \geq_{pw} as

pointwise extension of \geq from $\widehat{\mathbb{B}}_{\vee}$ to $\widehat{\mathbb{B}}_{\vee}^n$, \vee_{pw} and

\wedge_{pw} as pointwise extensions of logical \vee and logical \wedge

from $\widehat{\mathbb{B}}_{\vee}$ to $\widehat{\mathbb{B}}_{\vee}^n$ as meet and join operation, respectively.

Scenario 3: The Setting (2)

Auxiliary Functions

- ▶ $var : \mathcal{D}_{\hat{E}}^V \rightarrow V$, $edge : \mathcal{D}_{\hat{E}}^V \rightarrow E$ defined by
 $\forall d_{\hat{e}}^v \in \mathcal{D}_{\hat{E}}^V. var(d_{\hat{e}}^v) =_{df} v$, $edge(d_{\hat{e}}^v) =_{df} \hat{e}$

- ▶ $ix : \mathcal{D}_{\hat{E}}^V \rightarrow \{1, \dots, n\}$, $ix^{-1} : \{1, \dots, n\} \rightarrow \mathcal{D}_{\hat{E}}^V$

...bijective mappings which map every definition $d_{\hat{e}}^v \in \mathcal{D}_{\hat{E}}^V$ to a number in $\{1, \dots, n\}$ and vice versa.

The $ix(d_{\hat{e}}^v)^{th}$ element of an element

$$\bar{b} = (b_1, \dots, b_{ix(d_{\hat{e}}^v)}, \dots, b_n) \in \mathbb{B}^n$$

is the reaching definitions information for $d_{\hat{e}}^v$ stored in \bar{b} .

- ▶ $\cdot \downarrow_i : \mathbb{B}^n \rightarrow \{1, \dots, n\} \rightarrow \mathbb{B}$

...projection function which yields the i^{th} element of an element $\bar{b} \in \mathbb{B}^n$, i.e., $\forall i \in \{1, \dots, n\}. \bar{b} \downarrow_i =_{df} b_i$.

Scenario 3: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\mathbb{B}^n, \vee_{pw}, \wedge_{pw}, \geq_{pw}, \overline{true}, \overline{false}) = \widehat{\mathbb{B}}^n_{\vee}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{rd, cps}^{\mathcal{D}_{\hat{E}}^V} : E \rightarrow (\mathbb{B}^n \rightarrow \mathbb{B}^n) \text{ where}$$

$$\forall e \in E \forall d_{\hat{e}}^V \in \mathbb{B}^n. \llbracket e \rrbracket_{rd, cps}^{\mathcal{D}_{\hat{E}}^V}(\bar{b}) =_{df} \bar{b}'$$

$$\text{where } \forall i \in \{1, \dots, n\}. \bar{b}' \downarrow_i =_{df}$$

$$(\bar{b} \downarrow_i \wedge \neg \text{Mod}_e^{\text{var}(ix^{-1}(i))}) \vee \text{At}_e^{\text{edge}(ix^{-1}(i))})$$

- ▶ Initial information: $\bar{b}_s \in \mathbb{B}^n$

- ▶ Direction of information flow: forward

R'ing Def's Specification for $\mathcal{D}_{\hat{E}}^V$, cross-product spec. (cps)

- ▶ Specification: $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, cps} = (\widehat{\mathbb{B}}^n_{\vee}, \llbracket \cdot \rrbracket_{rd, cps}^{\mathcal{D}_{\hat{E}}^V}, \bar{b}_s, fw)$

Scenario 3: Bitvector Implementation (1)

Bitvector Implementation of $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, cps}$

- ▶ $\widehat{\mathcal{IB}}_V^n$ can efficiently be implemented in terms of bitvectors $\vec{bv} = [d_1, \dots, d_n]$, $d_i \in \{0, 1\}$, $1 \leq i \leq n$, of length n .
- ▶ Let \mathcal{BV}^n denote the set of all bitvectors of length n .
- ▶ Let $\vec{bv}[i] = d_i$ for all $\vec{bv} = [d_1, \dots, d_n] \in \mathcal{BV}^n$, $1 \leq i \leq n$.
- ▶ Let $\vec{0} =_{df} [0, \dots, 0] \in \mathcal{BV}^n$ and $\vec{1} =_{df} [1, \dots, 1] \in \mathcal{BV}^n$.
- ▶ Let $\min_{\mathcal{BV}}$ and $\max_{\mathcal{BV}}$ be the **bitwise minimum** ('logical \wedge ') and the **bitwise maximum function** ('logical \vee ') over bitvectors, i.e., $\forall \vec{bv}_1, \vec{bv}_2 \in \mathcal{BV}^n \forall i \in \{1, \dots, n\}$.
 - ▶ $(\vec{bv}_1 \min_{\mathcal{BV}} \vec{bv}_2)[i] =_{df} \min(\vec{bv}_1[i], \vec{bv}_2[i])$
 - ▶ $(\vec{bv}_1 \max_{\mathcal{BV}} \vec{bv}_2)[i] =_{df} \max(\vec{bv}_1[i], \vec{bv}_2[i])$

Scenario 3: Bitvector Implementation (2)

Auxiliary Functions

$$\blacktriangleright V_{\mathcal{D}} =_{df} \{var(d_{\hat{e}}^V) \mid d_{\hat{e}}^V \in \mathcal{D}_{\hat{E}}^V\},$$
$$\hat{E}_{\mathcal{D}} =_{df} \{edges(d_{\hat{e}}^V) \mid d_{\hat{e}}^V \in \mathcal{D}_{\hat{E}}^V\}$$

$$\blacktriangleright ix : \mathcal{D}_{\hat{E}}^V \rightarrow \{1, \dots, n\}, ix^{-1} : \{1, \dots, n\} \rightarrow \mathcal{D}_{\hat{E}}^V$$

...bijective mappings which map every definition $d_{\hat{e}}^V \in \mathcal{D}_{\hat{E}}^V$ to a number in $\{1, \dots, n\}$ and vice versa.

The $ix(d_{\hat{e}}^V)^{th}$ element of a bitvector

$$\vec{bv} = [d_1, \dots, d_{ix(d_{\hat{e}}^V)}, \dots, d_n] \in \mathcal{BV}^n$$

is the reaching definitions information for $d_{\hat{e}}^V$ stored in \vec{bv} .

Scenario 3: Bitvector Implementation (3)

Extending Local Predicates to Bitvectors

$$\blacktriangleright \overset{\rightarrow}{Mod}_e^{V_{\mathcal{D}}} \in \mathcal{BV}^n$$

$$\forall i \in \{1, \dots, n\}. \overset{\rightarrow}{Mod}_e^{V_{\mathcal{D}}}[i] =_{df} \begin{cases} 1 & \text{if } Mod_e^{var(ix^{-1}(i))} \\ 0 & \text{otherwise} \end{cases}$$

$$\blacktriangleright \overset{\rightarrow}{At}_e^{\hat{E}_{\mathcal{D}}} \in \mathcal{BV}^n$$

$$\forall i \in \{1, \dots, n\}. \overset{\rightarrow}{At}_e^{\hat{E}_{\mathcal{D}}}[i] =_{df} \begin{cases} 1 & \text{if } At_e^{edge(ix^{-1}(i))} \\ 0 & \text{otherwise} \end{cases}$$

Scenario 3: Bitvector Implementation (4)

Bitvector Implementation

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\mathcal{BV}^n, \max_{\mathcal{BV}}, \min_{\mathcal{BV}}, \geq_{\mathcal{BV}}, \vec{1}, \vec{0}) = \widehat{\mathcal{BV}^n}_{max}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V} : E \rightarrow (\mathcal{BV}^n \rightarrow \mathcal{BV}^n) \text{ where}$$

$$\forall e \in E \forall \vec{bv} \in \mathcal{BV}^n. \llbracket e \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V}(\vec{bv}) =_{df}$$

$$(\vec{bv} \min_{\mathcal{BV}} \neg \text{Mod}_e^{V_D}) \max_{\mathcal{BV}} \text{At}_e^{\hat{E}_D}$$

- ▶ Initial information: $\vec{bv}_s \in \mathcal{BV}^n$
- ▶ Direction of information flow: forward

R'ing Def's Spec. for $\mathcal{D}_{\hat{E}}^V$, bitvector implementation (bvi)

- ▶ Specification: $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, bvi} = (\widehat{\mathcal{BV}^n}_{max}, \llbracket \cdot \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V}, \vec{bv}_s, fw)$

Towards Termination and Optimality

Lemma 4.1.3.1 (Descending Chain Condition)

$\widehat{\mathcal{BV}}_{max}^n$ satisfies the descending chain condition (wrt $\sqsupseteq \stackrel{df}{=} \leq_{\mathcal{BV}}$).

Lemma 4.1.3.2 (Distributivity)

$\llbracket \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V}$ is distributive (wrt $\sqcap \stackrel{df}{=} \max_{\mathcal{BV}}$).

Corollary 4.1.3.3 (Monotonicity)

$\llbracket \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V}$ is monotonic.

Termination

Theorem 4.1.3.4 (Termination)

Applied to $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, bvi} = (\widehat{\mathcal{BV}}_{max}^n, \llbracket \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V}, \vec{bv}_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, bvi}$.

Proof. Immediately with Lemma 4.1.3.1, Corollary 4.1.3.3, and Termination Theorem 3.4.4.

Optimality

Theorem 4.1.3.5 (Optimality)

Applied to $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, bvi} = (\widehat{\mathcal{BV}}_{max}^n, \llbracket \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V}, \vec{bV}_s, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, bvi}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, bvi}$).

Proof. Immediately with Lemma 4.1.3.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.1.3.4.

Note

- ▶ All results of Chapter 4.1.3 hold for $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, cps} = (\widehat{\mathcal{IB}}_{\vee}^n, \llbracket \rrbracket_{rd, cps}^{\mathcal{D}_{\hat{E}}^V}, \vec{b}_s, fw)$, too.
- ▶ Applied to $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, bvi} = (\widehat{\mathcal{BV}}_{max}^n, \llbracket \rrbracket_{rd, bvi}^{\mathcal{D}_{\hat{E}}^V}, \vec{bV}_s, fw)$, Algorithm 3.4.3 takes advantage of the efficient bitvector operations of actual processors.

Chapter 4.1.4

Reaching Definitions for a Set of Definitions: Gen/Kill Implementation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

315/185

Scenario 4: The Setting

...reaching definitions for a set of definitions

$$\{d_{\hat{e}_1}^{v_1}, \dots, d_{\hat{e}_k}^{v_k}\} \equiv \mathcal{D}_{\hat{E}}^V, k \in \mathbb{N}.$$

Defining Gen/Kill Predicates

- ▶ $Gen_e^{\mathcal{D}_{\hat{E}}^V} =_{df} \{d_{\hat{e}}^v \in \mathcal{D}_{\hat{E}}^V \mid At_e^{\hat{e}}\}$
- ▶ $Kill_e^{\mathcal{D}_{\hat{E}}^V} =_{df} \{d_{\hat{e}}^v \in \mathcal{D}_{\hat{E}}^V \mid Mod_e^v\}$

Scenario 4: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathcal{P}(\mathcal{D}_{\hat{E}}^V), \cup, \cap, \supseteq, \mathcal{D}_{\hat{E}}^V, \emptyset) = \widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V} : E \rightarrow (\mathcal{P}(\mathcal{D}_{\hat{E}}^V) \rightarrow \mathcal{P}(\mathcal{D}_{\hat{E}}^V)) \text{ where}$$
$$\forall e \in E \forall \mathcal{D} \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V). \llbracket e \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V}(\mathcal{D}) =_{df}$$

$$(\mathcal{D} \setminus \text{Kill}_e^{\mathcal{D}_{\hat{E}}^V}) \cup \text{Gen}_e^{\mathcal{D}_{\hat{E}}^V}$$

- ▶ Initial information: $\mathcal{D}_s \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V)$
- ▶ Direction of information flow: forward

Reaching Definitions Specification for $\mathcal{D}_{\hat{E}}^V$

- ▶ Specification: $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, gk} = (\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)})_{\cup}, \llbracket \cdot \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V}, \mathcal{D}_s, fw$

Towards Termination and Optimality

Compare

► $\llbracket \cdot \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V} : E \rightarrow (\mathcal{P}(\mathcal{D}_{\hat{E}}^V) \rightarrow \mathcal{P}(\mathcal{D}_{\hat{E}}^V))$ where

$$\forall e \in E \forall \mathcal{D} \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V). \llbracket e \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V}(\mathcal{D}) =_{df}$$

$$(\mathcal{D} \setminus \text{Kill}_e^{\mathcal{D}_{\hat{E}}^V}) \cup \text{Gen}_e^{\mathcal{D}_{\hat{E}}^V}$$

► $\llbracket \cdot \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V} : E \rightarrow (\mathcal{P}(\mathcal{D}_{\hat{E}}^V) \rightarrow \mathcal{P}(\mathcal{D}_{\hat{E}}^V))$ where

$$\forall e \in E \forall \mathcal{D} \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V). \llbracket e \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V}(\mathcal{D}) =_{df}$$

$$\{d_{\hat{e}}^v \in \mathcal{D}_{\hat{E}}^V \mid (d_{\hat{e}}^v \in \mathcal{D} \wedge \neg \text{Mod}_e^v) \vee \text{At}_{\hat{e}}^v\}$$

Obviously

Lemma 4.1.4.1 (Equality)

$$\llbracket \cdot \rrbracket_{rd}^{\mathcal{D}_{\hat{E}}^V} = \llbracket \cdot \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V}$$

Termination and Optimality

Theorem 4.1.4.2 (Termination)

Applied to $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, gk} = (\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}, \llbracket \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V}, \mathcal{D}_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, gk}$.

Proof. Immediately with Lemma 4.1.4.1, Lemma 4.1.2.1, Corollary 4.1.2.3, and Termination Theorem 3.4.4.

Theorem 4.1.4.3 (Optimality)

Applied to $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, gk} = (\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_{\cup}, \llbracket \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V}, \mathcal{D}_s, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, gk}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{rd, \mathcal{D}_{\hat{E}}^V, gk}$).

Proof. Immediately with Lemma 4.1.4.1, Lemma 4.1.2.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.1.4.2.

Recalling the *MaxFP* Equation System

Equation System 3.4.1 (*MaxFP* Equation System)

$$\mathit{inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \prod \{ \mathbb{I}(m, n) \mathbb{I}(\mathit{inf}(m)) \mid m \in \mathit{pred}(n) \} & \text{otherwise} \end{cases}$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

320/185

The *MinFP* Equation System

Equation System 3.4.1_{min} (*MinFP* EQS)

$$\mathit{inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \sqcup \{ \llbracket (m, n) \rrbracket (\mathit{inf}(m)) \mid m \in \mathit{pred}(n) \} & \text{otherwise} \end{cases}$$

Specializing Equation System 3.4.1_{min} for reaching definitions yields:

EQS 4.1.4.4 (EQS 3.4.1_{min} for Reaching Def's)

$$\mathit{Reaches}(n) = \begin{cases} \mathcal{D}_s & \text{if } n = \mathbf{s} \\ \cup \{ \llbracket (m, n) \rrbracket_{rd, gk}^{\mathcal{D}_s^V} (\mathit{Reaches}(m)) \mid m \in \mathit{pred}(n) \} & \text{otherwise} \end{cases}$$

Specializing EQS 3.4.1_{min} for Reaching Def's

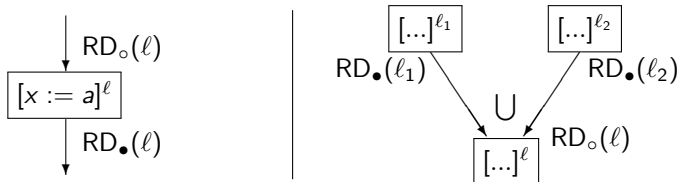
Expanding $\llbracket \rrbracket_{rd, gk}^{\mathcal{D}_{\hat{E}}^V}$ in Equation System 4.1.4.4 yields:

EQS 4.1.4.5 (EQS 3.4.1_{min} for Reaching Def's)

$Reaches(n) =$

$$\begin{cases} \mathcal{D}_s & \text{if } n = s \\ \bigcup \{ (Reaches(m) \setminus Kill_{(m,n)}^{\mathcal{D}_{\hat{E}}^V}) \cup Gen_{(m,n)}^{\mathcal{D}_{\hat{E}}^V} \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Recalling the RD Analysis of Chapter 2.1.1 (1)



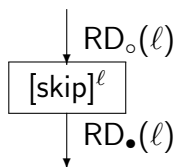
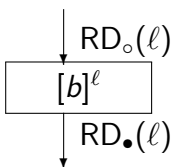
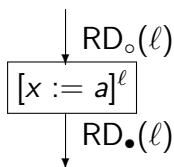
$$\begin{aligned}
 RD_o(\ell) &= \begin{cases} \emptyset & : \text{ if } \ell = \text{init}(S_*) \\ \bigcup \{RD_\bullet(\ell') \mid (\ell', \ell) \in \text{flow}(S_*)\} & : \text{ otherwise} \end{cases} \\
 RD_\bullet(\ell) &= (RD_o(\ell) \setminus \text{kill}_{RD}(B^\ell)) \cup \text{gen}_{RD}(B^\ell) \quad \text{where } B^\ell \in \text{blocks}(S_*)
 \end{aligned}$$

EQS 4.1.4.5 (EQS 3.4.1_{min} for Reaching Definitions) – recalled

$Reaches(n) =$

$$\begin{cases} \mathcal{D}_s & \text{if } n = s \\ \bigcup \{ (Reaches(m) \setminus Kill_{\hat{E}}^{D_V}) \cup Gen_{\hat{E}}^{D_V} \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Recalling the RD Analysis of Chapter 2.1.1 (2)



$$\text{gen}_{\text{RD}}([x := a]^\ell) = \{(x, \ell)\}$$

$$\text{gen}_{\text{RD}}([b]^\ell) = \emptyset$$

$$\text{gen}_{\text{RD}}([\text{skip}]^\ell) = \emptyset$$

$$\text{kill}_{\text{RD}}([x := a]^\ell) = \{(x, ?)\} \cup \{(x, \ell') \mid B^{\ell'} \text{ is assignment to } x\}$$

$$\text{kill}_{\text{RD}}([b]^\ell) = \emptyset$$

$$\text{kill}_{\text{RD}}([\text{skip}]^\ell) = \emptyset$$

Summing up

The [Reaching Definitions Equations](#) of

- ▶ [Equation System 4.1.4.5](#)
- ▶ [Chapter 2.2.1](#)

are [equivalent](#) up to the insignificant [formal difference](#) of considering

- ▶ [edge-labelled](#)
- ▶ [node-labelled](#)

[flow graphs](#), respectively.

Chapter 4.1.5

Reaching Definitions Analysis: Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

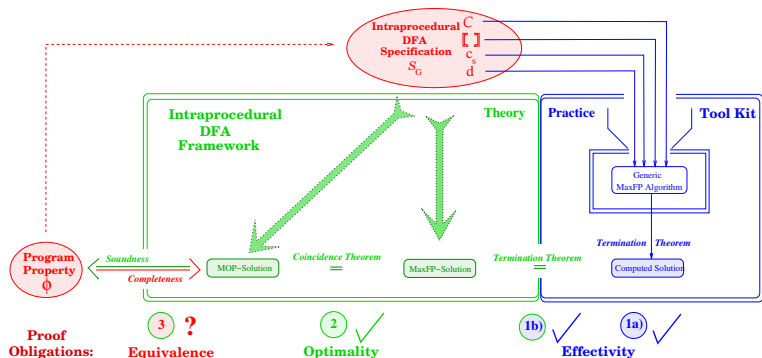
Chap. 9

Chap. 10

326/185

Closing the Proof Final Gap

...proving soundness and completeness of $\mathcal{S}_G^{rd, d_e^v}$ for the reaching definitions property:



Defining Reaching Definitions Informally

...intuitively:

- ▶ A **definition reaches a node** if there is a path from the node of the definition to the node without any redefinition of the left-hand side variable of the definition along this path (cf. [Definition 2.2.1.1](#)).

Note

- ▶ The informal 'definition' of reaching definitions does not foresee the possibility of a definition that reaches the procedure entry itself.
- ▶ Situations where this reaching definition property is ensured by the calling context of the procedure, are thus not captured and can not be dealt with.

Useful Notation

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, and let *Predicate* be a predicate defined for edges $e \in E$.

We define for paths:

- ▶ Let $p = \langle e_1, \dots, e_q \rangle \in \mathbf{P}[m, n]$.
 - ▶ p_i , $1 \leq i \leq q$, denotes the i^{th} edge e_i of p .
 - ▶ $p_{]k, l]}$ denotes the subpath $\langle e_{k+1}, \dots, e_l \rangle$ of p .
 - ▶ λ_p denotes the length of p , i.e., the number q of edges of p .

We define for paths and predicates:

- ▶ $\text{Predicate}_p^{\forall} \iff \forall 1 \leq i \leq \lambda_p. \text{Predicate}_{p_i}$
- ▶ $\text{Predicate}_p^{\exists} \iff \exists 1 \leq i \leq \lambda_p. \text{Predicate}_{p_i}$

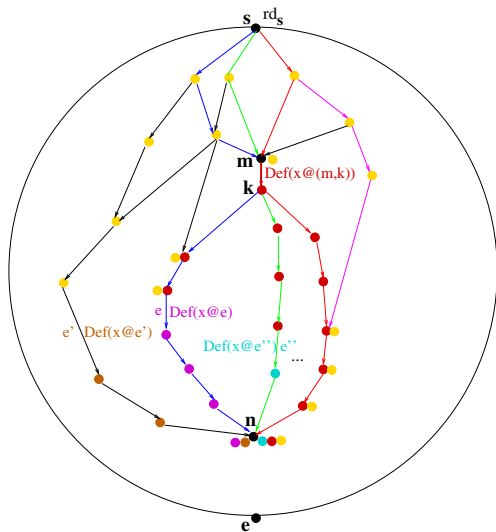
Defining Reaching Definitions Formally

Definition 4.1.5.1 (Reaching Definitions)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let $d_{\hat{e}}^v$ be a definition, let $rd_{\mathbf{s}} \in \mathbb{B}$ the reaching definitions information at \mathbf{s} ensured by the calling context of G .

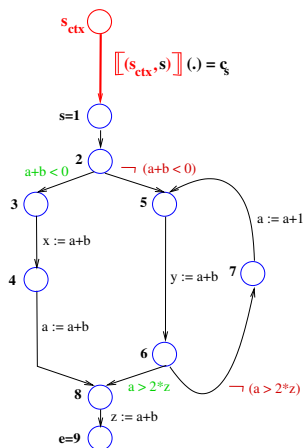
$$\text{Reaches}^{d_{\hat{e}}^v}(n) \iff_{df} \begin{cases} rd_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \exists p \in \mathbf{P}[\mathbf{s}, n]. \\ \quad (rd_{\mathbf{s}} \wedge \neg \text{Mod}_{p}^{v\forall}) \vee \\ \quad \exists i \leq \lambda_p. \text{At}_{p_i}^{\hat{e}} \wedge (i = \lambda_p \vee \neg \text{Mod}_{p]i, \lambda_p}^{v\forall}) & \text{otherwise} \end{cases}$$

Illustrating the Essence of Definition 4.1.5.1

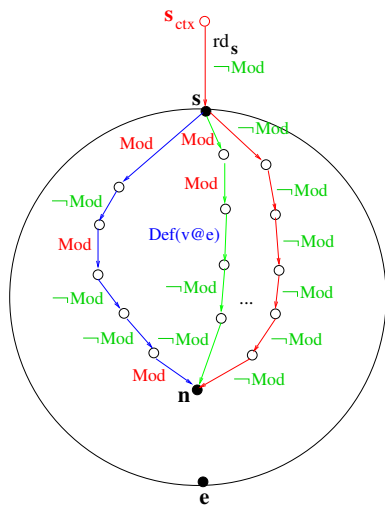


Context Edges

...introducing a **context edge** leads to a simpler uniform definition of **reaching definitions**:



Context Edges and Reaching Definitions



...leading to:

$$\forall n \in N \setminus \{s_{ctx}\}. \text{Reaches}^{dv}_{\hat{e}}(n) \iff_{df} \exists p \in \mathbf{P}[s_{ctx}, n]. \exists i \leq \lambda_p. \text{At}_{p_i}^{\hat{e}} \wedge (i = \lambda_p \vee \neg \text{Mod}^{v}_{p[i, \lambda_p]})$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

333/185

Closing the Final Proof Gap

Theorem 4.1.5.2 (Soundness and Completeness)

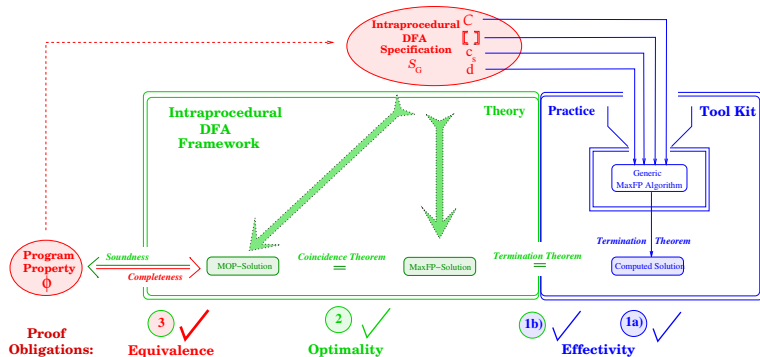
Let $G = (N, E, s, e)$ be a flow graph, let $d_{\hat{e}}^V$ be a definition, let $rd_s \in \mathbb{IB}$ be the reaching definitions information at s ensured by the calling context of G , and let $MOP_{S_G^{rd, d_{\hat{e}}^V}}$ be the MOP solution of the DFA specification $S_G^{rd, d_{\hat{e}}^V} = (\widehat{\mathbb{IB}}_{\vee}, [\llbracket \rrbracket_{rd}^{d_{\hat{e}}^V}, rd_s, fw])$.

Then:

$$\forall n \in N. \text{Reaches}^{d_{\hat{e}}^V}(n) \iff MOP_{S_G^{rd, d_{\hat{e}}^V}}(n)$$

Gap Closed: Soundness, Completeness Proven

...soundness and completeness of $\mathcal{S}_G^{rd, d_e^v}$ for reaching definitions proven:



Chapter 4.2

Very Busy Expressions

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

336/185

Chapter 4.2.1

Very Busyness for a Single Term

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

337/185

Scenario 1: The Setting (1)

...very busy for a single term t .

Lattice

- ▶ $\widehat{\text{IB}} =_{df} (\text{IB}, \wedge, \vee, \leq, \text{false}, \text{true})$

...lattice of Boolean truth values: least element *false*, greatest element *true*, $\text{false} \leq \text{true}$, logical \wedge and logical \vee as meet and join operation, respectively.

Special Functions

- ▶ Constant Functions $Cst_{\text{true}}, Cst_{\text{false}} : \text{IB} \rightarrow \text{IB}$

$$Cst_{\text{true}} =_{df} \lambda b. \text{true}$$

$$Cst_{\text{false}} =_{df} \lambda b. \text{false}$$

- ▶ Identity $Id_{\text{IB}} : \text{IB} \rightarrow \text{IB}$

$$Id_{\text{IB}} =_{df} \lambda b. b$$

Scenario 1: The Setting (2)

Let $\iota_e \equiv x := \text{exp}$ be the instruction at edge e .

Local Predicates

▶ Comp_e^t

...*true*, if t is **computed** by ι_e (i.e., t is a subterm of the right-hand side expression exp of ι_e), otherwise *false*.

▶ Mod_e^t

...*true*, if t is **modified** by ι_e (i.e., ι_e assigns a new value to some operand of t), otherwise *false*.

▶ $\text{Transp}_e^t =_{df} \neg \text{Mod}_e^t$

...*true*, if e is **transparent** for t (i.e., ι_e does not assign a new value to any operand of t), otherwise *false*.

Scenario 1: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathbb{B}, \wedge, \vee, \leq, \text{false}, \text{true}) = \widehat{\mathbb{B}}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{vb}^t : E \rightarrow (\mathbb{B} \rightarrow \mathbb{B}) \text{ where}$$

$$\forall e \in E \forall b \in \mathbb{B}. \llbracket e \rrbracket_{vb}^t(b) =_{df} (b \wedge \text{Transp}_e^t) \vee \text{Comp}_e^t$$

- ▶ Initial information: $b_e \in \mathbb{B}$
- ▶ Direction of information flow: backward

Very Busy Specification for t

- ▶ Specification: $\mathcal{S}_G^{vb,t} = (\widehat{\mathbb{B}}, \llbracket \cdot \rrbracket_{vb}^t, b_e, bw)$

Towards Termination and Optimality

Lemma 4.2.1.1 (Data Flow Functions)

$$\forall e \in E. \llbracket e \rrbracket_{vb}^t = \begin{cases} Cst_{true} & \text{if } Comp_e^t \\ Id_{\mathbb{B}} & \text{if } \neg Comp_e^t \wedge Transp_e^t \\ Cst_{false} & \text{otherwise} \end{cases}$$

Lemma 4.2.1.2 (Descending Chain Condition)

$\widehat{\mathbb{B}}$ satisfies the descending chain condition.

Lemma 4.2.1.3 (Distributivity)

$\llbracket \rrbracket_{vb}^t$ is distributive.

Proof. Immediately with Lemma 4.2.1.1.

Corollary 4.2.1.4 (Monotonicity)

$\llbracket \rrbracket_{vb}^t$ is monotonic.

Termination and Optimality

Theorem 4.2.1.5 (Termination)

Applied to $\mathcal{S}_G^{vb,t} = (\widehat{IB}, \llbracket \rrbracket_{vb}^t, b_e, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb,t}$.

Proof. Immediately with Lemma 4.2.1.2, Corollary 4.2.1.4, and Termination Theorem 3.4.4.

Theorem 4.2.1.6 (Optimality)

Applied to $\mathcal{S}_G^{vb,t} = (\widehat{IB}, \llbracket \rrbracket_{vb}^t, b_e, bw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{vb,t}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{vb,t}$).

Proof. Immediately with Lemma 4.2.1.3, Coincidence Theorem 3.5.2, and Termination Theorem 4.2.1.5.

Chapter 4.2.2

Very Busyness for a Set of Terms

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

343/185

Scenario 2: The Setting

...very busy for a set of terms T , T finite.

Lattice

► $\widehat{\mathcal{P}(T)} =_{df} (\mathcal{P}(T), \cap, \cup, \subseteq, \emptyset, T)$

...power set lattice over T : least element \emptyset , greatest element T , subset relation \subseteq as ordering relation, set intersection \cap and set union \cup as meet and join operation, respectively.

Scenario 2: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathcal{P}(T), \cap, \cup, \subseteq, \emptyset, T) = \widehat{\mathcal{P}(T)}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{vb}^T : E \rightarrow (\mathcal{P}(T) \rightarrow \mathcal{P}(T)) \text{ where}$$

$$\forall e \in E \forall T' \in \mathcal{P}(T). \llbracket e \rrbracket_{vb}^T(T') =_{df} \{t \in T \mid (t \in T' \wedge \text{Transp}_e^t) \vee \text{Comp}_e^t\}$$

- ▶ Initial information: $T_e \in \mathcal{P}(T)$
- ▶ Direction of information flow: backward

Very Busyness Specification for T

- ▶ Specification: $\mathcal{S}_G^{vb, T} = (\widehat{\mathcal{P}(T)}, \llbracket \cdot \rrbracket_{vb}^T, T_e, bw)$

Towards Termination and Optimality

Lemma 4.2.2.1 (Descending Chain Condition)

$\widehat{\mathcal{P}(T)}$ satisfies the descending chain condition.

Lemma 4.2.2.2 (Distributivity)

$\llbracket \rrbracket_{vb}^T$ is distributive.

Corollary 4.2.2.3 (Monotonicity)

$\llbracket \rrbracket_{vb}^T$ is monotonic.

Termination and Optimality

Theorem 4.2.2.4 (Termination)

Applied to $\mathcal{S}_G^{vb,T} = (\widehat{\mathcal{P}(T)}, \llbracket \rrbracket_{vb}^T, T_e, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb,T}$.

Proof. Immediately with Lemma 4.2.2.1, Corollary 4.2.2.3, and Termination Theorem 3.4.4.

Theorem 4.2.2.5 (Optimality)

Applied to $\mathcal{S}_G^{vb,T} = (\widehat{\mathcal{P}(T)}, \llbracket \rrbracket_{vb}^T, T_e, bw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{vb,T}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{vb,T}$).

Proof. Immediately with Lemma 4.2.2.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.2.2.4.

Chapter 4.2.3

Very Busyness for a Set of Terms: Bitvector Implementation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

348/185

Scenario 3: The Setting (1)

...very busy for a set of terms T , T finite, $|T| = n$.

Lattice

► $\widehat{\mathbb{B}}^n =_{df} (\mathbb{B}^n, \wedge_{pw}, \vee_{pw}, <_{pw}, \overline{false}, \overline{true})$

... n -ary cross-product lattice over \mathbb{B} : least element $\overline{false} =_{df} (false, \dots, false) \in \mathbb{B}^n$, greatest element $\overline{true} =_{df} (true, \dots, true) \in \mathbb{B}^n$, ordering relation $<_{pw}$ as pointwise extension of $<$ from $\widehat{\mathbb{B}}$ to $\widehat{\mathbb{B}}^n$, \wedge_{pw} and \vee_{pw} as pointwise extensions of logical \wedge and logical \vee from $\widehat{\mathbb{B}}$ to $\widehat{\mathbb{B}}^n$ as meet and join operation, respectively.

Scenario 3: The Setting (2)

Auxiliary Functions

► $ix : T \rightarrow \{1, \dots, n\}, ix^{-1} : \{1, \dots, n\} \rightarrow T$

...bijective mappings which map every term $t \in T$ to a number in $\{1, \dots, n\}$ and vice versa.

The $ix(t)^{th}$ element of an element

$$\bar{b} = (b_1, \dots, b_{ix(t)}, \dots, b_n) \in \mathbb{IB}^n$$

is the very busyness information for t stored in \bar{b} .

► $\cdot \downarrow_i : \mathbb{IB}^n \rightarrow \{1, \dots, n\} \rightarrow \mathbb{IB}$

...projection function which yields the i^{th} element of an element $\bar{b} \in \mathbb{IB}^n$, i.e., $\forall i \in \{1, \dots, n\}. \bar{b} \downarrow_i =_{df} b_i$.

Scenario 3: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\mathbb{B}^n, \wedge_{pw}, \vee_{pw}, <_{pw}, \overline{false}, \overline{true}) = \widehat{\mathbb{B}}^n$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{vb,cps}^T : E \rightarrow (\mathbb{B}^n \rightarrow \mathbb{B}^n) \text{ where}$$

$$\forall e \in E \forall v \in \mathbb{B}^n. \llbracket e \rrbracket_{vb,cps}^T(\bar{b}) =_{df} \bar{b}'$$

$$\text{where } \forall i \in \{1, \dots, n\}. \bar{b}' \downarrow_i =_{df}$$

$$(\bar{b} \downarrow_i \wedge \text{Transp}_e^{ix^{-1}(i)}) \vee \text{Comp}_e^{ix^{-1}(i)}$$

- ▶ Initial information: $\bar{b}_e \in \mathbb{B}^n$
- ▶ Direction of information flow: backward

Very Busyness Specification for T , cross-product spec. (cps)

- ▶ Specification: $\mathcal{S}_G^{vb,T,cps} = (\widehat{\mathbb{B}}^n, \llbracket \cdot \rrbracket_{vb,cps}^T, \bar{b}_e, bw)$

Scenario 3: Bitvector Implementation (1)

Bitvector Implementation of $\mathcal{S}_G^{vb, T, cps}$

- ▶ $\widehat{\mathbb{B}}^n$ can efficiently be implemented in terms of bitvectors $\vec{bv} = [d_1, \dots, d_n]$, $d_i \in \{0, 1\}$, $1 \leq i \leq n$, of length n .
- ▶ Let \mathcal{BV}^n denote the set of all bitvectors of length n .
- ▶ Let $\vec{bv}[i] = d_i$ for all $\vec{bv} = [d_1, \dots, d_n] \in \mathcal{BV}^n$, $1 \leq i \leq n$.
- ▶ Let $\vec{0} =_{df} [0, \dots, 0] \in \mathcal{BV}^n$ and $\vec{1} =_{df} [1, \dots, 1] \in \mathcal{BV}^n$.
- ▶ Let $\min_{\mathcal{BV}}$ and $\max_{\mathcal{BV}}$ be the **bitwise minimum** ('logical \wedge ') and the **bitwise maximum function** ('logical \vee ') over bitvectors, i.e., $\forall \vec{bv}_1, \vec{bv}_2 \in \mathcal{BV}^n \forall i \in \{1, \dots, n\}$.
 - ▶ $(\vec{bv}_1 \min_{\mathcal{BV}} \vec{bv}_2)[i] =_{df} \min(\vec{bv}_1[i], \vec{bv}_2[i])$
 - ▶ $(\vec{bv}_1 \max_{\mathcal{BV}} \vec{bv}_2)[i] =_{df} \max(\vec{bv}_1[i], \vec{bv}_2[i])$

Scenario 3: Bitvector Implementation (2)

Auxiliary Functions

► $ix : T \rightarrow \{1, \dots, n\}, ix^{-1} : \{1, \dots, n\} \rightarrow T$

...bijective mappings which map every term $t \in T$ to a number in $\{1, \dots, n\}$ and vice versa.

The $ix(t)^{th}$ element of a bitvector

$$\vec{bv} = [d_1, \dots, d_{ix(t)}, \dots, d_n] \in \mathcal{BV}^n$$

is the very busyness information for t stored in \vec{bv} .

Scenario 3: Bitvector Implementation (3)

Extending Local Predicates to Bitvectors

$$\blacktriangleright \overset{\rightarrow}{Comp}_e^T \in \mathcal{BV}^n$$

$$\forall i \in \{1, \dots, n\}. \overset{\rightarrow}{Comp}_e^T [i] =_{df} \begin{cases} 1 & \text{if } Comp_e^{ix^{-1}(i)} \\ 0 & \text{otherwise} \end{cases}$$

$$\blacktriangleright \overset{\rightarrow}{Transp}_e^T \in \mathcal{BV}^n$$

$$\forall i \in \{1, \dots, n\}. \overset{\rightarrow}{Transp}_e^T [i] =_{df} \begin{cases} 1 & \text{if } Transp_e^{ix^{-1}(i)} \\ 0 & \text{otherwise} \end{cases}$$

Scenario 3: Bitvector Implementation (4)

Bitvector Implementation

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\mathcal{BV}^n, \min_{\mathcal{BV}}, \max_{\mathcal{BV}}, <_{\mathcal{BV}}, \vec{0}, \vec{1}) = \widehat{\mathcal{BV}}^n$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{vb, bvi}^T : E \rightarrow (\mathcal{BV}^n \rightarrow \mathcal{BV}^n) \text{ where}$$

$$\forall e \in E \forall \vec{bv} \in \mathcal{BV}^n. \llbracket e \rrbracket_{vb, bvi}^T(\vec{bv}) =_{df}$$

$$(\vec{bv} \min_{\mathcal{BV}} \xrightarrow{\quad} \text{Transp}_e^T) \max_{\mathcal{BV}} \xrightarrow{\quad} \text{Comp}_e^T$$

- ▶ Initial information: $\vec{bv}_e \in \mathcal{BV}^n$

- ▶ Direction of information flow: backward

Very Busyness Specification for T , bitvector impl. (bvi)

- ▶ Specification: $\mathcal{S}_G^{vb, T, bvi} = (\widehat{\mathcal{BV}}^n, \llbracket \cdot \rrbracket_{vb, bvi}^T, \vec{bv}_e, bw)$

Towards Termination and Optimality

Lemma 4.2.3.1 (Descending Chain Condition)

$\widehat{\mathcal{BV}}^n$ satisfies the descending chain condition.

Lemma 4.2.3.2 (Distributivity)

$\llbracket \mathbb{I}_{vb,bvi}^T \rrbracket$ is distributive.

Corollary 4.2.3.3 (Monotonicity)

$\llbracket \mathbb{I}_{vb,bvi}^T \rrbracket$ is monotonic.

Termination

Theorem 4.2.3.4 (Termination)

Applied to $\mathcal{S}_G^{vb, T, bvi} = (\widehat{\mathcal{BV}}^n, \llbracket \rrbracket_{vb, bvi}^T, \vec{bv}_e, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb, T, bvi}$.

Proof. Immediately with Lemma 4.2.3.1, Corollary 4.2.3.3, and Termination Theorem 3.4.4.

Optimality

Theorem 4.2.3.5 (Optimality)

Applied to $\mathcal{S}_G^{vb,T,bvi} = (\widehat{\mathcal{BV}}^n, [\mathbb{I}]_{vb,bvi}^T, \vec{b}_e, bw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{vb,T,bvi}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{vb,T,bvi}$).

Proof. Immediately with Lemma 4.2.3.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.2.3.4.

Note

- ▶ All results of Chapter 4.2.3 hold for $\mathcal{S}_G^{vb,T,cps} = (\widehat{\mathcal{IB}}^n, [\mathbb{I}]_{vb,cps}^T, \vec{b}_e, bw)$, too.
- ▶ Applied to $\mathcal{S}_G^{vb,T,bvi} = (\widehat{\mathcal{BV}}^n, [\mathbb{I}]_{vb,bvi}^T, \vec{b}_e, bw)$, Algorithm 3.4.3 takes advantage of the efficient bitvector operations of actual processors.

Chapter 4.2.4

Very Busysem for a Set of Terms: Gen/Kill Implementation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

359/185

Scenario 4: The Setting

...very busy for a set of terms T , T finite.

Defining Gen/Kill Predicates

- ▶ $Gen_e^T =_{df} \{t \in T \mid Comp_e^t\}$
- ▶ $Kill_e^T =_{df} \{t \in T \mid Mod_e^t\}$

Note

- ▶ $Kill_e^T = \{t \in T \mid \neg Transp_e^t\}$

Scenario 4: DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathcal{P}(T), \cap, \cup, \subseteq, \emptyset, T) = \widehat{\mathcal{P}(T)}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{vb, gk}^T : E \rightarrow (\mathcal{P}(T) \rightarrow \mathcal{P}(T)) \text{ where}$$

$$\forall e \in E \forall T' \in \mathcal{P}(T). \llbracket e \rrbracket_{vb, gk}^T(T') =_{df} (T' \setminus Kill_e^T) \cup Gen_e^T$$

- ▶ Initial information: $T_e \in \mathcal{P}(T)$
- ▶ Direction of information flow: backward

Very Busyness Specification for T

- ▶ Specification: $\mathcal{S}_G^{vb, T, gk} = (\widehat{\mathcal{P}(T)}, \llbracket \cdot \rrbracket_{vb, gk}^T, T_e, bw)$

Towards Termination and Optimality

Compare

- ▶ $\llbracket \cdot \rrbracket_{vb, gk}^T : E \rightarrow (\mathcal{P}(T) \rightarrow \mathcal{P}(T))$ where
 $\forall e \in E \forall T' \in \mathcal{P}(T). \llbracket e \rrbracket_{vb, gk}^T(T') =_{df} (T' \setminus Kill_e^T) \cup Gen_e^T$
- ▶ $\llbracket \cdot \rrbracket_{vb}^T : E \rightarrow (\mathcal{P}(T) \rightarrow \mathcal{P}(T))$ where
 $\forall e \in E \forall T' \in \mathcal{P}(T). \llbracket e \rrbracket_{vb}^T(T') =_{df} \{t \in T \mid (t \in T' \wedge Transp_e^t) \vee Comp_e^t\}$

Obviously

Lemma 4.2.4.1 (Equality)

$$\llbracket \cdot \rrbracket_{vb}^T = \llbracket \cdot \rrbracket_{vb, gk}^T$$

Termination and Optimality

Theorem 4.2.4.2 (Termination)

Applied to $\mathcal{S}_G^{vb, T, gk} = (\widehat{\mathcal{P}(T)}, \llbracket \rrbracket_{vb, gk}^T, T_e, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb, T, gk}$.

Proof. Immediately with Lemma 4.2.4.1, Corollary 4.2.2.3, and Termination Theorem 3.4.4.

Theorem 4.2.4.3 (Optimality)

Applied to $\mathcal{S}_G^{vb, T, gk} = (\widehat{\mathcal{P}(T)}, \llbracket \rrbracket_{vb, gk}^T, T_e, bw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{vb, T, gk}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{vb, T, gk}$).

Proof. Immediately with Lemma 4.2.4.1, Lemma 4.2.2.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.2.4.2.

Recalling the *MaxFP* Equation System

Equation System 3.4.1 (*MaxFP* Equation System)

$$\mathit{inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \bigcap \{ \mathbb{I}(m, n) \mathbb{I}(\mathit{inf}(m)) \mid m \in \mathit{pred}(n) \} & \text{otherwise} \end{cases}$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

364/185

The Backward *MaxFP* Equation System

Equation System 3.4.1^{bw} (Backward *MaxFP* EQS)

$$\mathit{inf}(n) = \begin{cases} c_e & \text{if } n = \mathbf{e} \\ \bigcap \{ \llbracket (n, m) \rrbracket (\mathit{inf}(m)) \mid m \in \mathit{succ}(n) \} & \text{otherwise} \end{cases}$$

Specializing Equation System 3.4.1^{bw} for very busyness yields:

EQS 4.2.4.4 (EQS 3.4.1^{bw} for Very Busyness)

$$\mathit{VeryBusy}(n) = \begin{cases} T_e & \text{if } n = \mathbf{e} \\ \bigcap \{ \llbracket (n, m) \rrbracket_{vb, gk}^T (\mathit{VeryBusy}(m)) \mid m \in \mathit{succ}(n) \} & \text{otherwise} \end{cases}$$

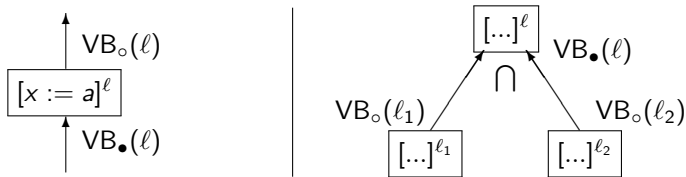
Specializing EQS 3.4.1^{bw} for Very Busyness

Expanding $\llbracket \cdot \rrbracket_{vb,gk}^T$ in Equation System 4.2.4.4 yields:

EQS 4.2.4.5 (EQS 3.4.1^{bw} for Very Busyness)

$$\text{VeryBusy}(n) = \begin{cases} T_e & \text{if } n = \mathbf{e} \\ \bigcap \{ (\text{VeryBusy}(m) \setminus \text{Kill}_{(n,m)}^T) \cup \text{Gen}_{(n,m)}^T \mid m \in \text{succ}(n) \} & \text{otherwise} \end{cases}$$

Recalling the VB Analysis of Chapter 2.3.2 (1)



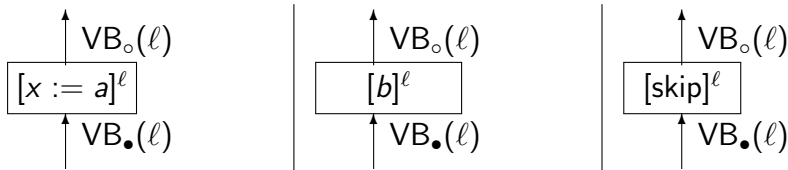
$$\begin{aligned}
 \text{VB}_o(\ell) &= (\text{VB}_\bullet(\ell) \setminus \text{kill}_{\text{VB}}(B^\ell)) \cup \text{gen}_{\text{VB}}(B^\ell) \quad \text{where } B^\ell \in \text{blocks}(S_\star) \\
 \text{VB}_\bullet(\ell) &= \begin{cases} \emptyset & : \text{ if } \ell = \text{final}(S_\star) \\ \bigcap \{ \text{VB}_o(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_\star) \} & : \text{ otherwise} \end{cases}
 \end{aligned}$$

Equation Syst. 4.2.4.5 (EQS 3.4.1^{bw} for Very Busy) – recalled

VeryBusy(*n*) =

$$\begin{cases} T_s & \text{if } n = \mathbf{e} \\ \bigcap \{ (\text{VeryBusy}(m) \setminus \text{Kill}_{(n,m)}^T) \cup \text{Gen}_{(n,m)}^T \mid m \in \text{succ}(n) \} & \text{otherwise} \end{cases}$$

Recalling the VB Analysis of Chapter 2.3.2 (2)



$$\text{gen}_{\text{VB}}([x := a]^l) = \text{AExp}(a)$$

$$\text{gen}_{\text{VB}}([b]^l) = \text{AExp}(b)$$

$$\text{gen}_{\text{VB}}([\text{skip}]^l) = \emptyset$$

$$\text{kill}_{\text{VB}}([x := a]^l) = \{a' \in \text{AExp}_* \mid x \in \text{FV}(a')\}$$

$$\text{kill}_{\text{VB}}([b]^l) = \emptyset$$

$$\text{kill}_{\text{VB}}([\text{skip}]^l) = \emptyset$$

Summing up

The **Very Busyness Equations** of

- ▶ Equation System 4.2.4.5
- ▶ Chapter 2.3.2

are **equivalent** up to the insignificant **formal difference** of considering

- ▶ edge-labelled
- ▶ node-labelled

flow graphs, respectively.

Chapter 4.2.5

Very Busyness Analysis: Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

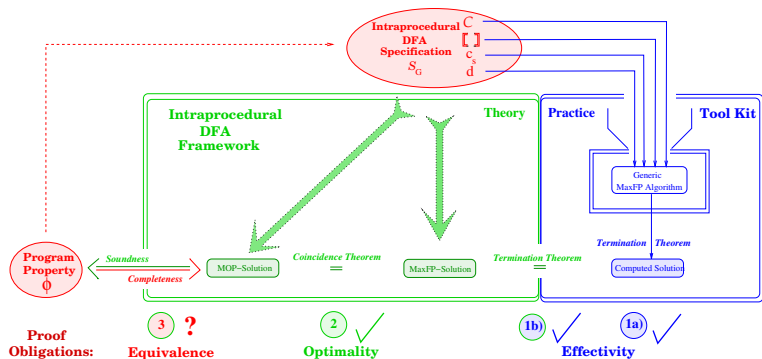
Chap. 9

Chap. 10

370/185

Closing the Final Proof Gap

...proving **soundness** and **completeness** of $\mathcal{S}_G^{vb,t}$ for the very **busyness** property:



Defining Very Busyness Informally

...intuitively:

- ▶ An expression is **very busy at a node** if, no matter what path is taken from that node to the exit of the program, the expression is computed before any of the variables occurring in it is redefined (cf. [Definition 2.3.2.1](#)).

Note

- ▶ If **exit of the program** is replaced by **exit of the procedure**, the informal 'definition' of very busyness does not foresee the possibility of the very busyness of an expression at the procedure exit itself.
- ▶ Situations where this very busyness is ensured by the calling context of the procedure, are thus not captured and can not be dealt with.

Useful Notation (recalled)

Let $G = (N, E, s, e)$ be a flow graph, and let *Predicate* be a predicate defined for edges $e \in E$.

We define for paths:

- ▶ Let $p = \langle e_1, \dots, e_q \rangle \in \mathbf{P}[m, n]$.
 - ▶ p_i , $1 \leq i \leq q$, denotes the i^{th} edge e_i of p .
 - ▶ $p_{[k, l]}$ denotes the subpath $\langle e_k, \dots, e_{l-1} \rangle$ of p .
 - ▶ λ_p denotes the length of p , i.e., the number q of edges of p .

We define for paths and predicates:

- ▶ $\text{Predicate}_p^{\forall} \iff \forall 1 \leq i \leq \lambda_p. \text{Predicate}_{p_i}$
- ▶ $\text{Predicate}_p^{\exists} \iff \exists 1 \leq i \leq \lambda_p. \text{Predicate}_{p_i}$

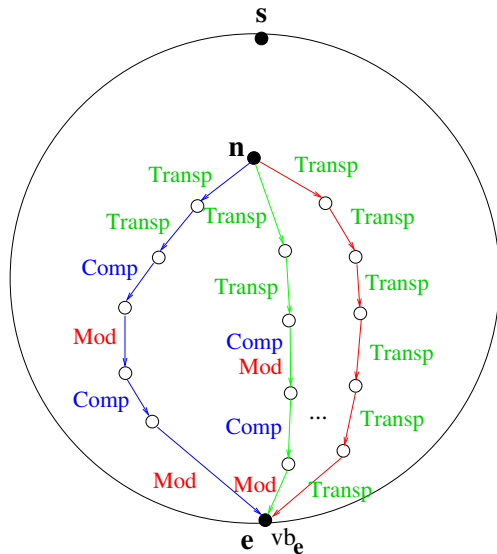
Defining Very Busyness Formally

Definition 4.2.5.1 (Very Busyness)

Let $G = (N, E, s, e)$ be a flow graph, let t be an expression, let $vb_e \in \mathbb{B}$ be the very busyness information for t at e ensured by the calling context of G .

$$\text{VeryBusy}^t(n) \iff_{df} \begin{cases} vb_e & \text{if } n = e \\ \forall p \in \mathbf{P}[n, e]. (vb_e^t \wedge \text{Transp}_p^{t\forall}) \vee \\ \quad \exists i \leq \lambda_p. \text{Comp}_{p_i}^t \wedge \text{Transp}_{p[1,i[}^{t\forall} & \text{otherwise} \end{cases}$$

Illustrating the Essence of Definition 4.2.5.1



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

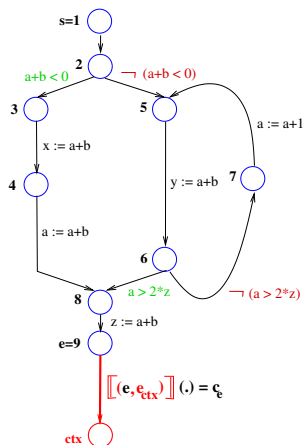
Chap. 9

Chap. 10

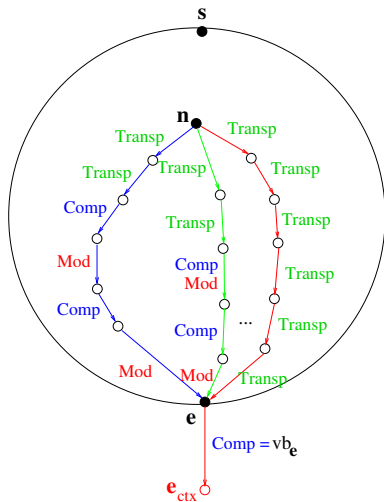
375/185

Context Edges

...introducing a **context edge** leads to a simpler uniform definition of **very busy**:



Context Edges and Very Busyness



...leading to:

$$\forall n \in N \setminus \{e_{ctx}\}. \text{VeryBusy}^t(n) \iff_{df} \forall p \in \mathbf{P}[n, e_{ctx}]. \exists i \leq \lambda_p. \text{Comp}_{p_i}^t \wedge \text{Transp}_{p[1,i]}^{t\forall}$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.2.4

4.2.5

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

377/185

Closing the Final Proof Gap

Theorem 4.2.5.2 (Soundness and Completeness)

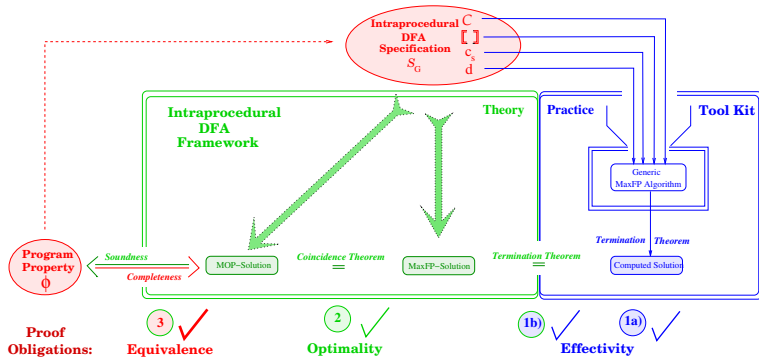
Let $G = (N, E, s, e)$ be a flow graph, t an expression, $vb_e \in \text{IB}$ the very busyness information for t at e ensured by the calling context of G , and let $MOP_{S_G^{vb,t}}$ be the MOP solution of the DFA specification $S_G^{vb,t} = (\widehat{\text{IB}}, \llbracket \cdot \rrbracket_{vb}^t, vb_e, bw)$.

Then:

$$\forall n \in N. \text{VeryBusy}^t(n) \iff MOP_{S_G^{vb,t}}(n)$$

Gap Closed: Soundness, Completeness Proven

...soundness and completeness of $\mathcal{S}_G^{vb,t}$ for very busy pro-
ven:



Chapter 4.3

Summary, Looking Ahead

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

380/185

Summary, Looking Ahead (1)

The terms

- ▶ Gen/Kill Analyses
- ▶ Bitvector Analyses

are used synonymously.

Gen/Kill Analyses are

- ▶ efficient
- ▶ scale well to more complex DFA scenarios (interprocedural, parallel, etc.)
- ▶ most important in practice.

Summary, Looking Ahead (2)

In fact, despite their conceptual simplicity, information obtained by **Gen/Kill Analyses** is fundamental for

- ▶ numerous powerful, widely used optimizations, e.g.,
 - ▶ **Partial Redundancy Elimination** (Busy Code Motion, Lazy Code Motion, cf. [Chapter 7](#) and [Chapter 8](#))
 - ▶ **Strength Reduction** (Lazy Strength Reduction, cf. [Appendix D](#))
 - ▶ **Partial Dead-Code Elimination** (cf. LVA 185.276 Analysis and Verification)
 - ▶ **Assignment Motion** (cf. LVA 185.276 Analysis and Verification)
 - ▶ ...

Chapter 4.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10




Part III

Chap. 11




Chap. 12

383/185



Further Reading for Chapter 4 (1)

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 1, Introduction; Chapter 9.2, Introduction to Data-Flow Analysis; Chapter 9.3, Foundations of Data-Flow Analysis)
-  Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 4.4, Data Flow Analysis)
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 10.2, A Taxonomy for Transformations — Machine-Independent Transformations, Machine-Dependent Transformations)

Further Reading for Chapter 4 (2)

-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 2, Classical Bit Vector Data Flow Analysis)
-  Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 4, Flow Graph)
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 8.3, Taxonomy of Data-Flow Problems and Solution Methods)

Further Reading for Chapter 4 (3)

-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)
-  Barry K. Rosen. *High-level Data Flow Analysis*. Communications of the ACM 20(10):141-156, 1977.

Chapter 5

Constant Propagation and Folding

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

387/185

Chapter 5.1

Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

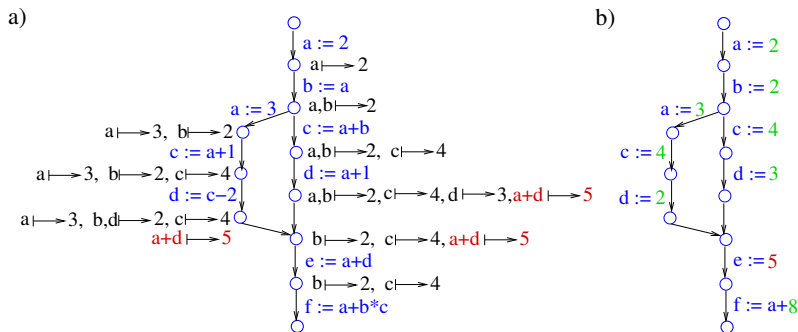
Chap. 7

Chap. 8

388/185

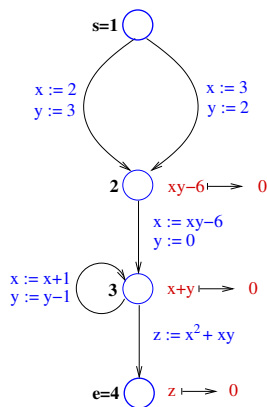
Motivating Example

...constant propagation and folding (for short: constant propagation (CP)) aims at discovering and replacing occurrences of terms whose evaluation will always yield the same value at runtime by this value.



Illustrating the Challenges of CP

...show that the terms $xy-6$, $x+y$, and z are constants of value 0 at the nodes 2, 3, and 4, respectively, .



Markus Müller-Olm, Helmut Seidl (SAS 2002)

Undecidability of Constant Propagation

Theorem 5.1.1 (Undecidability, Reif&Lewis 1977)

In the arithmetic domain, the problem of discovering all text expressions covered by constant signs is undecidable.

John H. Reif, Harry R. Lewis. [Symbolic Evaluation and the Global Value Graph](#). In Conference Record of the 4th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), 104-118, 1977.

Proof Sketch of Theorem 5.1.1 (1)

The proof proceeds by **reducing Hilbert's 10th problem** to the problem of discovering all text expressions covered by constant signs:

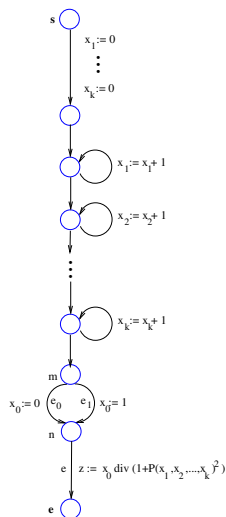
► Hilbert's 10th Problem

Let $\{x_1, \dots, x_k\}$ be a set of variables, $k > 5$, and let $P(x_1, \dots, x_k)$ be a (multivariate) polynomial.

It is not decidable, if $P(x_1, \dots, x_k)$ has a root in the **natural numbers** (Matijasevič 1970).

Proof Sketch of Theorem 5.1.1 (2)

Consider the program G below:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

393/185

Proof Sketch of Theorem 5.1.1 (3)

Prove the equivalence:

P has no root in the natural numbers iff
 z is constant (at node e of G)

which completes the proof. \square

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

394/185

The Bad News of Theorem 5.1.1

There is **no hope** of developing **an algorithm** that, when applied to an arbitrary program G ,

- ▶ determines for every term occurrence in G in a finite number of steps, if the evaluation of this occurrence will always yield the same value when its site is reached and its value computed at the runtime of G .

The Good News of Theorem 5.1.1 (1)

The **impossibility** of solving the **constant propagation (CP)** problem once and for all

- ▶ gives room for both **theoreticians** and **practitioners** to strive for constant propagation algorithms **tailored** and **optimized** for different purposes and goals.

The Good News of Theorem 5.1.1 (2)

The undecidability of the general CP problem inspires...

For the [theoretician](#)

- ▶ a quest for discovering settings with a [decidable](#) CP problem
 - ▶ [Restricting the class of admissible programs](#)
 - ▶ [Finite constants](#): arbitrary term operators, decidable for arbitrary control flow, complete for acyclic control flow, EXPTIME algorithm ([Steffen&Knoop 1989](#))
 - ▶ [Restricting the set of admissible expression operators](#)
 - ▶ [Presburger constants](#): $+$, $-$ as term operators, decidable and complete for programs with arbitrary control flow, polynomial time algorithm ([Müller-Olm&Rüthing 2001](#))
 - ▶ [Polynomial constants](#): $+$, $-$, $*$ as term operators, decidable and complete for programs with arbitrary control flow, time complexity of the proposed decision algorithm not yet known, PSPACE-hardness as a lower complexity bound ([Müller-Olm&Seidl 2002](#))

The Good News of Theorem 5.1.1 (3)

For the **practitioner**

- ▶ a quest for discovering **sweet spot settings** with a **useful and efficiently, scalable** decidable CP problem
 - ▶ **Simple constants**, intraprocedural
 - ▶ based on **expression pools** (Kildall 1973)
 - ▶ based on **definition-use chains**
 - ▶ based on **abstract state transformers**
 - ▶ based on the **global value graph** (Reif&Lewis 1977)
 - ▶ based on the **SSA value graph** (Knoop&Rüthing 2000)
 - ▶ **Q constants**, intraprocedural (Kam&Ullman 1977)
 - ▶ **Conditional constants**, intraprocedural (Wegman&Zadeck 1985)
 - ▶ **Linear constants**, interprocedural (Sagiv, Reps, Horwitz 1996)
 - ▶ **Copy constants**, interprocedural
 - ▶ **Strong constants**, parallel (Knoop 1998)
 - ▶ ...

The Good News of Theorem 5.1.1 (4)

In essence

Theoreticians cope with the **undecidability** of **CP** by

- ▶ trading **generality as little as possible** for decidability (neglecting efficiency and scalability).

Practitioners cope with the **undecidability** of **CP** by

- ▶ trading **generality as much as necessary** for efficient, scalable and useful decidability.

Note

- ▶ This is quite a typical situation in program analysis and optimization, and a virtually unexhaustable source of challenging and important research questions.

In the following

...we will focus on some of the approaches for constant propagation inspired by **practical demands** and **needs**, i.e., approaches offering a **good cost/benefit ratio**:

- ▶ Simple constants
- ▶ Linear constants
- ▶ Copy constants
- ▶ Q constants
- ▶ Conditional constants

Additionally, we consider

- ▶ Finite constants

as an example of an (in a well-defined formal sense) **optimal class of constants**.

Chapter 5.2

Preliminaries

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Workplan

Introducing and defining

- ▶ **Syntax of terms** (variables, operator and constant symbols,...)
- ▶ **Semantics of terms** (data domain, interpretation of operator and constant symbols, states,...)
- ▶ **Semantics of instructions** (state transformers,...)
- ▶ **Semantics of programs** (collecting semantics)
- ▶ **Constant propagation problem**
 - ▶ **Non-deterministic constant propagation** (Chap. 5.3)
 - ▶ **Deterministic constant propagation** (Chap. 5.9.3)

formally.

Towards the Syntax of Terms

...variables, constants, operators.

Let

- ▶ **V** be a set of *variables*,
- ▶ **C** be a set of *constant symbols (constants)*,
- ▶ **O** be a set of *k*-ary *operator symbols (or operators)*,
 $k \geq 1$.

Let

- ▶ **V**, **C**, and **O** be disjoint.

Syntax of Terms

Definition 5.2.1 (Terms)

1. Every variable $v \in \mathbf{V}$, every constant $c \in \mathbf{C}$ is a **term**.
2. If $op \in \mathbf{O}$ is a k -ary operator and t_1, \dots, t_k are terms, then (op, t_1, \dots, t_k) is a **term**.
3. There are no **terms** other than those which can be constructed by means of the above two rules.

The **set of all terms** is denoted by \mathbf{T} .

Towards the Semantics of Terms

We require:

- ▶ a **data domain ID**,
- ▶ an **interpretation** of constant and operator symbols over **ID**,
- ▶ a set of **states** over **ID**.

Data Domain

Let

- ▶ ID be a **data domain** of interest.

(E.g., the set of natural numbers \mathbb{N} , the set of integers \mathbb{Z} , the set of Boolean truth values \mathbb{B} , etc.).

The

- ▶ elements of ID are called **(data) values**.

We assume

- ▶ ID includes a **distinguished element** \perp representing the value *undefined*.

Interpreting Constant and Operator Symbols

Definition 5.2.2 (Interpretation)

An **interpretation** $I =_{df} (ID, I_0)$ of \mathbf{C} and \mathbf{O} is a pair, where

- ▶ ID is a data domain,
- ▶ I_0 is a function, which maps every
 - ▶ constant symbol $c \in \mathbf{C}$ to a datum $I_0(c) \in ID$,
 - ▶ k -ary operator symbol $op \in \mathbf{O}$ to a total strict function $I_0(op) : ID^k \rightarrow ID$, i.e., $I_0(op)(d_1, \dots, d_k) = \perp$, if there is a $j \in \{1, \dots, k\}$ with $d_j = \perp$.

States

Definition 5.2.3 (States over ID)

A **state** $\sigma : \mathbf{V} \rightarrow \text{ID}$ is a total mapping, which maps every variable to a data value $d \in \text{ID}$.

We denote the **set of all states** by

$$\Sigma =_{df} \{ \sigma \mid \sigma : \mathbf{V} \rightarrow \text{ID} \}$$

Semantics of Terms

Definition 5.2.4 (Semantics of Terms)

The **semantics** of terms $t \in \mathbf{T}$ is defined by the **evaluation** function

$$\mathcal{E} : \mathbf{T} \rightarrow (\Sigma \rightarrow \text{ID})$$

defined by

$$\forall t \in \mathbf{T} \forall \sigma \in \Sigma. \mathcal{E}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ l_0(c) & \text{if } t \equiv c \in \mathbf{C} \\ l_0(op)(\mathcal{E}(t_1)(\sigma), \dots, \mathcal{E}(t_k)(\sigma)) & \text{if } t \equiv (op, t_1, \dots, t_k) \end{cases}$$

Semantics of Instructions

Definition 5.2.5 (Semantics of Instructions)

- ▶ Let $\iota \equiv x := t$ be an assignment instruction. The **semantics** of ι is defined by the **state transformation function** (or **state transformer**) $\theta_\iota : \Sigma \rightarrow \Sigma$ defined by

$$\forall \sigma \in \Sigma \forall y \in \mathbf{V}. \theta_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- ▶ Let $\iota \equiv skip$ be the empty instruction. The **semantics** of ι is defined by the **identical state transformation function** (or **state transformer**) Id_Σ , i.e., $\theta_\iota =_{df} Id_\Sigma$, where $Id_\Sigma : \Sigma \rightarrow \Sigma$ is defined by $Id_\Sigma =_{df} \lambda \sigma. \sigma$.

Extending State Transformers to Paths

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let ι_e denote the instruction at edge e , $e \in E$.

Definition 5.2.6 (Extending θ from Edges to Paths)

The state transformers θ_{ι_e} , $e \in E$, are **extended** onto paths $p = \langle e_1, e_2, \dots, e_q \rangle$ in G by defining:

$$\theta_p =_{df} \begin{cases} Id_{\Sigma} & \text{if } q < 1 \\ \theta_{\langle e_2, \dots, e_q \rangle} \circ \theta_{\iota_{e_1}} & \text{otherwise} \end{cases}$$

Semantics of Programs: Collecting Semantics

Definition 5.2.7 (Collecting Semantics)

- ▶ The **collecting semantics** of G is defined by:

$$\mathcal{CS}_G : \Sigma \rightarrow N \rightarrow \mathcal{P}(\Sigma)$$

$$\forall \sigma \in \Sigma. \forall n \in N. \mathcal{CS}_G(\sigma, n) =_{df} \{ \theta_p(\sigma) \mid p \in \mathbf{P}[\mathbf{s}, n] \}$$

- ▶ The **collecting semantics** of G with respect to a fixed initial state $\sigma_s \in \Sigma$ is defined by:

$$\mathcal{CS}_{G, \sigma_s} : N \rightarrow \mathcal{P}(\Sigma)$$

$$\forall n \in N. \mathcal{CS}_{G, \sigma_s}(n) =_{df} \{ \theta_p(\sigma_s) \mid p \in \mathbf{P}[\mathbf{s}, n] \}$$

Chapter 5.3

Non-deterministic Constant Propagation: Problem Definition

Chapter 5.3.1

The Non-Deterministic CP Problem: Set View

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

Non-deterministic Constants

Let $\sigma_s \in \Sigma$ be an (initial) state, let $t \in \mathbf{T}$ be a term, and let $d \in \text{ID} \setminus \{\perp\}$ be a data value.

Definition 5.3.1.1 (Non-deterministic Constant)

t is a (non-deterministic) constant at node n for σ_s , i.e., the value of t at node n , $n \in N$, is a constant, if

$$\forall \sigma, \sigma' \in \mathcal{CS}_{G, \sigma_s}(n). \mathcal{E}(t)(\sigma) = \mathcal{E}(t)(\sigma') \neq \perp$$

Definition 5.3.1.2 (Non-det. Constant of Value d)

t is a (non-deterministic) constant of value d for σ_s at node n , $n \in N$, if

$$\{\mathcal{E}(t)(\sigma) \mid \sigma \in \mathcal{CS}_{G, \sigma_s}(n)\} = \{d\}$$

Deterministic Constants: Set View

Let $\sigma_s \in \Sigma$ be a state, and let n be a node of G .

Definition 5.3.1.3 (Constant Terms and Variables)

The set of **terms** and **variables** being (non-deterministic) **constants** of some value at n are given by the sets:

- ▶ $C_{G,\sigma_s}^T(n) =_{df} \{(t, d) \in \mathbf{T} \times \text{ID} \mid t \text{ is a constant of value } d \text{ for } \sigma_s \text{ at } n\}$
- ▶ $C_{G,\sigma_s}^V(n) =_{df} \{(v, d) \in \mathbf{V} \times \text{ID} \mid v \text{ is a constant of value } d \text{ for } \sigma_s \text{ at } n\}$

The Non-Det. Constant Propagation Problem

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let $\sigma_s \in \Sigma$ be a state.

The non-deterministic constant propagation (CP) problems for terms and variables are defined by:

Definition 5.3.1.4 (CP Problem for Terms, C^T)

The (non-deterministic) term CP problem C^T is to determine for every node $n \in N$ of G the set $C_{G, \sigma_s}^T(n)$.

Definition 5.3.1.5 (CP Problem for Variables, C^V)

The (non-deterministic) variable CP problem C^V is to determine for every node $n \in N$ of G the set $C_{G, \sigma_s}^V(n)$.

Chapter 5.3.2

The Non-Deterministic CP Problem: Valuation View

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

418/185

Non-deterministic Constants: Valuation View

The sets C_{G,σ_s}^T and C_{G,σ_s}^V induce for every node n two valuations (formally looking like states) $v_{C_{G,\sigma_s}^T}(n)$ and $v_{C_{G,\sigma_s}^V}(n)$, which are alternative representations of the sets $C_{G,\sigma_s}^T(n)$ and $C_{G,\sigma_s}^V(n)$, respectively (as justified by [Lemma 5.3.3.1](#)):

► $v_{C_{G,\sigma_s}^T} : N \rightarrow \mathbf{T} \rightarrow \text{ID}$ defined by

$$v_{C_{G,\sigma_s}^T}(n)(t) =_{df} \begin{cases} d & \text{if } (t, d) \in C_{G,\sigma_s}^T(n) \\ \perp & \text{otherwise} \end{cases}$$

► $v_{C_{G,\sigma_s}^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$ defined by

$$v_{C_{G,\sigma_s}^V}(n)(v) =_{df} \begin{cases} d & \text{if } (v, d) \in C_{G,\sigma_s}^V(n) \\ \perp & \text{otherwise} \end{cases}$$

Chapter 5.3.3

Equivalence of Set and Valuation View

Equivalence of Set and Valuation View

We have:

Lemma 5.3.3.1 (Equivalence)

$\forall n \in N \forall t \in \mathbf{T} \forall v \in \mathbf{V} \forall d \in \text{ID} \setminus \{\perp\}.$

1. $(t, d) \in C_{G, \sigma_s}^T(n)$ iff $v_{C_{G, \sigma_s}^T}(n)(t) = d$
2. $(v, d) \in C_{G, \sigma_s}^V(n)$ iff $v_{C_{G, \sigma_s}^V}(n)(v) = d$

Equivalent Characterization of the CP Problem

The Equivalence Lemma 5.3.3.1 implies:

Lemma 5.3.3.2 (Problem Equivalence)

Solving the non-deterministic CP problems

- ▶ C^T (i.e., non-det. CP for **terms**)
- ▶ C^V (i.e., non-det. CP for **variables**)

is equivalent to computing the higher-order (valuation) functions

- ▶ $v_{C_{G,\sigma_S}^T} : N \rightarrow \mathbf{T} \rightarrow \text{ID}$
- ▶ $v_{C_{G,\sigma_S}^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$

respectively.

Chapter 5.3.4

The C^T Approximation induced by C^V

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

423/185

The C^T Approximation induced by C^V

The solution C_{G,σ_s}^V of the CP problem C^V wrt the initial state σ_s , and its induced higher-order valuation (or state) function

$$v_{C_{G,\sigma_s}^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$$

induce a solution of the CP problem C^T for both the valuation and set view of C^T , respectively:

► Valuation-based:

$$v_{C_{G,\sigma_s}^T} : N \rightarrow \mathbf{T} \rightarrow \text{ID}$$

$$v_{C_{G,\sigma_s}^T}(n)(t) =_{df} \mathcal{E}(t)(v_{C_{G,\sigma_s}^V}(n))$$

► Set-based:

$$C_{C_{G,\sigma_s}^T}(n) =_{df} \{(t, d) \in \mathbf{T} \times \text{ID} \mid \mathcal{E}(t)(v_{C_{G,\sigma_s}^V}(n)) = d \neq \perp\}$$

$$\stackrel{L. 5.3.4.1}{=} \{(t, d) \in \mathbf{T} \times \text{ID} \mid v_{C_{G,\sigma_s}^T}(n)(t) = d \neq \perp\}$$

Equivalence and Approximation

We have:

Lemma 5.3.4.1 (Equivalence)

$$\forall n \in N. \forall \sigma_s \in \Sigma. \forall t \in \mathbf{T}. \forall d \in \text{ID}. \\ (t, d) \in C_{C_{G, \sigma_s}^V}^T(n) \iff v_{C_{G, \sigma_s}^V}^T(n)(t) = d$$

Lemma 5.3.4.2 (Approximation)

$$\forall n \in N. \forall \sigma_s \in \Sigma. C_{G, \sigma_s}^T(n) \supseteq C_{C_{G, \sigma_s}^V}^T(n)$$

In general, the inclusion is proper.

Interpretation, Finding (1)

Intuitively

- ▶ The [Approximation Lemma 5.3.4.2](#) states that a term can be a constant at some node n without that all of its variables are constants at n .

E.g., the equality of $xy - yx$ and 0 can be concluded without knowing the values of x and y ; actually, they need not be constant at all. For a more complex case consider $x^2 + xy = 0$ in the example of Müller-Olm and Seidl in [Chapter 5.1](#).

Hence

- ▶ Any sound algorithm for the CP problem C^V is in general conservative and suboptimal for the CP problem C^T .
- ▶ This would even hold for a (hypothetical) optimal algorithm (cf. [Undecidability Theorem 5.1.1](#)) for C^V .

Interpretation, Finding (2)

As a **matter of fact**

- ▶ The **Undecidability Theorem 5.1.1** rules out the possibility and existence of **optimal** CP algorithms for C^T and C^V .

Hence

- ▶ The best we can hope for are **conservative** CP algorithms for C^T and C^V trading optimality for decidability (and efficiency, scalability).

Chapter 5.3.5

CP Algorithms: Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

428/185

CP Algorithms: Soundness and Completeness

Let A^T (A^V) be a CP algorithm for C^T (C^V); let $A_{G,\sigma_s}^T(n) \subseteq \mathbf{T} \times \text{ID}$ ($A_{G,\sigma_s}^V(n) \subseteq \mathbf{V} \times \text{ID}$) denote the sets of terms (variables) discovered by A^T (A^V) to be constant at node n .

Definition 5.3.5.1 (Soundness of CP Algorithms)

A^T (A^V) is **sound** for C^T (C^V) if

$$\forall n \in N. \forall \sigma_s \in \Sigma. C_{G,\sigma_s}^T(n) \supseteq A_{G,\sigma_s}^T(n) \quad (C_{G,\sigma_s}^V(n) \supseteq A_{G,\sigma_s}^V(n))$$

Definition 5.3.5.2 (Completeness of CP Algorithms)

A^T (A^V) is **complete** for C^T (C^V) if

$$\forall n \in N. \forall \sigma_s \in \Sigma. C_{G,\sigma_s}^T(n) \subseteq A_{G,\sigma_s}^T(n) \quad (C_{G,\sigma_s}^V(n) \subseteq A_{G,\sigma_s}^V(n))$$

Chapter 5.3.6

CP Algorithms: Conservativity and Optimality

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

430/185

CP Algorithms: Conservativity and Optimality

Definition 5.3.6.1 (Conservativity of CP Algorithms)

A CP algorithm A^T (A^V) is **conservative** for C^T (C^V), if it is sound for C^T (C^V), i.e.

$$\forall n \in N. \forall \sigma_s \in \Sigma. C_{G, \sigma_s}^T(n) \supseteq A_{G, \sigma_s}^T(n) \quad (C_{G, \sigma_s}^V(n) \supseteq A_{G, \sigma_s}^V(n))$$

Definition 5.3.6.2 (Optimality of CP Algorithms)

A CP algorithm A^T (A^V) is **optimal** for C^T (C^V), if it is sound and complete for C^T (C^V), i.e.

$$\forall n \in N. \forall \sigma_s \in \Sigma. C_{G, \sigma_s}^T(n) = A_{G, \sigma_s}^T(n) \quad (C_{G, \sigma_s}^V(n) = A_{G, \sigma_s}^V(n))$$

Chapter 5.3.7

Classifying CP Algorithms

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

432/185

Classifying CP Algorithms (1)

In practice, CP algorithms fall into two groups with two variants each: Algorithms A , which compute and store values for

- ▶ **variables** at a program node, hence computing a mapping:

$$v_{AV} : N \rightarrow \mathbf{V} \rightarrow D, \quad D \in \{\text{ID}, \mathcal{P}(\text{ID})\}$$

- ▶ **terms** at a program node, hence computing a mapping:

$$v_{AT} : N \rightarrow \mathbf{T} \rightarrow D, \quad D \in \{\text{ID}, \mathcal{P}(\text{ID})\}$$

as the result of their analysis, called **variable** and **term valuation function**, respectively.

Classifying CP Algorithms (2)

We call a CP algorithm A

- ▶ singleton-valued, if A stores at most one value per variable or term at a program node.
- ▶ multiple-valued, if A stores one or more values per variable or term at a program node.

A	Number of Stored Values per Program Point	
	Singleton	Multiple
A^V	$v_{A^V, ID} : N \rightarrow \mathbf{V} \rightarrow ID$	$v_{A^V, \mathcal{P}(ID)} : N \rightarrow \mathbf{V} \rightarrow \mathcal{P}(ID)$
A^T	$v_{A^T, ID} : N \rightarrow \mathbf{T} \rightarrow ID$	$v_{A^T, \mathcal{P}(ID)} : N \rightarrow \mathbf{T} \rightarrow \mathcal{P}(ID)$

Looking ahead

The algorithms for

- ▶ Simple constants (Chap. 5.4)
- ▶ Linear constants (Chap. 5.5)
- ▶ Copy constants (Chap. 5.6)
- ▶ Q constants (Chap. 5.7)
- ▶ conditional constants (Chap. 5.9)

are singleton-valued algorithms for C^V (where the latter one and only the latter one addresses the deterministic variant of C^V).

The algorithm for

- ▶ Finite constants (Chap. 5.8)

is a singleton-valued algorithm for C^T .

Chapter 5.3.8

The C^T , C^V Approximations of CP Algorithms

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

436/185

Chapter 5.3.8.1

Singleton-Valued CP Algorithms for C^V

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

The C^V , C^T Valuation Approximations

...of a singleton-valued CP algorithm A^V for C^V .

The valuation approximation of the CP problem C^V of A^V is given by the variable valuation function

$$\blacktriangleright v_{A^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$$

computed by A^V .

The valuation approximation of the CP problem C^T of A^V is given by the term valuation function

$$\blacktriangleright v_{A^V}^T : N \rightarrow \mathbf{T} \rightarrow \text{ID}$$

induced by v_{A^V} and defined by:

$$\forall n \in N \forall t \in \mathbf{T}. v_{A^V}^T(n)(t) =_{df} \begin{cases} d & \text{if } \mathcal{E}(t)(v_{A^V}(n)) = d \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

The C^V, C^T Set Approximations

...of a singleton-valued CP algorithm A^V for C^V .

The set approximations of the CP problems C^V and C^T of A^V are induced by the variable and term valuation functions

- ▶ $v_{A^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$ (computed)
- ▶ $v_{A^V}^T : N \rightarrow \mathbf{T} \rightarrow \text{ID}$ (induced)

of A^V and defined by:

- ▶ $C_{A^V}(n) =_{df} \{(v, d) \in \mathbf{V} \times \text{ID} \mid v_{A^V}(n)(v) = d \neq \perp\}$
- ▶ $C_{A^V}^T(n) =_{df} \{(t, d) \in \mathbf{T} \times \text{ID} \mid v_{A^V}^T(n)(t) = d \neq \perp\}$

Chapter 5.3.8.2

Singleton-Valued CP Algorithms for C^T

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.3.1

5.3.2

5.3.3

5.3.4

5.3.5

5.3.6

5.3.7

5.3.8

5.4

5.5

5.6

5.7

5.8

5.9

5.10

440/185

The C^V , C^T Valuation & Set Approximations

...of a singleton-valued CP algorithm A^T for C^T .

The valuation approximation of the CP problem C^T of A^T is given by the term valuation function

$$\blacktriangleright v_{A^T} : N \rightarrow \mathbf{T} \rightarrow \text{ID}$$

computed by A^T .

The valuation and set approximations of the CP problems C^V and C^T of A^T are induced by v_{A^T} and defined by:

$$\blacktriangleright v_{A^T}^V : N \rightarrow \mathbf{V} \rightarrow \text{ID} \text{ defined by } v_{A^T}^V(n, v) =_{df} v_{A^T}(n, v)$$

$$\blacktriangleright C_{A^T}^V(n) =_{df} \{(v, d) \in \mathbf{V} \times \text{ID} \mid v_{A^T}(n)(v) = d \neq \perp\}$$

$$\blacktriangleright C_{A^T}(n) =_{df} \{(t, d) \in \mathbf{T} \times \text{ID} \mid v_{A^T}(n)(t) = d \neq \perp\}$$

Chapter 5.4

Simple Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.4.1

5.4.2

5.4.3

5.4.4

5.4.5

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chapter 5.4.1

DFA States, DFA Lattice

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.4.1

5.4.2

5.4.3

5.4.4

5.4.5

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

From Data Domains to DFA Lattices (1)

...domain extension.

Let

- ▶ ID be the **data domain** of interest (e.g. the set of natural numbers \mathbb{N} , the set of integers \mathbb{Z} , the set of Boolean truth values \mathbb{B} , etc.) with a distinguished element \perp representing the value *undefined*.

We extend ID by adding

- ▶ a new element \top not in ID , i.e., $\top \notin ID$.

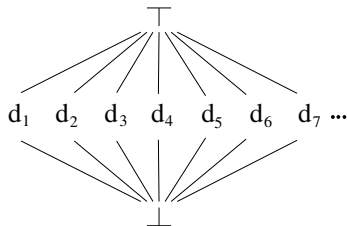
We denote the **extended domain** by

- ▶ $ID' =_{df} ID \cup \{\top\}$.

From Data Domains to DFA Lattices (2)

...lattice construction.

Given an extended data domain ID' , we construct the flat lattice $\mathcal{FL}_{ID'}$ (cf. Appendix A.4)



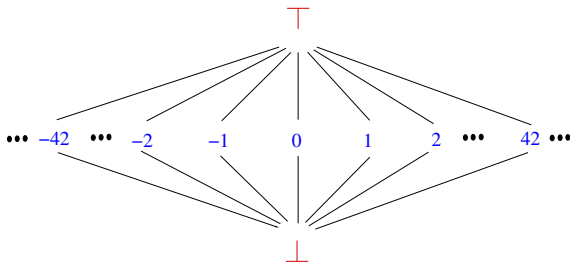
which constitutes the basic DFA lattice of the CP analysis.

Intuitively

- ▶ \top represents complete but inconsistent information.
- ▶ $d_i, i \geq 1$, represents precise information.
- ▶ \perp represents no information, the empty information.

The Basic DFA Lattice for CP over \mathbb{Z}

...is given by $\mathcal{FL}_{\mathbb{Z}}$:



...allowing to compute the class of simple constants over \mathbb{Z} .

Adapting the Notion of States: DFA States

Definition 5.4.1.1 (DFA States)

A **DFA state** $\sigma : \mathbf{V} \rightarrow ID'$ is a total mapping, which maps every variable to a datum $d \in ID'$.

The set of all DFA states is denoted by

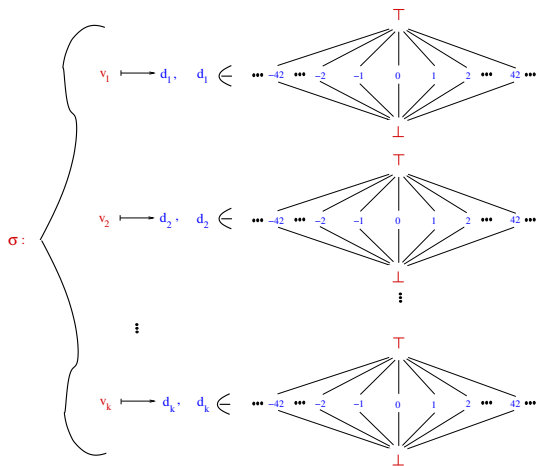
$$\Sigma' =_{df} \{ \sigma \mid \sigma : \mathbf{V} \rightarrow ID' \}$$

σ_{\perp} and σ_{\top} denote two distinguished states of Σ' defined by

$$\sigma_{\perp} = \lambda v. \perp, \quad \sigma_{\top} = \lambda v. \top$$

respectively.

Illustrating a DFA State σ over \mathbb{Z}



Initial DFA States

For an **initial DFA state**, we require that no variable is mapped to the special value \top , i.e, we require to either have precise information of the value of a variable, when entering a procedure, or no information at all. We define:

Definition 5.4.1.2 (Initial DFA States over ID')

The set of **initial DFA states** is defined by

$$\Sigma'_{init} =_{df} \{ \sigma \in \Sigma' \mid \forall v \in \mathbf{V}. \sigma(v) \neq \top \}$$

Note: The set of initial DFA states Σ'_{init} coincides with the set of (program) states Σ of **Definition 5.2.3**, i.e., $\Sigma'_{init} = \Sigma$.

Extending the Interpretation to $\top \in ID'$

Definition 5.4.1.3 (Extending the Interpretation)

Let $I =_{df} (ID, I_0)$ be an interpretation of constant and operator symbols over the data domain ID .

Then $I' =_{df} (ID', I'_0)$ extends I to an interpretation over ID' by defining

- ▶ $I'_0(c) =_{df} I_0(c)$ for every constant symbol $c \in \mathbf{C}$
- ▶ $I'_0(op) : ID'^n \rightarrow ID'$ for every operator symbol $op \in \mathbf{O}$ by

$$\forall (d_1, \dots, d_k) \in ID'^n. I'_0(op)(d_1, \dots, d_k) =_{df}$$

$$\begin{cases} I_0(op)(d_1, \dots, d_k) & \text{if } d_i = \perp \text{ for some } 1 \leq i \leq k, \text{ or} \\ & d_j \neq \top, 1 \leq j \leq k \\ \top & \text{if } d_i \neq \perp, 1 \leq i \leq k, \text{ and} \\ & d_j = \top \text{ for some } 1 \leq j \leq k \end{cases}$$

Adapting the Semantics of Terms to ID'

Definition 5.4.1.4 (Adapted Semantics of Terms)

The **semantics** of terms $t \in \mathbf{T}$ is defined by the **extended evaluation function**

$$\mathcal{E}' : \mathbf{T} \rightarrow (\Sigma' \rightarrow ID')$$

defined by

$$\forall t \in \mathbf{T} \forall \sigma \in \Sigma'. \mathcal{E}'(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ l'_0(c) & \text{if } t \equiv c \in \mathbf{C} \\ l'_0(op)(\mathcal{E}'(t_1)(\sigma), \dots, \mathcal{E}'(t_k)(\sigma)) & \text{if } t \equiv (op, t_1, \dots, t_k) \end{cases}$$

Adapting the Semantics of Instructions to Σ'

Definition 5.4.1.5 (Adapted Semantics of Instruc's)

- ▶ Let $\iota \equiv x := t$ be an assignment instruction. The **semantics** of ι is defined by the **extended state transformer** $\theta'_\iota : \Sigma' \rightarrow \Sigma'$ defined by

$$\forall \sigma \in \Sigma' \forall y \in \mathbf{V}. \theta'_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}'(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- ▶ Let $\iota \equiv \text{skip}$ be the empty instruction. The **semantics** of ι is defined by the **extended identical state transformer** $Id_{\Sigma'}$, i.e., $\theta'_\iota =_{df} Id_{\Sigma'}$, where $Id_{\Sigma'} : \Sigma' \rightarrow \Sigma'$ is defined by $Id_{\Sigma'} =_{df} \lambda \sigma. \sigma$.

The DFA Lattice for Simple Constants

The set of DFA states together with the pointwise ordering of states, $\sqsubseteq_{\Sigma'}$, constitutes a complete lattice (cf. Appendix A.4):

$$\forall \sigma, \sigma' \in \Sigma'. \sigma \sqsubseteq_{\Sigma'} \sigma' \text{ iff } \forall v \in \mathbf{V}. \sigma(v) \sqsubseteq_{\mathcal{FL}_{\mathbb{D}'}} \sigma'(v)$$

Lemma 5.3.1.6 (Lattice of DFA States)

$\widehat{\Sigma}' =_{df} (\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_{\perp}, \sigma_{\top})$ is a complete lattice with

- ▶ least element σ_{\perp} , greatest element σ_{\top} ,
- ▶ pointwise meet $\sqcap_{\Sigma'}$ and join $\sqcup_{\Sigma'}$ as meet and join operation, respectively.

Chapter 5.4.2

Simple Constants: Specification

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.4.1

5.4.2

5.4.3

5.4.4

5.4.5

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Simple Constants over \mathbb{Z} : DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_{\perp}, \sigma_{\top}) = \widehat{\Sigma}'$$

with Σ' set of DFA states over \mathbb{Z} .

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{sc} : E \rightarrow (\Sigma' \rightarrow \Sigma') \text{ where } \forall e \in E. \llbracket e \rrbracket_{sc} =_{df} \theta'_{l_e}$$

- ▶ Initial information: $\sigma_s \in \Sigma'_{init}$
- ▶ Direction of information flow: forward

Simple Constants Specification

- ▶ Specification: $\mathcal{S}_G^{sc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{sc}, \sigma_s, fw)$

Chapter 5.4.3

Termination, Safety, and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.4.1

5.4.2

5.4.3

5.4.4

5.4.5

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Towards Safety and Termination

Lemma 5.4.3.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.4.3.2 (Monotonicity)

$\llbracket \cdot \rrbracket_{sc}$ is monotonic.

Lemma 5.4.3.3 (Non-Distributivity)

$\llbracket \cdot \rrbracket_{sc}$ is not distributive.

Termination and Safety/Conservativity

Theorem 5.4.3.4 (Termination)

Applied to $\mathcal{S}_G^{sc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{sc}, \sigma_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of \mathcal{S}_G^{sc} .

Proof. Immediately with Lemma 5.4.3.1, Lemma 5.4.3.2, and Termination Theorem 3.4.4.

Theorem 5.4.3.5 (Safety/Conservativity)

Applied to $\mathcal{S}_G^{sc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{sc}, \sigma_s, fw)$, Algorithm 3.4.3 is *MOP* conservative for \mathcal{S}_G^{sc} (i.e., it terminates with a lower approximation of the *MOP* solution of \mathcal{S}_G^{sc}).

Proof. Immediately with Lemma 5.4.3.2, Safety Theorem 3.5.1, and Termination Theorem 5.4.3.4.

Non-Coincidence

Theorem 5.4.3.6 (Non-Coincidence/Non-Opt.)

Applied to $\mathcal{S}_G^{sc} = (\widehat{\Sigma}', \llbracket \rrbracket_{sc}, \sigma_s, fw)$, Algorithm 3.4.3 is in general not *MOP* optimal for \mathcal{S}_G^{sc} (i.e., it terminates with a properly lower approximation of the *MOP* solution of \mathcal{S}_G^{sc}).

Proof. Immediately with Lemma 5.4.3.3, Coincidence Theorem 3.5.2, and Termination Theorem 5.4.3.4.

Corollary 5.4.3.7 (Safety, Non-Coincidence)

The *MaxFP* solution for \mathcal{S}_G^{sc} , is always a safe approximation of the *MOP* solution of \mathcal{S}_G^{sc} . In general, the *MOP* solution and the *MaxFP* solution of \mathcal{S}_G^{sc} do not coincide.

Chapter 5.4.4

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.4.1

5.4.2

5.4.3

5.4.4

5.4.5

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Soundness and Completeness of $MOP_{S_G^{sc}}$

Theorem 5.4.4.1 (Soundness and Completeness)

The MOP solution of S_G^{sc} is

1. sound and complete for the variable constant propagation problem C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^V}(n) =_{\Sigma'} MOP_{S_G^{sc}, \sigma_s}(n)$$

2. sound but not complete for the term constant propagation problem C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^T}(n) \sqsubseteq_{\Sigma'} MOP_{S_G^{sc}, \sigma_s}^T(n)$$

In general, the inclusion is proper.

Soundness and Completeness of $MaxFP_{S_G^{sc}}$

Corollary 5.4.4.2 (Soundness and Completeness)

The $MaxFP$ solution of S_G^{sc} is

1. sound but not complete for C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C^V_{G, \sigma_s}}(n) \not\sqsupseteq_{\Sigma'} MaxFP_{S_G^{sc}, \sigma_s}(n)$$

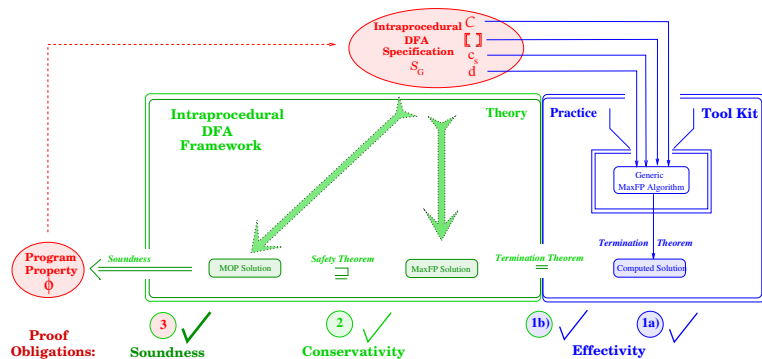
2. sound but not complete for C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C^T_{G, \sigma_s}}(n) \not\sqsupseteq_{\Sigma'} MaxFP^T_{S_G^{sc}, \sigma_s}(n)$$

In general, both inclusions are proper.

In Summary: Soundness, *MOP*-Conservativity

...of \mathcal{S}_G^{SC} for simple constants proven:



Chapter 5.4.5

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.4.1

5.4.2

5.4.3

5.4.4

5.4.5

5.5

5.6

5.7

5.8

5.9

5.10

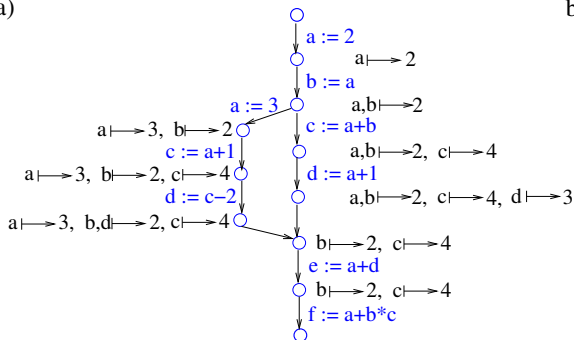
5.11

5.12

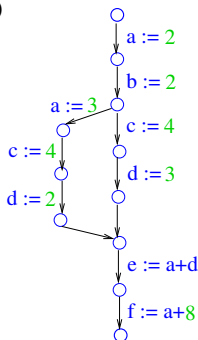
Chap. 6

Simple Constants over \mathbb{Z} : Illustrating Example

a)



b)



...all terms except of $a + d$ and $a + 8$ are simple constants.

Note: The term $a + d$ is a constant of value 5, though not a simple constant; the term $a + 8$ is not a (non-deterministic) constant.

Chapter 5.5

Linear Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.5.4

5.5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chapter 5.5.1

DFA States, DFA Lattice

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.5.4

5.5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

DFA States, DFA Lattice, and More

The **linear constants (LC)** analysis shares the

- ▶ extended data domain ID'
- ▶ basic flat DFA lattice $\mathcal{FL}_{ID'}$
- ▶ set of DFA states Σ'
- ▶ extended interpretation $I' =_{df} (ID', I'_0)$

with the **simple constants (SC)** analysis.

LC-specific are

- ▶ the adaption of the semantics of terms to ID'
- ▶ the adaption of the semantics of instructions to Σ'

LC-specific Term Semantics Adaption

Definition 5.5.1.1 (Adapted Semantics of Terms)

The **LC-specific semantics** of terms $t \in \mathbf{T}$ is defined by the extended evaluation function

$$\mathcal{E}_{lc}: \mathbf{T} \rightarrow (\Sigma' \rightarrow \text{ID}')$$

defined by

$$\forall t \in \mathbf{T} \forall \sigma \in \Sigma'. \mathcal{E}_{lc}(t)(\sigma) =_{df}$$

$$\left\{ \begin{array}{ll} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ l'_0(c) & \text{if } t \equiv c \in \mathbf{C} \\ l'_0(\oplus)(\mathcal{E}'(*, c, x)(\sigma), \mathcal{E}_{lc}(d)(\sigma)) & \\ & \text{if } t \equiv (\oplus, (*, c, x), d) \equiv c * x \oplus d, \\ & c, d \in \mathbf{C}, \oplus \in \{+, -\} \\ \perp & \text{otherwise} \end{array} \right.$$

LC-specific Instructions Semantics Adaption

Definition 5.5.1.2 (Adapted Semantics of Instruc's)

- ▶ Let $\iota \equiv x := t$ be an assignment instruction. The **semantics** of ι is defined by the LC-specific state transformer $\theta_\iota^{lc} : \Sigma' \rightarrow \Sigma'$ defined by

$$\forall \sigma \in \Sigma' \forall y \in \mathbf{V}. \theta_\iota^{lc}(\sigma)(y) =_{df} \begin{cases} \mathcal{E}_{lc}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- ▶ Let $\iota \equiv skip$ be the empty instruction. The **semantics** of ι is defined by the **extended identical state transformer** $Id_{\Sigma'}$, i.e., $\theta_\iota^{lc} =_{df} Id_{\Sigma'}$, where $Id_{\Sigma'} : \Sigma' \rightarrow \Sigma'$ is defined by $Id_{\Sigma'} =_{df} \lambda \sigma. \sigma$.

Chapter 5.5.2

Linear Constants: Specification

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.5.4

5.5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Linear Constants over \mathbb{Z} : DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_{\perp}, \sigma_{\top}) = \widehat{\Sigma}'$$

with Σ' set of DFA states over \mathbb{Z} .

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{lc} : E \rightarrow (\Sigma' \rightarrow \Sigma') \text{ where } \forall e \in E. \llbracket e \rrbracket_{lc} =_{df} \theta_{\nu_e}^{lc}$$

- ▶ Initial information: $\sigma_s \in \Sigma'_{init}$
- ▶ Direction of information flow: forward

Linear Constants Specification

- ▶ Specification: $\mathcal{S}_G^{lc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{lc}, \sigma_s, fw)$

Chapter 5.5.3

Termination, Safety, and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.5.4

5.5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Towards Coincidence and Termination

Lemma 5.5.3.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.5.3.2 (Distributivity)

$\llbracket \cdot \rrbracket_{lc}$ is distributive.

Corollary 5.5.3.3 (Monotonicity)

$\llbracket \cdot \rrbracket_{lc}$ is monotonic.

Termination and Coincidence/Optimality

Theorem 5.5.3.4 (Termination)

Applied to $\mathcal{S}_G^{lc} = (\widehat{\Sigma}', \llbracket \rrbracket_{lc}, \sigma_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of \mathcal{S}_G^{lc} .

Proof. Immediately with Lemma 5.5.3.1, Lemma 5.5.3.3, and Termination Theorem 3.4.4.

Theorem 5.5.3.5 (Coincidence/Optimality)

Applied to $\mathcal{S}_G^{lc} = (\widehat{\Sigma}', \llbracket \rrbracket_{lc}, \sigma_s, fw)$, Algorithm 3.4.3 is *MOP* optimal for \mathcal{S}_G^{lc} (i.e., it terminates with the *MOP* solution of \mathcal{S}_G^{lc}).

Proof. Immediately with Lemma 5.5.3.2, Coincidence Theorem 3.5.2, and Termination Theorem 5.5.3.4.

Chapter 5.5.4

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.5.4

5.5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Soundness, Non-Completeness of $MOP_{S_G^{lc}}$

Theorem 5.5.4.1 (Soundness, Non-Completeness)

The MOP solution of S_G^{lc} is

1. sound but not complete for the variable constant propagation problem C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^V}(n) \sqsupseteq_{\Sigma'} MOP_{S_G^{lc},\sigma_s}(n)$$

2. sound but not complete for the term constant propagation problem C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^T}(n) \sqsupseteq_{\Sigma'} MOP_{S_G^{lc},\sigma_s}^T(n)$$

In general, both inclusions are proper.

Soundness, Non-Completeness of $MaxFP_{S_G^{lc}}$

Corollary 5.5.4.2 (Soundness, Non-Completeness)

The $MaxFP$ solution of S_G^{lc} is

1. sound but not complete for C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_G^V, \sigma_s}(n) \not\sqsubseteq_{\Sigma'} MaxFP_{S_G^{lc}, \sigma_s}(n)$$

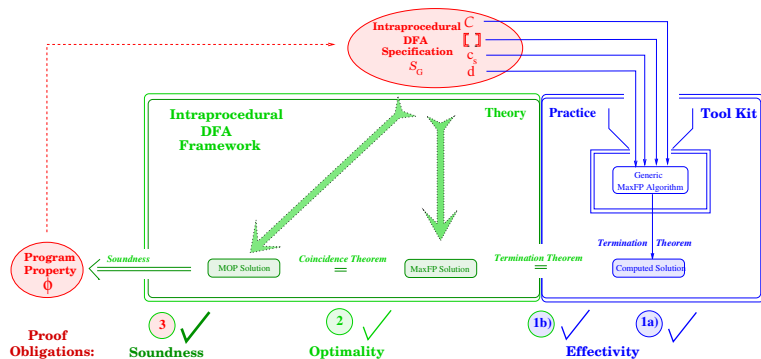
2. sound but not complete for C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_G^T, \sigma_s}(n) \not\sqsubseteq_{\Sigma'} MaxFP_{S_G^{lc}, \sigma_s}^T(n)$$

In general, both inclusions are proper.

In Summary: Soundness, *MOP*-Optimality

...of \mathcal{S}_G^{lc} for linear constants proven:



Chapter 5.5.5

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.5.4

5.5.5

5.6

5.7

5.8

5.9

5.10

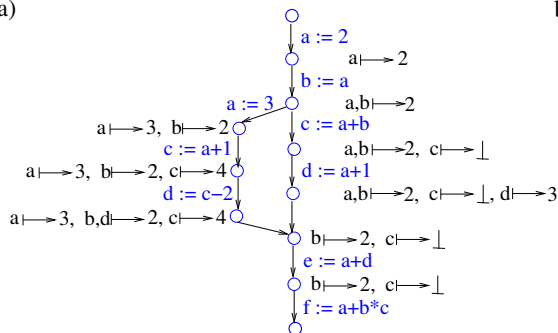
5.11

5.12

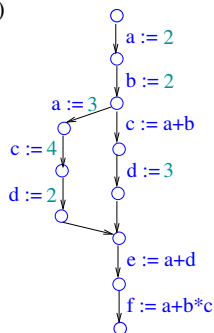
Chap. 6

Linear Constants over \mathbb{Z} : Illustrating Example

a)



b)



...the terms $a + b$, $a + d$, $b * c$, and $a + b * c$ are not linear constants, though they are simple constants (except of $a + 8 \equiv a + b * c$).

Chapter 5.6

Copy Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.6.1

5.6.2

5.6.3

5.6.4

5.6.5

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chapter 5.6.1

DFA States, DFA Lattice

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.6.1

5.6.2

5.6.3

5.6.4

5.6.5

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

DFA States, DFA Lattice, and More

The **copy constants (CpC)** analysis shares the

- ▶ extended data domain ID'
- ▶ basic flat DFA lattice $\mathcal{FL}_{ID'}$
- ▶ set of DFA states Σ'
- ▶ extended interpretation $I' =_{df} (ID', I'_0)$

with the **simple constants (SC)** and the **linear constants (LC)** analysis.

CpC-specific are

- ▶ the adaption of the semantics of terms to ID'
- ▶ the adaption of the semantics of instructions to Σ'

CpC-specific Term Semantics Adaption

Definition 5.6.1.1 (Adapted Semantics of Terms)

The CpC-specific semantics of terms $t \in \mathbf{T}$ is defined by the extended evaluation function

$$\mathcal{E}_{cpc}: \mathbf{T} \rightarrow (\Sigma' \rightarrow \text{ID}')$$

defined by

$$\forall t \in \mathbf{T} \forall \sigma \in \Sigma'. \mathcal{E}_{cpc}(t)(\sigma) =_{df}$$

$$\begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ l'_0(c) & \text{if } t \equiv c \in \mathbf{C} \\ \perp & \text{otherwise} \end{cases}$$

CpC-specific Instructions Semantics Adaption

Definition 5.6.1.2 (Adapted Semantics of Instruc's)

- ▶ Let $\iota \equiv x := t$ be an assignment instruction. The **semantics** of ι is defined by the CpC-specific state transformer $\theta_{\iota}^{cpc} : \Sigma' \rightarrow \Sigma'$ defined by

$$\forall \sigma \in \Sigma' \forall y \in \mathbf{V}. \theta_{\iota}^{cpc}(\sigma)(y) =_{df} \begin{cases} \mathcal{E}_{cpc}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- ▶ Let $\iota \equiv skip$ be the empty instruction. The **semantics** of ι is defined by the **extended identical state transformer** $Id_{\Sigma'}$, i.e., $\theta_{\iota}^{cpc} =_{df} Id_{\Sigma'}$, where $Id_{\Sigma'} : \Sigma' \rightarrow \Sigma'$ is defined by $Id_{\Sigma'} =_{df} \lambda \sigma. \sigma$.

Chapter 5.6.2

Copy Constants: Specification

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.6.1

5.6.2

5.6.3

5.6.4

5.6.5

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Copy Constants over \mathbb{Z} : DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_{\perp}, \sigma_{\top}) = \widehat{\Sigma}'$$

with Σ' set of DFA states over \mathbb{Z} .

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{cpc} : E \rightarrow (\Sigma' \rightarrow \Sigma') \text{ where } \forall e \in E. \llbracket e \rrbracket_{cpc} =_{df} \theta_{le}^{cpc}$$

- ▶ Initial information: $\sigma_s \in \Sigma'_{init}$

- ▶ Direction of information flow: forward

Copy Constants Specification

- ▶ Specification: $\mathcal{S}_G^{cpc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{cpc}, \sigma_s, fw)$

Chapter 5.6.3

Termination, Safety, and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.6.1

5.6.2

5.6.3

5.6.4

5.6.5

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Towards Coincidence and Termination

Lemma 5.6.3.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.6.3.2 (Distributivity)

$\llbracket \cdot \rrbracket_{cpc}$ is distributive.

Corollary 5.6.3.3 (Monotonicity)

$\llbracket \cdot \rrbracket_{cpc}$ is monotonic.

Termination and Coincidence/Optimality

Theorem 5.6.3.4 (Termination)

Applied to $\mathcal{S}_G^{cpc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{cpc}, \sigma_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of \mathcal{S}_G^{cpc} .

Proof. Immediately with Lemma 5.6.3.1, Lemma 5.6.3.3, and Termination Theorem 3.4.4.

Theorem 5.6.3.5 (Coincidence/Optimality)

Applied to $\mathcal{S}_G^{cpc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{cpc}, \sigma_s, fw)$, Algorithm 3.4.3 is *MOP* optimal for \mathcal{S}_G^{cpc} (i.e., it terminates with the *MOP* solution of \mathcal{S}_G^{cpc}).

Proof. Immediately with Lemma 5.6.3.2, Coincidence Theorem 3.5.2, and Termination Theorem 5.6.3.4.

Chapter 5.6.4

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.6.1

5.6.2

5.6.3

5.6.4

5.6.5

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Soundness, Non-Completeness of $MOP_{S_G^{cpc}}$

Theorem 5.6.4.1 (Soundness, Non-Completeness)

The MOP solution of S_G^{cpc} is

1. sound but not complete for the variable constant propagation problem C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{init}. v_{C_{G,\sigma_s}^V}(n) \sqsubseteq_{\Sigma'} MOP_{S_G^{cpc}, \sigma_s}(n)$$

2. sound but not complete for the term constant propagation problem C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{init}. v_{C_{G,\sigma_s}^T}(n) \sqsubseteq_{\Sigma'} MOP_{S_G^{cpc}, \sigma_s}^T(n)$$

In general, both inclusions are proper.

Soundness, Non-Completeness of $MaxFP_{S_G^{cpc}}$

Corollary 5.6.4.2 (Soundness, Non-Completeness)

The $MaxFP$ solution of S_G^{cpc} is

1. sound but not complete for C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_G^V}(n) \not\sqsupseteq_{\Sigma'} MaxFP_{S_G^{cpc}, \sigma_s}(n)$$

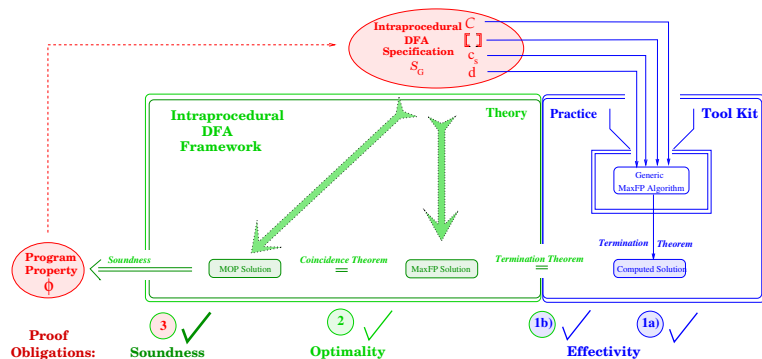
2. sound but not complete for C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_G^T}(n) \not\sqsupseteq_{\Sigma'} MaxFP_{S_G^{cpc}, \sigma_s}^T(n)$$

In general, both inclusions are proper.

In Summary: Soundness, *MOP*-Optimality

...of \mathcal{S}_G^{cpc} for copy constants proven:



Chapter 5.6.5

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.6.1

5.6.2

5.6.3

5.6.4

5.6.5

5.7

5.8

5.9

5.10

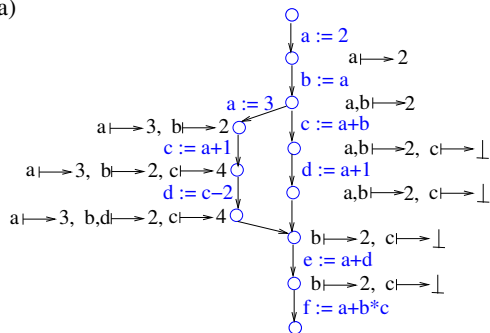
5.11

5.12

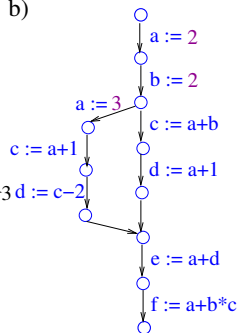
Chap. 6

Copy Constants over \mathbb{Z} : Illustrating Example

a)



b)



...only the right-hand side terms 2, 3, and a, are copy constants, though many of the other terms are linear or simple constants.

Chapter 5.7

Q Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

498/185

Chapter 5.7.1

Background and Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

Background and Motivation (1)

...the *MOP* solution as the specifying solution of a DFA problem is not decidable.

Theorem 3.3.3 (Undecidability, Kam&Ullman 1977)

– recalled

There is no algorithm A satisfying:

- ▶ The input of A are
 - ▶ a DFA specification $\mathcal{S}_G = (\hat{C}, [], c_s, fw)$
 - ▶ algorithms for the computation of the meet, the equality test, and the application of monotonic functions on the elements of a complete lattice
- ▶ The output of A is the *MOP* solution of \mathcal{S}_G .

Background and Motivation (2)

...for monotonic DFA problems, the *MaxFP* solution as their computable solution is generally a proper approximation of their *MOP* solution only.

Theorem 3.5.1 (Safety) – recalled

The *MaxFP* solution of $\mathcal{S}_G = (\hat{C}, \llbracket \cdot \rrbracket, c_s, fw)$ is a safe (i.e., lower) approximation of the *MOP* solution of \mathcal{S}_G , i.e.,

$$\forall n \in N. \text{MaxFP}_{\mathcal{S}_G}(n) \sqsubseteq \text{MOP}_{\mathcal{S}_G}(n)$$

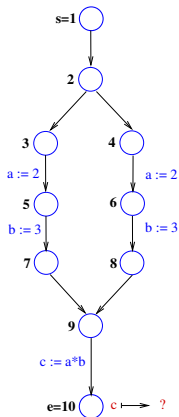
if the DFA functional $\llbracket \cdot \rrbracket$ is monotonic.

The Impact of Monotonicity on SCs (1)

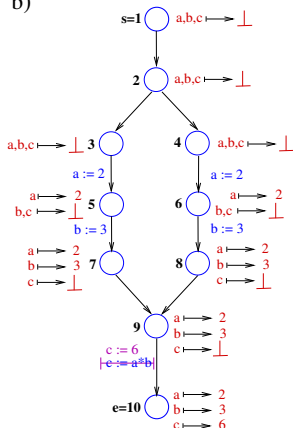
...SCs, a (non-distributive) monotonic DFA problem.

While $a * b$ is a simple constant in the example below...

a)



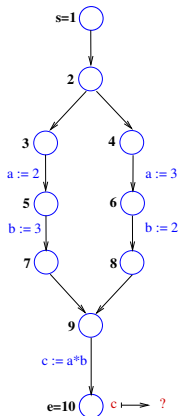
b)



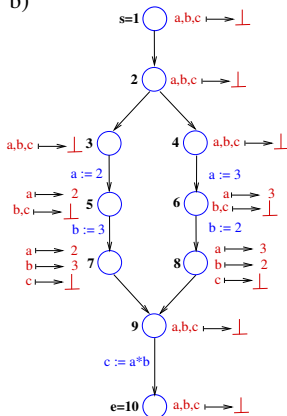
The Impact of Monotonicity on SCs (2)

...it is not in the only slightly modified example below:

a)



b)



The Proposal of Kam and Ullman

To **improve** on this **situation**

- ▶ **Kam and Ullman** propose using a slightly modified fixed point approach to cope with (non-distributive) **monotonic DFA** problems.

We call this **approach** the

- ▶ **Kam/Ullman *MaxFP* approach** (or ***Q-MaxFP* approach**).

Chapter 5.7.2

The *Q-MaxFP* Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

Chapter 5.7.2.1

Node-labelled Flow Graphs

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

506/185

Preliminaries

Following Kam and Ullman, we start considering a **node-labelled SI flow graph**¹

$$\blacktriangleright G = (N, E, \mathbf{s}, \mathbf{e})$$

Let

$$\blacktriangleright \mathcal{S}_G = (\widehat{\mathcal{C}}, \llbracket \rrbracket, c_s, fw)$$

be a **monotonic** (non-distributive) **DFA specification** with

$$\llbracket \rrbracket : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

¹We will adapt the Q approach to edge-labelled flow graphs later.

The *MaxFP* Approach (N-labelled Graphs)

...adapted to node-labelled SI graphs:

Equation System 5.7.2.1.1 (*MaxFP* EQS)

$$N\text{-inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \bigcap \{X\text{-inf}(m) \mid m \in \text{pred}(n)\} & \text{otherwise} \end{cases}$$

$$X\text{-inf}(n) = \llbracket n \rrbracket(N\text{-inf}(n))$$

...from its counterpart for edge-labelled SI graphs:

EQS 3.4.1 (*MaxFP* EQS) – recalled

$$\text{inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \bigcap \{ \llbracket (m, n) \rrbracket(\text{inf}(m)) \mid m \in \text{pred}(n) \} & \text{otherwise} \end{cases}$$

Rewriting the *MaxFP* Equation System

...for edge-labelled SI graphs:

EQS 5.7.2.1.1 (*MaxFP* EQS) – recalled

$$N\text{-inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \bigcap \{X\text{-inf}(m) \mid m \in \text{pred}(n)\} & \text{otherwise} \end{cases}$$

$$X\text{-inf}(n) = \llbracket n \rrbracket(N\text{-inf}(n))$$

...by expanding $N\text{-inf}(n)$ in the equation for $X\text{-inf}(n)$:

Equation System 5.7.2.1.2 (*MaxFP* EQS)

$$N\text{-inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \bigcap \{X\text{-inf}(m) \mid m \in \text{pred}(n)\} & \text{otherwise} \end{cases}$$

$$X\text{-inf}(n) = \begin{cases} \llbracket \mathbf{s} \rrbracket(c_s) & \text{if } n = \mathbf{s} \\ \llbracket n \rrbracket(\bigcap \{X\text{-inf}(m) \mid m \in \text{pred}(n)\}) & \text{otherwise} \end{cases}$$

...essential: Joining information early.

The Q -MaxFP Approach (N-labelled Graphs)

Equation System 5.7.2.1.3 (Q -MaxFP EQS)

$$NQ-inf(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \sqcap \{XQ-inf(m) \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

$$XQ-inf(n) = \begin{cases} \llbracket n \rrbracket(c_s) & \text{if } n = \mathbf{s} \\ \sqcap \{\llbracket n \rrbracket(XQ-inf(m)) \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

...essential: Delaying to joining (by \sqcap) information.

MaxFP Approach vs. Q-MaxFP Approach

MaxFP Approach – joining information early (eagerly):

EQS 5.7.2.1.2 (MaxFP EQS) – recalled

$$N\text{-inf}(n) = \begin{cases} c_s & \text{if } n = s \\ \prod\{X\text{-inf}(m) \mid m \in \text{pred}(n)\} & \text{otherwise} \end{cases}$$

$$X\text{-inf}(n) = \begin{cases} \llbracket s \rrbracket(c_s) & \text{if } n = s \\ \llbracket n \rrbracket(\prod\{X\text{-inf}(m) \mid m \in \text{pred}(n)\}) & \text{otherwise} \end{cases}$$

Q-MaxFP Approach – joining information late (lazily):

EQS 5.7.2.1.3 (Q-MaxFP EQS) – recalled

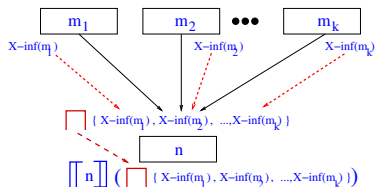
$$NQ\text{-inf}(n) = \begin{cases} c_s & \text{if } n = s \\ \prod\{XQ\text{-inf}(m) \mid m \in \text{pred}(n)\} & \text{otherwise} \end{cases}$$

$$XQ\text{-inf}(n) = \begin{cases} \llbracket s \rrbracket(c_s) & \text{if } n = s \\ \prod\{\llbracket n \rrbracket(XQ\text{-inf}(m)) \mid m \in \text{pred}(n)\} & \text{otherwise} \end{cases}$$

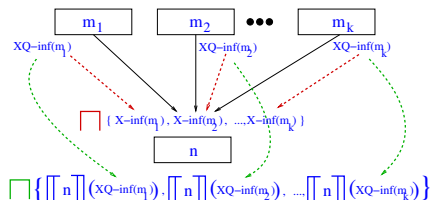
Illustrating the Essential Difference

...between *MaxFP* and *Q-MaxFP* approach:

MaxFP Approach



Q-MaxFP Approach



Note, for

- ▶ **distributive** DFA problems, we have:

$$\llbracket n \rrbracket (\bigcap \{X\text{-inf}(m) \mid m \in \text{pred}(n)\}) = \bigcap \{ \llbracket n \rrbracket (XQ\text{-inf}(m)) \mid m \in \text{pred}(n) \}$$

- ▶ **monotonic** DFA problems, we have:

$$\llbracket n \rrbracket (\bigcap \{X\text{-inf}(m) \mid m \in \text{pred}(n)\}) \sqsubseteq \bigcap \{ \llbracket n \rrbracket (XQ\text{-inf}(m)) \mid m \in \text{pred}(n) \}$$

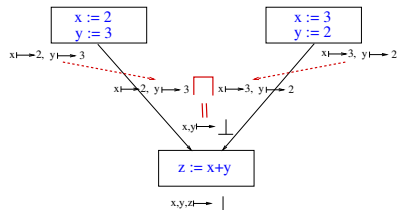
Hence, for

- ▶ **distributive** DFA problems, the *MaxFP* and *Q-MaxFP* solution **coincide** (Lemma 5.7.2.1.5).
- ▶ **monotonic** DFA problems, the *MaxFP* solution is a **lower approximation** of the *Q-MaxFP* solution (Lemma 5.7.2.1.6).

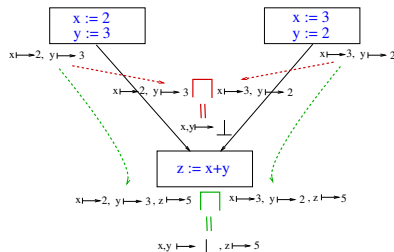
Thus, for

...(non-distributive) **monotonic** DFA problems, the *Q-MaxFP* approach gets an **advantage** over the *MaxFP* approach as is demonstrated below considering \mathcal{S}_C^{sc} for **simple constants**:

MaxFP Approach



Q-MaxFP Approach



...but not for **distributive** DFA problems.

Eager and Lazy Fixed Point Approach

The Equation Systems 5.7.2.1.2 and 5.7.2.1.3 give rise to consider the

- ▶ *MaxFP* approach
- ▶ *Q-MaxFP* approach

the

- ▶ *eager* (eagerly joining)
- ▶ *lazy* (lazily joining)

fixed point approach, respectively.

MaxFP, Q-MaxFP Solutions (N-lab'ed Graphs)

Definition 5.7.2.1.4 (MaxFP, Q-MaxFP Solution)

For every node $n \in N$, the *MaxFP* and *Q-MaxFP Solutions* of \mathcal{S}_G are defined by

- ▶ $N\text{-MaxFP}_{\mathcal{S}_G}(n) =_{df} \nu\text{-}N\text{-inf}(n)$
 $X\text{-MaxFP}_{\mathcal{S}_G}(n) =_{df} \nu\text{-}X\text{-inf}(n)$
- ▶ $NQ\text{-MaxFP}_{\mathcal{S}_G}(n) =_{df} \nu\text{-}NQ\text{-inf}(n)$
 $XQ\text{-MaxFP}_{\mathcal{S}_G}(n) =_{df} \nu\text{-}XQ\text{-inf}(n)$

where

- ▶ $\nu\text{-}N\text{-inf}, \nu\text{-}X\text{-inf}(n) : N \rightarrow \mathcal{C}$
- ▶ $\nu\text{-}NQ\text{-inf}(n), \nu\text{-}XQ\text{-inf}(n) : N \rightarrow \mathcal{C}$

denote the *greatest solutions* of [Equation System 5.7.2.1.2](#) and [Equation System 5.7.2.1.3](#), respectively.

Q Approach: Improvement (N-lab'ed Graphs)

We have:

Lemma 5.7.2.1.5 (Distributivity: Q Neutrality)

If \mathcal{S}_G is a **distributive** DFA problem, then $\forall n \in N$:

- ▶ $N\text{-MOP}_{\mathcal{S}_G}(n) = NQ\text{-MaxFP}_{\mathcal{S}_G}(n) = N\text{-MaxFP}_{\mathcal{S}_G}(n)$
- ▶ $X\text{-MOP}_{\mathcal{S}_G}(n) = XQ\text{-MaxFP}_{\mathcal{S}_G}(n) = X\text{-MaxFP}_{\mathcal{S}_G}(n)$

Lemma 5.7.2.1.6 (Monotonicity: Q Improvement)

If \mathcal{S}_G is a **monotonic** DFA problem, then $\forall n \in N$:

- ▶ $N\text{-MOP}_{\mathcal{S}_G}(n) \supseteq NQ\text{-MaxFP}_{\mathcal{S}_G}(n) \supseteq N\text{-MaxFP}_{\mathcal{S}_G}(n)$
- ▶ $X\text{-MOP}_{\mathcal{S}_G}(n) \supseteq XQ\text{-MaxFP}_{\mathcal{S}_G}(n) \supseteq X\text{-MaxFP}_{\mathcal{S}_G}(n)$

In general, all inclusions are proper.

Computing $MaxFP$ and $Q-MaxFP$ Solution: Pragmatics (N-labelled Graphs)

Note

- ▶ $\nu-XQ-inf$ and $\nu-X-inf$ can be computed without referring to (approximations of) $\nu-NQ-inf$ and $\nu-N-inf$, respectively.

Once

- ▶ $\nu-XQ-inf$ and $\nu-X-inf$ have been computed $\nu-NQ-inf$ and $\nu-N-inf$ can be computed by visiting each node once. No further fixed point computation or iteration is required.

Computing *MaxFP* and *Q-MaxFP* Solution: Algorithms (N-labelled Graphs)

The *greatest solutions* of

- ▶ Equation System 5.7.2.1.2
- ▶ Equation System 5.7.2.1.3

can be computed in the same fashion as the greatest solution of Equation System 3.4.1.

We denote these algorithms, which are straightforward generic adaptations of Algorithm 3.4.3, by

- ▶ *MaxFP* Algorithm 5.7.2.1.7²
- ▶ *Q-MaxFP* Algorithm 5.7.2.1.8²

respectively.

²We omit presenting the algorithms explicitly.

Q Approach: Main Results (N-lab'ed Graphs)

Theorem 5.7.2.1.9 (Termination)

The generic *MaxFP* Algorithm 5.7.2.1.7 and the generic *Q-MaxFP* Algorithm 5.7.2.1.8 terminate with the *MaxFP* solution and the *Q-MaxFP* solution of \mathcal{S}_G , respectively, if (1) the DFA lattice $\hat{\mathcal{C}}$ satisfies the **descending chain condition**, and (2) the DFA functional $\llbracket \cdot \rrbracket$ is **monotonic**.

Theorem 5.7.2.1.10 (Safety)

The *MaxFP* solution and the *Q-MaxFP* solution of \mathcal{S}_G are safe (i.e., lower) approximations of the *MOP* solution of \mathcal{S}_G satisfying for every node $n \in N$:

- ▶ $N\text{-MOP}_{\mathcal{S}_G}(n) \sqsupseteq NQ\text{-MaxFP}_{\mathcal{S}_G}(n) \sqsupseteq N\text{-MaxFP}_{\mathcal{S}_G}(n)$
- ▶ $X\text{-MOP}_{\mathcal{S}_G}(n) \sqsupseteq XQ\text{-MaxFP}_{\mathcal{S}_G}(n) \sqsupseteq X\text{-MaxFP}_{\mathcal{S}_G}(n)$

if the DFA functional $\llbracket \cdot \rrbracket$ is **monotonic**.

In general, all inclusions are proper.

Chapter 5.7.2.2

Edge-labelled Flow Graphs

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

Q Approach: From N to E-labelled Graphs

...focusing on **node exits** of **node-labelled SI flow graphs**

$$XQ-inf(n) = \begin{cases} \llbracket n \rrbracket(c_s) & \text{if } n = \mathbf{s} \\ \bigcap \{ \llbracket n \rrbracket(XQ-inf(m)) \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

yields the key for adapting the *Q-MaxFP* approach to edge-labelled SI flow graphs.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

522/185

The Q -MaxFP Approach (E-labelled Graphs)

...adapted to edge-labelled SI graphs.

Equation System 5.7.2.2.1 (Q -MaxFP EQS)

$$Q\text{-inf}(e) = \begin{cases} \llbracket e \rrbracket(c_s) & \text{if } \text{start}(e) = \mathbf{s} \\ \bigcap \{ \llbracket e \rrbracket(Q\text{-inf}(f)) \mid f \in \text{pred}(e) \} & \text{otherwise} \end{cases}$$

where $\text{pred}(e) =_{df} \{ f \mid \text{end}(f) = \text{start}(e) \}$.

Recall and compare with:

$$XQ\text{-inf}(n) = \begin{cases} \llbracket n \rrbracket(c_s) & \text{if } n = \mathbf{s} \\ \bigcap \{ \llbracket n \rrbracket(XQ\text{-inf}(m)) \mid m \in \text{pred}(n) \} & \text{otherwise} \end{cases}$$

Q -MaxFP Solution (E-labelled Graphs)

Definition 5.7.2.2.2 (Q -MaxFP Solution)

For every edge $e \in E$, the Q -MaxFP Solution of \mathcal{S}_G is defined by

$$\blacktriangleright Q\text{-MaxFP}_{\mathcal{S}_G}(e) =_{df} \nu\text{-}Q\text{-inf}(e)$$

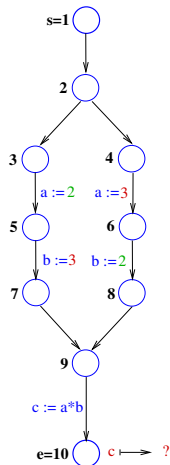
where

$$\blacktriangleright \nu\text{-}Q\text{-inf} : E \rightarrow \mathcal{C}$$

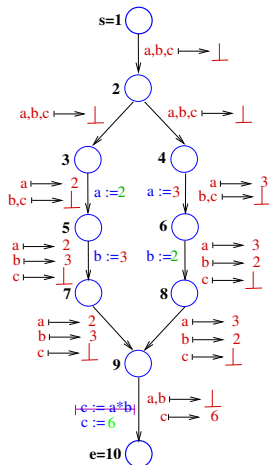
denotes the greatest solution of Equation System 5.7.2.2.1.

Illustrating the *Q-MaxFP* Approach and the *Q-MaxFP* Solution (E-labelled Graphs)

a)



b)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

525/185

Q -MaxFP Solution (E-labelled Graphs): Induced Node Annotation

The greatest solution of Equation System 5.7.2.2.1, ν - Q -inf(e), $e \in E$, induces an annotation of the nodes of G as follows:

Definition 5.7.2.2.3 (Induced Node Annotation)

For every node $n \in N$, we define:

$$Q\text{-MaxFP}(n) =_{df} \nu\text{-}Q\text{-inf}(n) =_{df} \begin{cases} c_s & \text{if } n = s \\ \prod \{ \nu\text{-}Q\text{-inf}(e) \mid \text{end}(e) = n \} & \text{otherwise} \end{cases}$$

Note: There is no fixed point computation involved in computing ν - Q -inf(n), $n \in N$.

Q Approach: Improvement (E-lab'ed Graphs)

We have:

Lemma 5.7.2.2.4 (Q Improvement Lemma)

For every node $n \in N$, for every $e \in E$ with $end(e) = n$, we have:

$$\blacktriangleright Q\text{-MaxFP}_{S_G}(e) \supseteq Q\text{-MaxFP}_{S_G}(n) \supseteq \text{MaxFP}_{S_G}(n)$$

In general, all inclusions are proper.

Computing the Q -MaxFP Solution: Algorithm (E-labelled Graphs)

The **greatest solution** of

- ▶ Equation System 5.7.2.2.1

can be computed in the same fashion as the greatest solution of Equation System 3.4.1.

We denote this algorithm, which is a generic straightforward adaption of Algorithm 3.4.3, by

- ▶ Q -MaxFP-Algorithm 5.7.2.2.5³

³We omit presenting the algorithm explicitly.

Q Approach: Main Results (E-lab'ed Graphs)

Theorem 5.7.2.2.6 (Termination)

The generic *Q-MaxFP* Algorithm 5.7.2.2.5 terminates with the *Q-MaxFP* solution of \mathcal{S}_G , if (1) the DFA lattice $\hat{\mathcal{C}}$ satisfies the **descending chain condition**, and (2) the DFA functional $\llbracket \cdot \rrbracket$ is **monotonic**.

Theorem 5.7.2.2.7 (Safety)

The *Q-MaxFP* solution is a safe (i.e., lower) approximation of the *MOP* solution of \mathcal{S}_G satisfying for every node $n \in N$, for every $e \in E$ with $\text{end}(e) = n$:

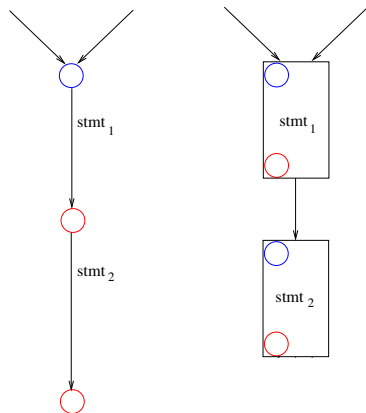
$$\begin{aligned} \blacktriangleright \text{MOP}_{\mathcal{S}_G}(n) &\sqsupseteq \text{Q-MaxFP}_{\mathcal{S}_G}(e) \\ &\sqsupseteq \text{Q-MaxFP}_{\mathcal{S}_G}(n) \sqsupseteq \text{MaxFP}_{\mathcal{S}_G}(n) \end{aligned}$$

if the DFA functional $\llbracket \cdot \rrbracket$ is **monotonic**.

In general, all inclusions are proper.

Q Approach: Choosing N or E-labelled Graphs

...at first sight, both variants appear equally suited.



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

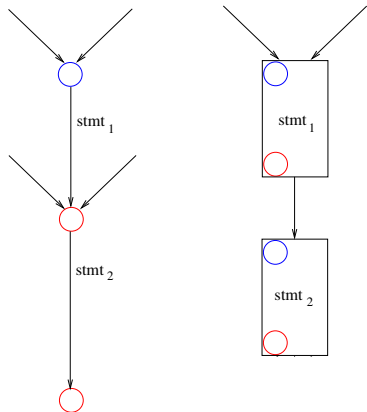
5.10

5.11

530/185

Q Approach: Choosing N or E-labelled Graphs

However, a closer look reveals...



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

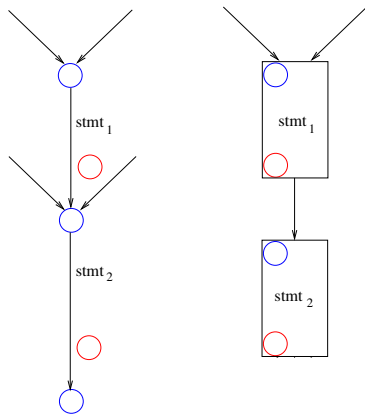
5.10

5.11

531/185

Q Approach: Choosing N or E-labelled Graphs

...edge-labelled graphs are more 'compact'



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

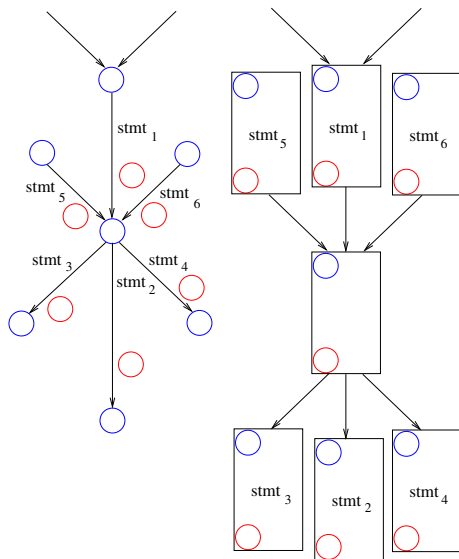
5.10

5.11

532/185

Q Approach: Choosing N or E-labelled Graphs

...making them more appropriate



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

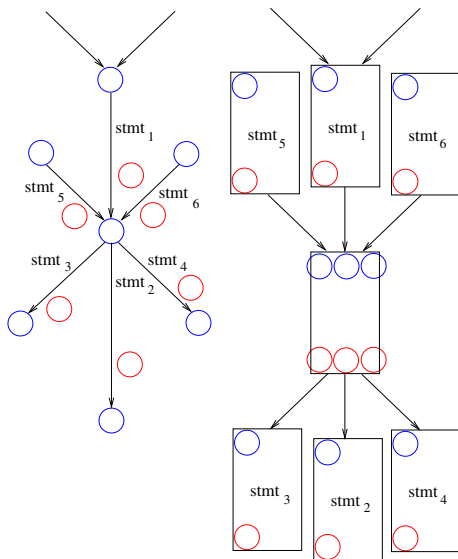
5.10

5.11

533/185

Q Approach: Choosing N or E-labelled Graphs

...for taking advantage of the Q heuristics.



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

534/185

Q Approach: Choosing N or E-labelled Graphs

The Q approach

- ▶ applies to both node and edge-labelled flow graphs.

The heuristics of the Q approach, however,

- ▶ is more effective on edge-labelled flow graphs than on node-labelled ones because of their greater compactness.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

535/185

Chapter 5.7.3

Q Constants: The Specification

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

Q Constants over \mathbb{Z} : DFA Specification

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be an edge-labelled SI flow graph.

Q Constants Specification

- Specification: $\mathcal{S}_G^{qc} =_{df} \mathcal{S}_G^{sc} = (\hat{\Sigma}', \llbracket \cdot \rrbracket_{sc}, \sigma_{\mathbf{s}}, fw)$

Note:

- Q constants (QCs) and simple constants (SCs) share the same specification.
- The only difference between the QC and SC problem is that \mathcal{S}_G^{sc} and \mathcal{S}_G^{qc} are fed into and solved by the *MaxFP* and *Q-MaxFP* approach, respectively.

Chapter 5.7.4

Termination, Safety, and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

Towards Safety and Termination

Because of $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{qc}, \sigma_s, fw) = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{sc}, \sigma_s, fw) = \mathcal{S}_G^{sc}$
we get as corollaries of Lemma 5.4.3.1, 5.4.3.2, and 5.4.3.3:

Corollary 5.7.4.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Corollary 5.7.4.2 (Monotonicity)

$\llbracket \cdot \rrbracket_{qc} (= \llbracket \cdot \rrbracket_{sc})$ is monotonic.

Corollary 5.7.4.3 (Non-Distributivity)

$\llbracket \cdot \rrbracket_{qc} (= \llbracket \cdot \rrbracket_{sc})$ is not distributive.

Termination and Safety/Conservativity

Theorem 5.7.4.4 (Termination)

Applied to $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{qc}, \sigma_s, fw)$, Algorithm 5.7.2.13 terminates with the *Q-MaxFP* solution of \mathcal{S}_G^{qc} .

Proof. Immediately with Corollary 5.7.4.1, Corollary 5.7.4.2, and Termination Theorem 5.7.2.14.

Theorem 5.7.4.5 (Safety/Conservativity)

Applied to $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', \llbracket \cdot \rrbracket_{qc}, \sigma_s, fw)$, Algorithm 5.7.2.13 is *MOP* conservative for \mathcal{S}_G^{qc} (i.e., it terminates with a lower approximation of the *MOP* solution of \mathcal{S}_G^{qc}).

Proof. Immediately with Corollary 5.7.4.2, Safety Theorem 5.7.2.15, and Termination Theorem 5.7.4.4.

Non-Coincidence

Theorem 5.7.4.6 (Non-Coincidence/Non-Opt.)

Applied to $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', \llbracket \rrbracket_{qc}, \sigma_s, fw)$, Algorithm 5.7.2.13 is in general not *MOP* optimal for \mathcal{S}_G^{qc} (i.e., it terminates with a properly lower approximation of the *MOP* solution of \mathcal{S}_G^{qc}).

Proof. Immediately with Corollary 5.7.4.3, Safety Theorem 5.7.2.15, and Termination Theorem 5.7.4.4.

Corollary 5.7.4.7 (Safety, Non-Coincidence)

The *Q-MaxFP* solution for \mathcal{S}_G^{qc} , is always a safe approximation of the *MOP* solution of \mathcal{S}_G^{qc} . In general, the *MOP* solution and the *Q-MaxFP* solution of \mathcal{S}_G^{qc} do not coincide.

Chapter 5.7.5

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

542/185

Soundness and Completeness of $MOP_{S_G^{qc}}$

The equality $S_G^{qc} = S_G^{sc}$ implies $MOP_{S_G^{qc}} = MOP_{S_G^{sc}}$ yielding as an immediate corollary of Theorem 5.4.4.1:

Corollary 5.7.5.1 (Soundness and Completeness)

The MOP solution of S_G^{qc} is

1. sound and complete for the variable constant propagation problem C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^V}(n) = MOP_{S_G^{qc}, \sigma_s}(n)$$

2. sound but not complete for the term constant propagation problem C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^T}(n) \sqsubseteq_{\Sigma'} MOP_{S_G^{qc}, \sigma_s}^T(n)$$

In general, the inclusion is proper.

Soundness and Completeness of $Q\text{-MaxFP}_{S_G^{qc}}$

Corollary 5.7.5.2 (Soundness and Completeness)

The $Q\text{-MaxFP}$ solution of S_G^{qc} is

1. sound but not complete for C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_G^V}(n) \not\sqsubseteq_{\Sigma'} Q\text{-MaxFP}_{S_G^{qc}, \sigma_s}(n)$$

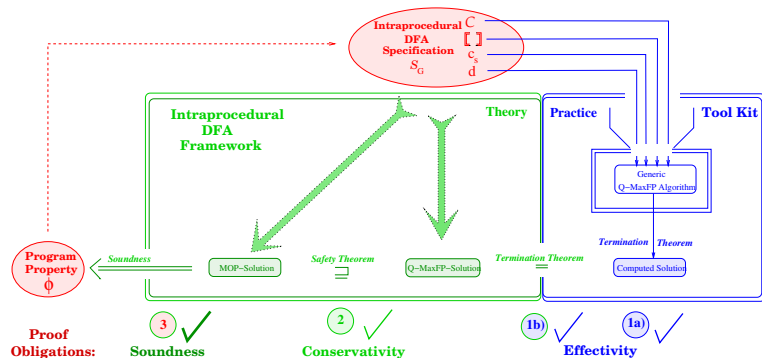
2. sound but not complete for C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_G^T}(n) \not\sqsubseteq_{\Sigma'} Q\text{-MaxFP}_{S_G^{qc}, \sigma_s}^T(n)$$

In general, both inclusions are proper.

In Summary: Soundness, *MOP*-Conservativity

...of \mathcal{S}_G^{qc} for Q constants proven:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

Chapter 5.7.6

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

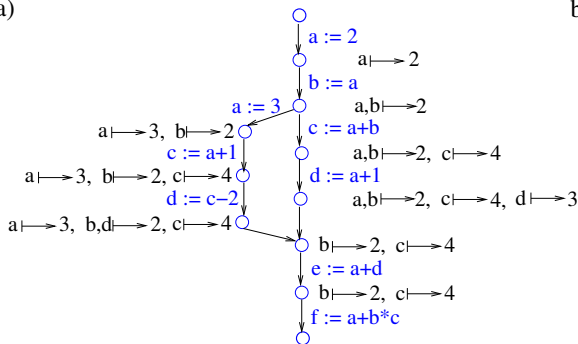
5.9

5.10

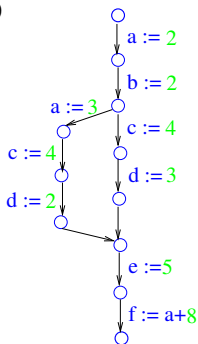
5.11

Q Constants over \mathbb{Z} : Illustrating Example

a)



b)



...all terms are **Q constants** except of $a + 8$, which, however, is not a (non-deterministic) constant at all.

Chapter 5.7.7

Summary

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.7.1

5.7.2

5.7.3

5.7.4

5.7.5

5.7.6

5.7.7

5.8

5.9

5.10

5.11

The Essence of the Q Approach

The Q approach

- ▶ a **heuristically** copes with the information loss caused by 'early (eagerly)' joining information in the *MaxFP* approach for (non-distributive) **monotonic** DFA problems.

Intuitively, the Q approach

- ▶ accomplishes 'a look-ahead of one edge' by joining information 'late (lazily)' avoiding thereby the loss of information in part.

Benefits and Limitations of the Q Approach

Benefits

Introduced and proposed with an application to constant propagation (i.e., **Q constants**)

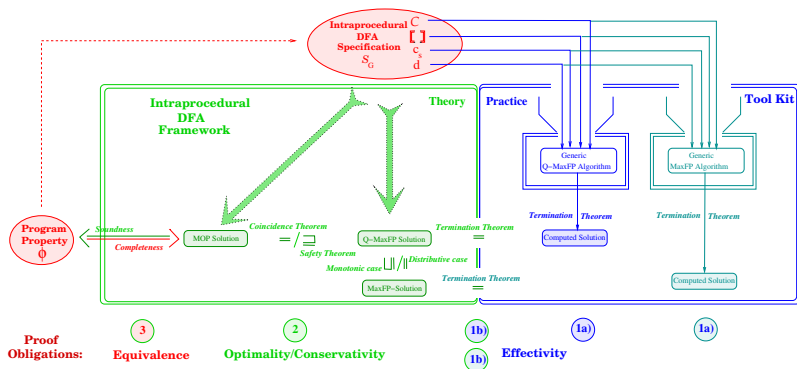
- ▶ the **Q approach** can beneficially be used for **every monotonic DFA problem** at (in practice) **almost no additional costs** compared to the standard *MaxFP* approach.

Limitations

- ▶ In practice, the **impact of the Q approach on improving the precision of analysis results will be limited** because its look-ahead heuristics is limited to one edge (though enhanced by the effect of fixed point computation).
- ▶ Avoiding the loss of information by joining information completely, the look-ahead would need to be arbitrarily large in general; there is **no finite upper limit on the required look-ahead for avoiding information loss**.

Overall

...the **Q** approach is a valuable addition to the practitioner's tool box coming and offering advantages for monotonic DFA problems at almost no costs:



Chapter 5.8

Finite Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

Chapter 5.8.1

Background and Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

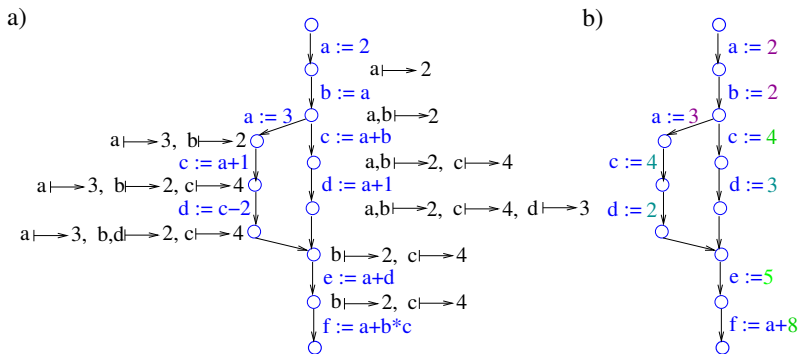
5.8.9

5.8.10

CP by Q Constants: Everything Alright?

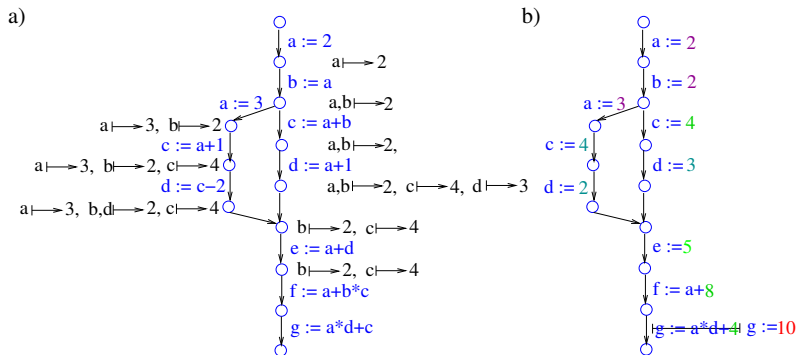
Note

- ▶ All terms are **Q constants** (except of $a + 8$, which, however, is not a (non-deterministic) constant at all).



Unfortunately not

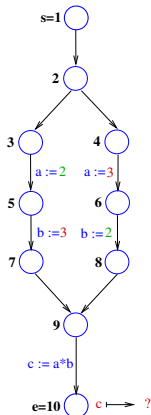
... $a * d$ and g are constants of value 6 and 10, respectively, however, they are **no** Q constants.



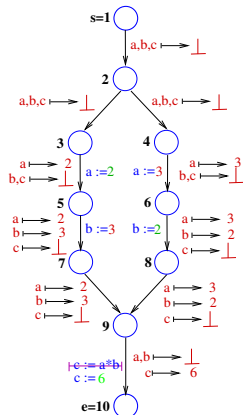
Achievement of the Q Constants Heuristics

...joining information 'lazily' accomplishes a 'look-ahead' of 1 edge after a join node:

a)



b)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

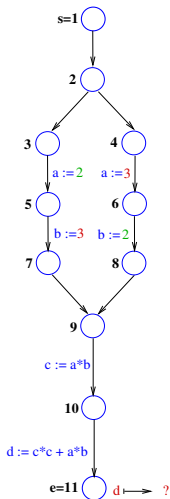
5.8.9

5.8.10

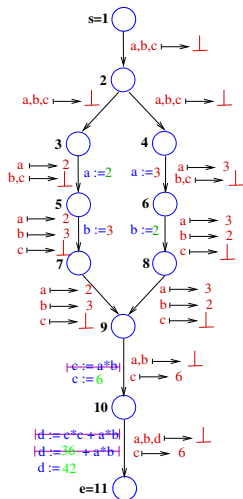
Limitations of the Q Constants Heuristics

...but not of 2 edges as required here (or even more in general):

a)

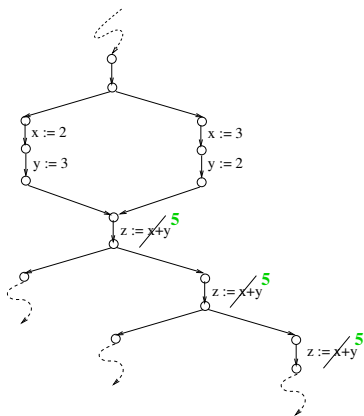


b)



The Look-Ahead Challenge

...a need for a look-ahead of unlimited length in general:



After Finite Constants Propagation

...the approach for **finite constants** deals **systematically** with this **challenge**.

Chapter 5.8.2

Finite Constants: The Very Idea

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

559/185

Finite Constants: The Very Idea

Intuitively

- ▶ finite constants achieve a look-ahead of arbitrary but finite depth.

Technically, this is achieved by

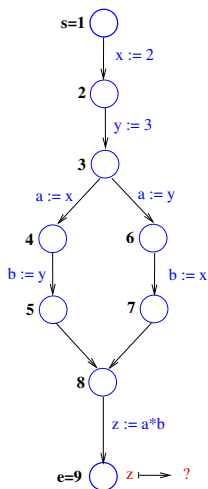
- ▶ pre-computing for every program point a finite set of interesting terms and
- ▶ focusing the analysis at every program point to this set of terms instead of the program variables only.

Hence, unlike the CP algorithms considered so far

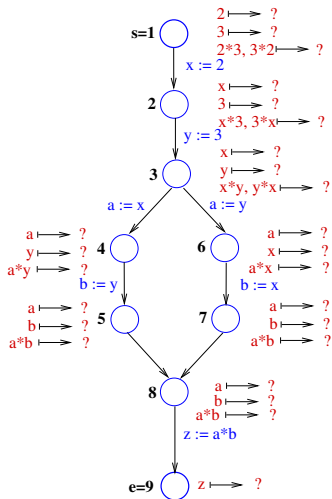
- ▶ the algorithm for finite constants is a CP algorithm for C^T , not for C^V .

Finite Constants: Pre-Computing Term Sets

a)

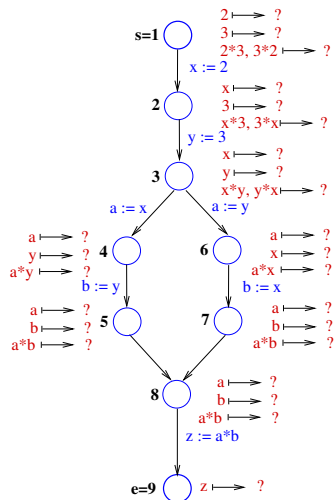


b)

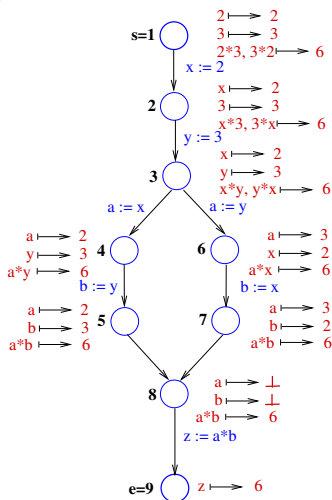


Finite Constants: The Analysis (Conceptually)

a)



b)



Chapter 5.8.3

Finite Operational Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

563/185

Backward Substitution for Instructions

Definition 5.8.3.1 (Backward Substitution δ)

- ▶ Let $\iota \equiv x := t$ be an assignment instruction. The **backward substitution** of ι is the function $\delta_\iota : \mathbf{T} \rightarrow \mathbf{T}$ defined by

$$\forall t' \in \mathbf{T}. \delta_\iota(t') =_{df} t'[t/x]$$

- ▶ Let $\iota \equiv skip$ be the empty instruction. The **backward substitution** of ι is the **identical mapping** on the set of terms \mathbf{T} , i.e., $\delta_\iota =_{df} Id_{\mathbf{T}}$, where $Id_{\mathbf{T}} : \mathbf{T} \rightarrow \mathbf{T}$ is defined by $Id_{\mathbf{T}} =_{df} \lambda t. t$.

Backward Substitution for Paths

Definition 5.8.3.2 (Extending δ to Paths)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph. The backward substitutions δ_{ι_e} of instructions at edges $e \in E$ are extended onto paths $p = \langle e_1, e_2, \dots, e_q \rangle$ in G by defining:

$$\delta_p =_{df} \begin{cases} Id_{\mathcal{T}} & \text{if } q < 1 \\ \delta_{\langle e_1, \dots, e_{q-1} \rangle} \circ \delta_{\iota_{e_q}} & \text{otherwise} \end{cases}$$

Substitution Lemma: Relating δ and θ

Lemma 5.8.3.3 (Substitution Lemma for Edges)

$$\forall t \in \mathbf{T}. \forall e \in E. \forall \sigma \in \Sigma. \mathcal{E}(\delta_{l_e}(t))(\sigma) = \mathcal{E}(t)(\theta_{l_e}(\sigma))$$

Lemma 5.8.3.4 (Substitution Lemma for Paths)

$$\forall t \in \mathbf{T}. \forall n \in N. \forall \sigma \in \Sigma. \forall p \in \mathbf{P}[m, n].$$

$$\mathcal{E}(\delta_p(t))(\sigma) = \mathcal{E}(t)(\theta_p(\sigma))$$

Corollary 5.8.3.5 (Substitution L. for Paths from \mathbf{s})

$$\forall t \in \mathbf{T}. \forall n \in N. \forall \sigma \in \Sigma. \forall p \in \mathbf{P}[\mathbf{s}, n].$$

$$\mathcal{E}(\delta_p(t))(\sigma_s) = \mathcal{E}(t)(\theta_p(\sigma_s))$$

t -Associated Paths

Definition 5.8.3.6 (t -Associated Path)

Let $p = \langle e_1, e_2, \dots, e_q \rangle$ be a path, and let $t \in \mathbf{T}$ be a term. The t -associated path p_t for p is defined by

$$p_t = \langle (t_1, e_1), (t_2, e_2), \dots, (t_q, e_q) \rangle$$

with $t_q = \delta_{\iota_{e_q}}(t)$ and $t_j = \delta_{\iota_{e_j}}(t_{j+1})$ for all $1 \leq j < q$.

Corollary 5.8.3.7 (Subst. L. for t -assoc. Paths)

$$\forall t \in \mathbf{T}. \forall n \in \mathbf{N}. \forall \sigma \in \Sigma. \forall p \in \mathbf{P}[\mathbf{s}, n].$$

$$\mathcal{E}(t_s)(\sigma_s) = \mathcal{E}(t)(\theta_p(\sigma_s))$$

where $p_t = \langle (t_1, e_1), (t_2, e_2), \dots, (t_q, e_q) \rangle$ is the t -associated path for p and $t_s \equiv t_1$.

Relevant Paths

Definition 5.8.3.8 (Relevant Path of length k)

A t -associated path $p_t = \langle (t_1, e_1), (t_2, e_2), \dots, (t_q, e_q) \rangle$ is called a **relevant path of length** (at most) k for t and n , iff

- ▶ $dst(e_q) = n \wedge t_q = \delta_{\iota_{e_q}}(t)$
- ▶ $q = k \vee (q < k \wedge src(e_1) = \mathbf{s})$
- ▶ $\forall i, j \in \{1, \dots, q\}. (t_i, e_i) = (t_j, e_j) \Rightarrow i = j$

The set of all **relevant paths of length** (at most) k for t and n is denoted by **RP** $_k(t, n)$.

Definition 5.8.3.9 (Relevant Paths from \mathbf{s} to n)

The set of all **relevant paths** from start node \mathbf{s} to node n is denoted by **RP** $_{IN}(t, n)$.

Finite Operational Constants

Definition 5.8.3.10 ((Finite) Operat. d -Constants)

Let $k \in \mathbb{IN} \cup \{\mathbb{IN}\}$, $d \in \text{ID} \setminus \{\perp\}$, $n \in N$, $\sigma_s \in \Sigma_{\text{Init}}$. Then:

1. t is a k -constant of value d at node n for σ_s ,
 $t \in C_{k,\sigma_s}(n, d)$, iff

$$\forall p_t = \langle (t_1, e_1), (t_2, e_2), \dots, (t_q, e_q) \rangle \in \mathbf{RP}_k(t, n).$$

$$(\text{start}(e_1) = \mathbf{s} \Rightarrow \mathcal{E}(t_1)(\sigma_s) = d) \vee$$

$$(\text{start}(e_1) \neq \mathbf{s} \Rightarrow \mathcal{E}(t_1)(\sigma_\perp) = d)$$

2. the set of **finite operational constants** of value d at node n for σ_s , $C_{\text{fop},\sigma_s}(n, d)$, is defined by:

$$C_{\text{fop},\sigma_s}(n, d) =_{df} \bigcup \{C_{k,\sigma_s}(n, d) \mid k \in \mathbb{IN}\}$$

3. the set of **operational constants** of value d at node n for σ_s , $C_{\text{op},\sigma_s}(n, d)$, is defined by:

$$C_{\text{op},\sigma_s}(n, d) =_{df} \bigcup \{C_{k,\sigma_s}(n, d) \mid k \in \mathbb{IN} \cup \{\mathbb{IN}\}\}$$

Finite Operational Constants: Main Result

Definition 5.8.3.11 ((Finite) Operational Constants)

Let $n \in N$ and $\sigma_s \in \Sigma_{Init}$. Then the set of

1. **finite operational constants** at node n for σ_s , $C_{fop,\sigma_s}(n)$, is defined by: $C_{fop,\sigma_s}(n) =_{df} \bigcup \{C_{fop,\sigma_s}(n, d) \mid d \in ID \setminus \{\perp\}\}$
2. **operational constants** at node n for σ_s , $C_{op,\sigma_s}(n)$, is defined by: $C_{op,\sigma_s}(n) =_{df} \bigcup \{C_{op,\sigma_s}(n, d) \mid d \in ID \setminus \{\perp\}\}$

Theorem 5.8.3.12 (Main Result)

For all $n \in N$, $\sigma_s \in \Sigma_{Init}$, and $d \in ID \setminus \{\perp\}$ we have:

1. $C_{op,\sigma_s}(n) = C_{G,\sigma_s}^T(n)$
2. $\exists k \in \mathbb{N}. C_{fop,\sigma_s}(n, d) = C_{k,\sigma_s}(n, d)$

Chapter 5.8.4

Finite Denotational Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

571/185

Partitions

Definition 5.8.4.1 (Partitions)

Let $T \subseteq \mathbf{T}$ be a set of terms. Then we define:

- ▶ $Part(T)$ denotes the set of all **partitions** of T .
- ▶ $Part =_{df} \bigcup \{Part(T) \mid T \subseteq \mathbf{T}\}$
- ▶ $CSet(p) =_{df} \{t \mid t \text{ lies in a class of } p\}$, $p \in Part$, denotes the **carrier set** of partition p .

Note

- ▶ Partitions can be viewed as **equivalence relations** on their **carrier sets**.
- ▶ This allows us to define a **meet** and a **join operation** on $Part$ in terms of **set theoretical intersection** and **union** of the **equivalence relations** corresponding to the partitions, respectively.

Induced Partitions

The **evaluation function** \mathcal{E} induces for every set of terms $T \subseteq \mathbf{T}$ and every initial state $\sigma_s \in \Sigma_{Init}$ a unique partition with carrier set T , which we denote by $Part_{\mathcal{E}, \sigma_s}(T)$.

Definition 5.8.4.2 (Induced Partitions)

Let $T \subseteq \mathbf{T}$, and $\sigma_s \in \Sigma_{Init}$. Then we define:

- ▶ $\forall t_1, t_2 \in T. (t_1, t_2) \in Part_{\mathcal{E}, \sigma_s}(T)$
 $\iff \mathcal{E}(t_1)(\sigma_s) = \mathcal{E}(t_2)(\sigma_s)$
- ▶ $Part_{\mathcal{E}, \sigma_s} =_{df} \{Part_{\mathcal{E}, \sigma_s}(T) \mid T \subseteq \mathbf{T}\}$ denotes the set of **initial partitions** induced by σ_s .

Complete Lattice of the Set of Partitions

Lemma 5.8.4.3 (Complete Lattice)

The quintuple

$$\widehat{Part} =_{df} (Part, \sqcap, \sqcup, \sqsubseteq, \{\{t\} \mid t \in \mathbf{T}\}, \{\mathbf{T}\})$$

is a complete lattice, where \sqcap , \sqcup , and \sqsubseteq are given by the set theoretical intersection \cap , union \cup , and subset relation \subseteq of the equivalence relations represented by the partitions, respectively.

Local Abstract Semantics

Definition 5.8.4.4 (Local Abstract Semantics)

The semantic functional

$$\llbracket \cdot \rrbracket : E \rightarrow (Part \rightarrow Part)$$

defines a **local abstract semantics** (for edges) by

$$\forall e \in E. \forall p \in Part. \llbracket e \rrbracket(p) =_{df} \{(r, s) \mid (\delta_{l_e}(r), \delta_{l_e}(s)) \in p\}$$

Lemma 5.8.4.5 (Distributivity)

The local semantic functions $\llbracket e \rrbracket$, $e \in E$, are distributive.

Finite Denotational Constants

Definition 5.8.4.6 ((Finite) Denotat. d -Constants)

Let $\sigma_s \in \Sigma_{Init}$, $p_s \in Part_{\mathcal{E}, \sigma_s}$, $d \in ID \setminus \{\perp\}$, and $n \in N$. Then:

- ▶ t is a p_s -constant of value d at node n , $t \in C_{p_s}(n, d)$, iff $(t, d) \in \nu\text{-inf}_{p_s}(n)$, where $\nu\text{-inf}_{p_s}$ denotes the greatest solution of the *MaxFP* Equation System 3.4.1.
- ▶ the set of **finite denotational constants** of value d at node n for σ_s , $C_{fden, \sigma_s}(n, d)$, is defined by:

$$C_{fden, \sigma_s}(n, d) =_{df} \bigcup \{C_p(n, d) \mid p \in Part_{\mathcal{E}, \sigma_s} \wedge |CSet(p)| \in \mathbb{IN}\}$$

- ▶ the set of **denotational constants** of value d at node n for σ_s , $C_{den, \sigma_s}(n, d)$, is defined by:

$$C_{den, \sigma_s}(n, d) =_{df} \bigcup \{C_p(n, d) \mid p \in Part_{\mathcal{E}, \sigma_s}\}$$

Finite Denotational Constants: Main Result

Definition 5.8.4.7 ((Finite) Denotational Constants)

Let $n \in N$ and $\sigma_s \in \Sigma_{Init}$. Then the set of

1. **finite denotational constants** at node n for σ_s , $C_{fden, \sigma_s}(n)$, is defined by:

$$C_{fden, \sigma_s}(n) =_{df} \bigcup \{ C_{fden, \sigma_s}(n, d) \mid d \in ID \setminus \{\perp\} \}$$

2. **denotational constants** at node n for σ_s , $C_{den, \sigma_s}(n)$, is defined by: $C_{den, \sigma_s}(n) =_{df} \bigcup \{ C_{den, \sigma_s}(n, d) \mid d \in ID \setminus \{\perp\} \}$

Theorem 5.8.4.8 (Main Result)

For all $n \in N$, $\sigma_s \in \Sigma_{Init}$, and $d \in ID \setminus \{\perp\}$ we have:

1. $C_{den, \sigma_s}(n) = C_{G, \sigma_s}^T(n)$
2. $(\forall T \subseteq \mathbf{T}. |T| \in \mathbb{IN}). (\exists p_{fden} \in Part_{\mathcal{E}, \sigma_s}. |CSet(p_{fden})| \in \mathbb{IN} \wedge C_{fden, \sigma_s}(n, d) \cap T \subseteq C_{p_{fden}}(n, d))$

Looking ahead: Computability of ν -inf

...for finite initial partitions (i.e. with finite carrier sets).

Theorem 5.8.4.9 (ν -inf $_{p_s}$ Computability)

Let $\sigma_s \in \Sigma_{Init}$, and $p_s \in Part_{\mathcal{E}, \sigma_s}$ be an initial partition with finite carrier set $CSet(p_s)$, i.e., $|CSet(p_s)| \in \mathbb{IN}$. Then:

The *MaxFP Algorithm 3.4.3* terminates with the greatest solution of *Equation System 3.4.1*, i.e. it terminates with ν -inf $_{p_s}$.

In particular, *Algorithm 3.4.3* computes for every node $n \in N$ and value $d \in ID \setminus \{\perp\}$ the set $C_{p_s}(n, d)$ of p_s -constants of value d at node n .

Chapter 5.8.5

Finite Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

Equivalence

...of (finite) operational and (finite) denotational constants.

Theorem 5.8.5.1 (d -Equivalence)

For all $n \in N$, $\sigma_s \in \Sigma_{init}$, and $d \in ID \setminus \{\perp\}$ we have:

1. $C_{op, \sigma_s}(n, d) = \{t \mid (t, d) \in C_{G, \sigma_s}^T(n)\} = C_{den, \sigma_s}(n, d)$
2. $C_{fop, \sigma_s}(n, d) = C_{fden, \sigma_s}(n, d)$

Corollary 5.8.5.2 (Equivalence)

For all $n \in N$, $\sigma_s \in \Sigma_{init}$ we have:

1. $C_{op, \sigma_s}(n) = C_{G, \sigma_s}^T(n) = C_{den, \sigma_s}(n)$
2. $C_{fop, \sigma_s}(n) = C_{fden, \sigma_s}(n)$

Finite Constants

Definition 5.8.5.3 (Finite Constants)

Let $n \in N$ and $\sigma_s \in \Sigma_{Init}$. Then the set of

1. **finite constants** of value $d \in ID \setminus \{\perp\}$ at node n for σ_s , $C_{fc, \sigma_s}(n, d)$, is defined by:

$$\begin{aligned} C_{fc, \sigma_s}(n, d) &=_{df} C_{fop, \sigma_s}(n, d) \\ &= C_{fdn, p_s}(n, d) \end{aligned}$$

2. **finite constants** at node n for σ_s , $C_{fc, \sigma_s}(n)$, is defined by:

$$\begin{aligned} C_{fc, \sigma_s}(n) &=_{df} \bigcup \{ C_{fc, \sigma_s}(n, d) \mid d \in ID \setminus \{\perp\} \} \\ &= \bigcup \{ C_{fop, \sigma_s}(n, d) \mid d \in ID \setminus \{\perp\} \} \\ &= \bigcup \{ C_{fdn, p_s}(n, d) \mid d \in ID \setminus \{\perp\} \} \end{aligned}$$

Optimality of Finite Constants

...for acyclic control flow.

Theorem 5.8.5.4 (Optimality for Acyclic Graphs)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be an acyclic flow graph, let $n \in N$ and $\sigma_s \in \Sigma_{Init}$. Then:

$$C_{fop, \sigma_s}(n) = C_{G, \sigma_s}^T(n) = C_{fden, \sigma_s}(n)$$

This means, an algorithm computing the class of finite constants is for acyclic programs sound and complete for C^T .

Chapter 5.8.6

Deciding Finite Constants: Algorithm Sketch

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

The Decision Algorithm for Finite Constants

The decision algorithm for finite constants is a two-stage procedure, sketched below:

Algorithm 5.8.6.1 (Decision Algorithm Sketch)

Let $\sigma_s \in \Sigma_{Init}$ be an initial state, let $t \in \mathbf{T}$, and let $n \in N$.

1. Algorithm 5.8.6.2:

Compute a finite subset of the set of all terms

$TS_{G,\sigma_s}(n,t) \subseteq \mathbf{T}$ such that for all finite subsets of the set of all terms hold: $T_{finite} \subseteq \mathbf{T}$ satisfy:

$$\forall t \in \mathbf{T}. \forall d \in ID \setminus \{\perp\}.$$

$$t \in C_{Part_{\mathcal{E},\sigma_s}(T_{finite})}(n,d) \Rightarrow t \in C_{Part_{\mathcal{E},\sigma_s}(TS_{G,\sigma_s}(n,t))}(n,d)$$

2. MaxFP Algorithm 3.4.3:

Compute $C_{Part_{\mathcal{E},\sigma_s}(TS_{G,\sigma_s}(n,t))}(n,d)$ for all values $d \in ID \setminus \{\perp\}$.

Initial Partition: Carrier Set Computation (1)

Algorithm 5.8.6.2 (Computing $TS_{G,\sigma_s}(n, t)$)

...i.e., the carrier set of the initial start partition wrt t and n .

1. Transform G by adding a new node n' to N such that
 - ▶ n' has the same set of predecessors as n but no successors: $pred(n') = pred(n) \wedge succ(n') = \emptyset$
 - ▶ (m, n') , $m \in pred(n)$, represents the same assignment as (m, n) : $\iota_{(m,n')} = \iota_{(m,n)}$

Let $N' =_{df} N \cup \{n'\}$ and $E' =_{df} E \cup \{(m, n') \mid m \in pred(n)\}$ denote the sets of resulting nodes and edges, respectively.

2. Construct a regular expression ρ over N' representing the set of paths $\mathbf{P}[s, n']$ (e.g., using the algorithm of Tarjan, 1981).

(Note: '+' stands for non-deterministic branching, ';' for sequential composition, and '*' for indefinite looping).

Initial Partition: Carrier Set Computation (2)

3. Replace indefinite looping, \cdot^* , by **bounded looping**, \cdot^k , where k is the number of variables which occur on the left hand side of an assignment in the corresponding sub-expression of ρ , to arrive at the ($*$ -free) **regular expression** ρ_{bounded} .
4. Evaluate the functional $\Delta_\rho : \mathcal{P}(\mathbf{T}) \rightarrow \mathcal{P}(\mathbf{T})$, which is inductively defined by

$$\Delta_\rho(T) =_{df} \begin{cases} \{\delta_{\iota_\rho}(s) \mid s \in T\} & \text{if } \rho \in E' \\ \Delta_{\rho_1}(\Delta_{\rho_2}(T)) & \text{if } \rho = \rho_1; \rho_2 \\ \Delta_{\rho_1}(T) \cup \Delta_{\rho_2}(T) & \text{if } \rho = \rho_1 + \rho_2 \\ \bigcup \{\Delta_{\rho_1}^j(T) \mid j \in \{1, \dots, k\}\} & \text{if } \rho_1^k \end{cases}$$

for ρ_{bounded} and $\{t\} \in \mathcal{P}(\mathbf{T})$, i.e., evaluate $\Delta_{\rho_{\text{bounded}}}(\{t\})$.

(Note: $\Delta_\rho^0 =_{df} Id_{\mathcal{P}(\mathbf{T})}$ and $\Delta_\rho^j =_{df} \Delta_\rho^{j-1} \circ \Delta_\rho$, $j \geq 1$).

Initial Partition: Carrier Set Computation (3)

5. Set:

$$TS_{G, \sigma_s}(n, t) =_{df}$$

$$\{t' \in \mathbf{T}, d \in \text{ID} \mid t' \in \Delta_{\rho_{\text{bounded}}}(\{t\}) \wedge \mathcal{E}(t')(\sigma_s) = d \neq \perp\}$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

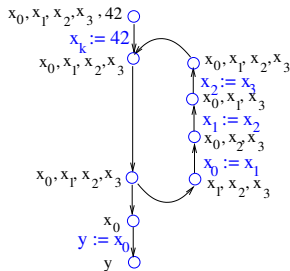
5.8.9

5.8.10

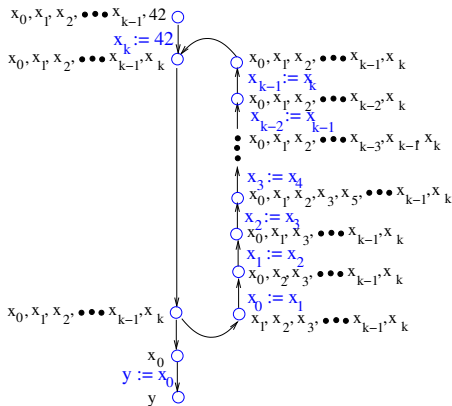
Illustrating the Idea

...of replacing indefinite looping ('*') by finite looping ('k'):

a)

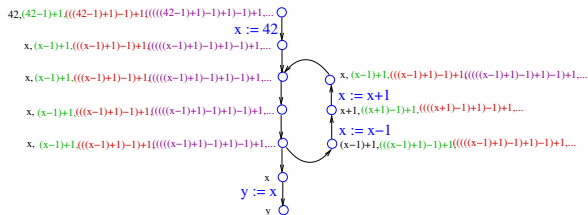


b)

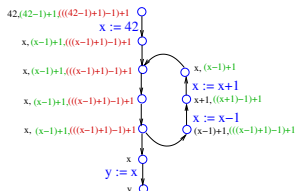


Illustrating the Limits

...of replacing indefinite looping:



by bounded looping:



Note: y is a constant but not a finite constant as defined so far; it is a finite constant of a higher level (cf. Chap. 5.8.10).

Decidability of Finite Constants (1)

Theorem 5.8.6.3 (Decidability)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, $n \in N$ a node, $\sigma_s \in \Sigma_{Init}$ an initial state, and $t \in \mathbf{T}$ a term. Then:

Algorithm 5.8.6.1 determines whether t is a finite constant at node n , i.e., whether

$$t \in C_{fin, \sigma_s}(n) = \bigcup \{C_{fin, \sigma_s}(n, d) \mid d \in ID \setminus \{\perp\}\}$$

In the positive case, Algorithm 5.8.6.1 determines also the value d of t at n .

Decidability of Finite Constants (2)

Corollary 5.8.6.4 (Decidability of Finite Constants)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, $n \in N$ a node, $\sigma_s \in \Sigma_{Init}$ an initial state, and $T_{finite} \subseteq \mathbf{T}$ a finite set of terms. Then:

$$C_{fin, \sigma_s}(n) \cap T_{finite}$$

is algorithmically decidable.

Note: The set of terms occurring in a program is finite. In particular, the set of terms occurring in an instruction at an edge are finite. Hence, the set of program terms, which are finite constants, can algorithmically be decided.

Chapter 5.8.7

Finite Constants: Specification

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

Finite Constants over \mathbb{Z} : DFA Specification

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df}$$

$$(Part, \cap, \cup, \subseteq, \{\{t\} \mid t \in \mathbf{T}\}, \{\mathbf{T}\}) = \widehat{Part}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{fc} : E \rightarrow (Part \rightarrow Part) \text{ where}$$

$$\forall e \in E. \forall p \in Part. \llbracket e \rrbracket_{fc}(p) =_{df} \{(r, s) \mid (\delta_{l_e}(r), \delta_{l_e}(s)) \in p\}$$

- ▶ Initial information: $p_{fc} \in Part_{\mathcal{E}, \sigma_s}(T_{fc, \sigma_s})$ for $\sigma_s \in \Sigma'_{Init}$ and T_{fc, σ_s} finite $\subseteq \mathbf{T}$ computed using [Algorithm 5.8.6.2](#).
- ▶ Direction of information flow: forward

Finite Constants Specification

- ▶ Specification: $S_G^{fc} = (\widehat{Part}, \llbracket \cdot \rrbracket_{fc}, p_{fc}, fw)$

Chapter 5.8.8

Termination, Safety, and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

Towards Safety and Termination

Lemma 5.8.8.1 (Descending Chain Condition)

The 'MaxFP relevant' part $\widehat{Part}_{p_{fc}}$ of \widehat{Part} satisfies the descending chain condition.

Note. The carrier set of the initial partition p_{fc} is finite.

Lemma 5.8.8.2 (Distributivity)

$\llbracket \rrbracket_{fc}$ is distributive.

Corollary 5.8.8.3 (Monotonicity)

$\llbracket \rrbracket_{fc}$ is monotonic.

Termination and Coincidence/Optimality

Theorem 5.8.8.4 (Termination)

Applied to $S_G^{fc} = (\widehat{Part}, \llbracket \rrbracket_{fc}, p_{fc}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of S_G^{fc} .

Proof. Immediately with Lemma 5.8.8.1, Lemma 5.8.8.3, Theorem 5.8.4.7, and Termination Theorem 3.4.4.

Theorem 5.8.8.5 (Coincidence/Optimality)

Applied to $S_G^{fc} = (\widehat{Part}, \llbracket \rrbracket_{fc}, p_{fc}, fw)$, Algorithm 3.4.3 is *MOP* optimal for S_G^{fc} (i.e., it terminates with the *MOP* solution of S_G^{fc}).

Proof. Immediately with Lemma 5.8.8.2, Coincidence Theorem 3.5.2, and Termination Theorem 5.8.8.4.

Chapter 5.8.9

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

Soundness and Completeness of $MOP_{S_G^{fc}}$

Theorem 5.8.9.1 (Soundness and Completeness)

The MOP solution of S_G^{fc} is

1. sound and complete for the term constant propagation problem C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^T}(n) =_{\Sigma'} v_{MOP_{S_G^{fc},\sigma_s}}(n)$$

if G is a flow graph with **acyclic control flow**.

2. sound but not complete for the term and variable constant propagation problems C^T and C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}.$$

$$v_{C_{G,\sigma_s}^T}(n) \supseteq_{\Sigma'} v_{C_{G,\sigma_s}^V}(n) \supseteq_{\Sigma'} v_{MOP_{S_G^{fc},\sigma_s}}(n)$$

if G is a flow graph with **arbitrary** (possibly cyclic) **control flow**.

Soundness and Completeness $MaxFP_{S_G^{fc}}$

Corollary 5.8.9.2 (Soundness and Completeness)

The $MaxFP$ solution of S_G^{fc} is

1. sound and complete for the term constant propagation problem C^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}. v_{C_{G,\sigma_s}^T}(n) = v_{MaxFP_{S_G^{fc},\sigma_s}}(n)$$

if G is a flow graph with **acyclic control flow**.

2. sound but not complete for the term and variable constant propagation problems C^T and C^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma'_{Init}.$$

$$v_{C_{G,\sigma_s}^T}(n) \sqsupseteq_{\Sigma'} v_{C_{G,\sigma_s}^V}(n) \sqsupseteq_{\Sigma'} v_{MaxFP_{S_G^{fc},\sigma_s}}(n)$$

if G is a flow graph with **arbitrary** (possibly cyclic) **control flow**.

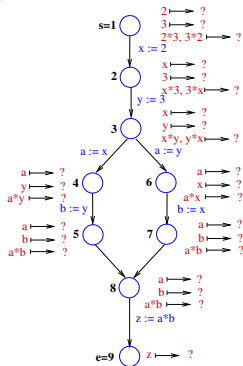
Chapter 5.8.10

Illustrating Example, Hierarchies of Classes of Finite Constants

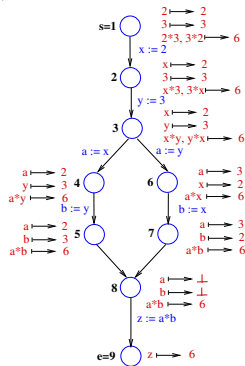
Finite Constants: Illustrating Example

...all terms are finite constants.

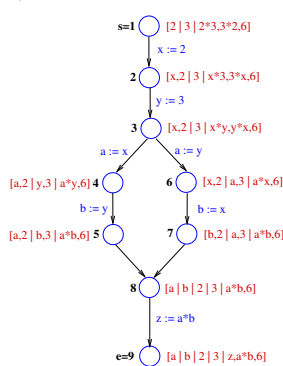
a)



b)



c)



Note: The CP algorithm for finite constants operates actually on term partitions (Fig. c)) rather than term valuations (Fig. b)).

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

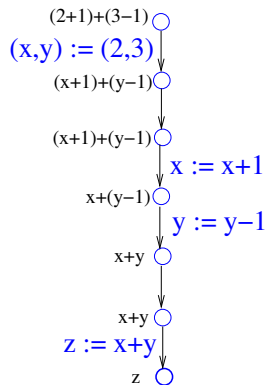
5.8.9

5.8.10

601/185

Hierarchy of Classes of Fin. Constants: Level 0

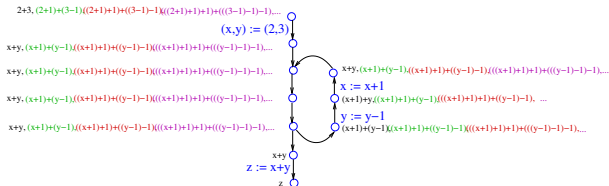
...z and $x+y$ are finite constants of level 0 using plain backward substitution (cf. Def. 5.8.3.1):



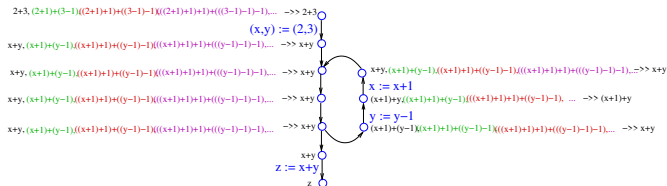
Hierarchy of Classes of Fin. Constants: Level 1

...z and x+y are **not** finite constants of level 0 (Fig. a)) **but** of level 1 (Fig. b)) using backward substitution w/ subsequent term evaluation:

a)



b)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

5.8.7

5.8.8

5.8.9

5.8.10

Hierarchy of Classes of Fin. Constants: Level 2

...z and x+y are **not** finite constants of level 0 and 1 (Fig. c)) **but** of level 2 (Fig. d)) using backward substitution w/ subsequent **symbolic term evaluation**:

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.8.1

5.8.2

5.8.3

5.8.4

5.8.5

5.8.6

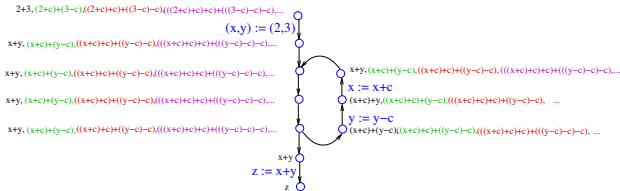
5.8.7

5.8.8

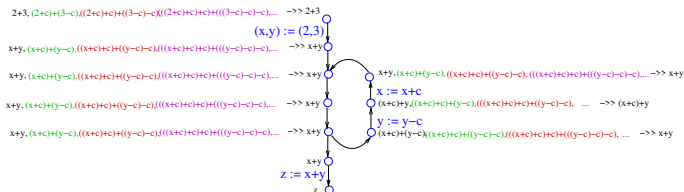
5.8.9

5.8.10

c)



d)



A Hierarchy of Classes of Finite Constants

...is induced by different choices/refinements of **backward substitution** ($\iota \equiv x := t$):

▶ **Level 0:** $\forall t' \in \mathbf{T}. \delta_{\iota}(t') =_{df} t'[t/x]$

$$\begin{aligned} \text{Bsp.: } \delta_{y:=y-1}((x+1)+y) &= (x+1) + (y-1) \\ \delta_{y:=y-c}((x+c)+y) &= (x+c) + (y-c) \end{aligned}$$

▶ **Level 1:** $\forall t' \in \mathbf{T}. \delta'_{\iota}(t') =_{df} \mathcal{E}(\delta_{\iota}(t'))(\sigma_{\perp})$

$$\begin{aligned} \text{Bsp.: } \delta'_{y:=y-1}((x+1)+y) &= x+y, \\ \delta'_{y:=y-c}((x+c)+y) &= (x+c) + (y-c) \end{aligned}$$

▶ **Level 2:** $\forall t' \in \mathbf{T}. \delta''_{\iota}(t') =_{df} \mathcal{E}_{\text{symbolic}}(\delta_{\iota}(t'))(\sigma_{\perp})$

$$\begin{aligned} \text{Bsp.: } \delta''_{y:=y-1}((x+1)+y) &= x+y \\ \delta''_{y:=y-c}((x+c)+y) &= x+y \end{aligned}$$

▶ ...

Technically

...we just need to adapt the notion of t -associated path:

Definition 5.8.10.1 (t -Associated Path)

Let $p = \langle e_1, e_2, \dots, e_q \rangle$ be a path, and let $t \in \mathbf{T}$ be a term. The t -associated path p_t for p is defined by

$$p_t = \langle (t_1, e_1), (t_2, e_2), \dots, (t_q, e_q) \rangle$$

with

$$\blacktriangleright t_q = \delta_{\iota_{e_q}}(t) \text{ and } t_j = \delta_{\iota_{e_j}}(t_{j+1}) \quad (\text{Level 0})$$

$$\blacktriangleright t_q = \delta'_{\iota_{e_q}}(t) \text{ and } t_j = \delta'_{\iota_{e_j}}(t_{j+1}) \quad (\text{Level 1})$$

$$\blacktriangleright t_q = \delta''_{\iota_{e_q}}(t) \text{ and } t_j = \delta''_{\iota_{e_j}}(t_{j+1}) \quad (\text{Level 2})$$

$\blacktriangleright \dots$

for all $1 \leq j < q$.

Note: The principle of **finiteness** in the operational/denotational view of **finite constants** is **preserved**, **not changing** at all.

Chapter 5.9

Conditional Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

607/185

Chapter 5.9.1

Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

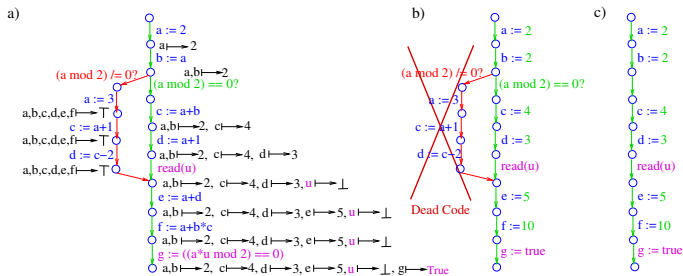
5.9.7

5.9.8

5.10

Illustrating Example

Up to now: Branches were non-deterministically interpreted.



...unfortunately, $a+8$ is not a constant, if branches are non-deterministically interpreted.

As a Matter of Fact

In the preceding example

- ▶ $a+8$ is not a (non-deterministic) constant.

Consequently

- ▶ $a+8$ is neither a simple constant nor a Q constant nor a finite constant.

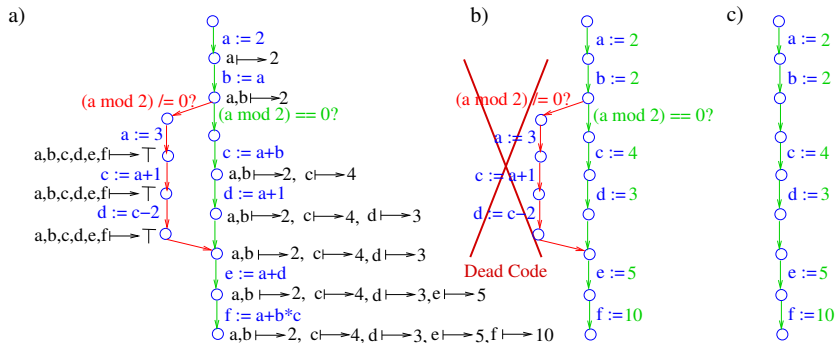
However

- ▶ $a+8$ could be detected a constant, if branching conditions were taken into account.

Note: Interpreting branches non-deterministically in DFA

- ▶ is done to avoid intricacies due to the undecidability of constant propagation,
- ▶ is counter-intuitive, however, for constant propagation itself.

Interpreting Branches Deterministically



- ▶ The term $a+8$ is a (deterministic) constant of value 10.
- ▶ Moreover, a whole program branch is detected to be dead code and can be eliminated!

Conditional Constants

Conditional constants

- ▶ are an efficiently decidable subset of the set of **deterministic constants**.
- ▶ build on and generalize the notion of **simple constants**.
- ▶ allow the **optimizations** shown in **Figure b)** and **Figure c)**, when applied to the flow graph shown in **Figure a)** of the previous slide.

Chapter 5.9.2

Preliminaries

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

613/185

From Simple to Conditional Constants

What needs to be extended and adapted?

- ▶ The notion of **terms**: Relators, logical constants and operators
- ▶ The notion of **interpretation**: Relators, logical constants and operators
- ▶ The semantics of **terms**: Boolean terms
- ▶ The semantics of **instructions**: Conditionals
- ▶ The **(basic) DFA lattice** for constant propagation: Truth values

We do not need to extend:

- ▶ The notion of **states**

since we **stay with arithmetical variables** and **do not introduce Boolean variables**.

Introducing Relators and Logical Operators

We introduce

- ▶ **LC** =_{df} {*true*, *false*}: the set of logical (or Boolean) constant symbols,
- ▶ **R** =_{df} {*==*, */=*, *<*, *>*, *<=*, *>=*, ...}: a set of binary relator symbols (or relators),
- ▶ **LO** =_{df} {*∧*, *∨*, *¬*}: the set of logical operator symbols.

We assume that

- ▶ **V**, **C**, **O** (for arithmetical terms), **LC**, **R**, and **LO** (for Boolean terms) are all pairwise disjoint.

Introducing Boolean Terms

Definition 5.9.2.1 (Boolean Terms)

1. Every constant symbol $b \in \mathbf{LC}$ is a **Boolean term**.
2. If $\rho \in \mathbf{R}$ is a binary relator and t_1 and t_2 are (arithmetical) terms, then $t_1 \rho t_2$ is a **Boolean term**.
3. if b_1 and b_2 are Boolean terms, then $b_1 \wedge b_2$, $b_1 \vee b_2$, and $\neg b_1$ are **Boolean terms**.
4. There are no **Boolean terms** other than those which can be constructed by means of the above three rules.

We denote the set of all Boolean terms by \mathbf{T}_B .

Arithmetical and Boolean Terms

We denote by

- ▶ $T_{AB} =_{df} T_A \cup T_B$ the set of all terms

where

- ▶ T_A denotes the set of arithmetical terms
- ▶ T_B denotes the set of Boolean terms.

Note: T_A equals T as introduced and used so far in Chapter 5.

Towards the Semantics of Boolean Terms

We need to extend:

- ▶ the data domain ID by adding the set of Boolean truth values $IB =_{df} \{True, False\}$,
- ▶ the interpretation from the (arithmetical) constant and operator symbols over ID to the Boolean constant symbols, relators, and logical operator symbols over IB .

Extending the Data Domain

Let

- ▶ $ID_{IB} =_{df} ID \dot{\cup} IB$ be the extended new data domain of interest.

(with ID e.g., the set of natural numbers \mathbb{N} , the set of integers \mathbb{Z} , etc.).

As before, we call the

- ▶ elements of ID_{IB} (data) values.

Moreover, we assume that

- ▶ ID_{IB} includes a distinguished element \perp representing the value *undefined* and a distinguished element \top representing 'universal' information.

Interpreting Relators and Logical Operators

Definition 5.9.2.2 (Extended Interpretation)

An **interpretation** $I =_{df} (ID_{IB}, I_0)$ of **C**, **O**, **LC**, **R**, and **LO** is a pair, where I_0 is a function, which maps every

- ▶ every constant symbol $c \in \mathbf{C}$ to a datum $I_0(c) \in ID$,
- ▶ every k -ary operator symbol $op \in \mathbf{O}$ to a total strict function $I_0(op) : ID^k \rightarrow ID$,
- ▶ the logical constant symbols *true* and *false* to the Boolean constants *True* and *False*, respectively,
- ▶ the relator symbols $\rho \in \mathbf{R} =_{df} \{==, /=, <, >, \dots\}$ to the strict relations *equal*, *not equal*, *less*, etc., on $ID \times ID$,
- ▶ the logical operator symbols \wedge , \vee , and \neg to the strict logical operations *and*, *or*, and *not* on $IB \times IB$ and IB .

and satisfies $I_0(o)(d_1, \dots, d_k) = \top$, $o \in \mathbf{O} \cup \mathbf{R} \cup \mathbf{LO}$, if $(\forall i \in \{1, \dots, k\}. d_i \neq \perp) \wedge (\exists i \in \{1, \dots, k\}. d_i = \top)$.

States, Initial States

We do not introduce **Boolean variables**. The notion of **states** thus remains essentially unchanged.

Definition 5.9.2.3 (States, Initial States over ID)

- ▶ A **state** $\sigma : \mathbf{V} \rightarrow \text{ID}$ is a total mapping, which maps every (arithmetical) variable to a data value $d \in \text{ID}$.

We denote the **set of all states** by

$$\Sigma =_{df} \{ \sigma \mid \sigma : \mathbf{V} \rightarrow \text{ID} \}$$

- ▶ σ_{\perp} and σ_{\top} denote two distinguished states of Σ defined by

$$\sigma_{\perp} = \lambda v. \perp, \quad \sigma_{\top} = \lambda v. \top$$

respectively.

- ▶ $\Sigma_{\text{Init}} =_{df} \{ \sigma \in \Sigma \mid \forall v \in \mathbf{V}. \sigma(v) \neq \top \}$ denotes the set of **initial states**.

Semantics of Terms

Definition 5.9.2.4 (Semantics of Terms)

The **semantics** of terms $t \in \mathbf{T}_{AB}$ is defined by the **evaluation function**

$$\mathcal{E} : \mathbf{T}_{AB} \rightarrow (\Sigma \rightarrow \text{ID}_{IB})$$

defined by

$$\forall t \in \mathbf{T}_{AB} \forall \sigma \in \Sigma. \mathcal{E}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ l_0(c) & \text{if } t \equiv c \in \mathbf{C} \cup \mathbf{L} \cup \mathbf{C} \\ l_0(o)(\mathcal{E}(t_1)(\sigma), \dots, \mathcal{E}(t_k)(\sigma)) & \text{if } t \equiv (o, t_1, \dots, t_k), \\ & o \in \mathbf{O} \cup \mathbf{R} \cup \mathbf{L} \cup \mathbf{O} \end{cases}$$

Semantics of Instructions

Definition 5.9.2.5 (Semantics of Instructions)

- ▶ Let $\iota \equiv x := t$, $t \in \mathbf{T}_A$, be an assignment instruction. The **semantics** of ι is defined by the **state transformation function** (or **state transformer**) $\theta_\iota : \Sigma \rightarrow \Sigma$ defined by $\theta_\iota(\sigma_T) =_{df} \sigma_T$ and

$$\forall \sigma \in \Sigma \setminus \{\sigma_T\} \forall y \in \mathbf{V}. \theta_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- ▶ Let $\iota \equiv skip$ be the empty instruction. The **semantics** of ι is defined by the **identical state transformation function** (or **state transformer**) Id_Σ , i.e., $\theta_\iota =_{df} Id_\Sigma$.

- ▶ Let $\iota \equiv t$, $t \in \mathbf{T}_B$, be a **conditional expression**. The **semantics** of ι is defined by $\theta_\iota(\sigma_T) =_{df} \sigma_T$ and

$$\forall \sigma \in \Sigma \setminus \{\sigma_T\}. \theta_\iota(\sigma)(y) =_{df} \begin{cases} \sigma_T & \text{if } \mathcal{E}(t)(\sigma) \in \{False, \top\} \\ \sigma & \text{otherwise} \end{cases}$$

State Transformer Lemma for σ_{\top}

Definition 5.9.2.5 directly yields:

Lemma 5.9.2.6 (State Transformer Lemma for σ_{\top})

Let ι be an assignment instruction, the empty statement, or a conditional expression. Then we have:

$$\theta_{\iota}(\sigma_{\top}) = \sigma_{\top}$$

Remarks on the Impact of Lemma 5.9.2.6 (1)

- ▶ Lemma 5.9.2.6 ensures that the distinguished state σ_{\top} is left **invariant** by every state transformer.
- ▶ This guarantees that the **state transformers of conditional expressions act as filters** that prevent propagating information alongside branches whose guarding conditional expressions are known to be violated (i.e., $\mathcal{E}(t)(\sigma) = \text{False}$) or can not yet be evaluated (i.e., $\mathcal{E}(t)(\sigma) = \top$).
- ▶ Conversely, the **filters** let information pass and propagate it further if the values of the guarding conditional expressions are known to be satisfied (i.e., $\mathcal{E}(t)(\sigma) = \text{True}$) or dubious (i.e., $\mathcal{E}(t)(\sigma) = \perp$).
- ▶ Overall, this causes the **semantics** to be **deterministic** and the **fixed point analysis** based on it to behave **deterministically**, too.

Remarks on the Impact of Lemma 5.9.2.6 (2)

Note:

- ▶ The case ‘can not yet be evaluated (i.e., $\theta_\iota(\sigma)(y) = \top$)’ can only occur in the actual fixed point program analysis by picking an edge carrying a conditional expression ‘too early’ from the workset.

The case can not occur and is irrelevant for the pathwise characterization of **deterministic constants** (the general problem, [Chapter 5.9.3](#)) and **conditional constants** (the computed class of constants, [Chapter 5.9.5](#)).

Extending State Transformers to Paths

Nothing has to be changed. For convenience, we recall:

Def. 5.2.6 (Ext. θ from Edges to Paths) – recalled

The state transformers θ_{ι_e} , $e \in E$, are extended onto paths $p = \langle e_1, e_2, \dots, e_q \rangle$ in G by defining:

$$\theta_p =_{df} \begin{cases} Id_{\Sigma} & \text{if } q < 1 \\ \theta_{\langle e_2, \dots, e_q \rangle} \circ \theta_{\iota_{e_1}} & \text{otherwise} \end{cases}$$

...where $G = (N, E, \mathbf{s}, \mathbf{e})$ denotes the flow graph of interest, and ι_e the instruction at edge e , $e \in E$.

Semantics of Programs: Det. Collecting Sem.

Defining the **deterministic collecting semantics** requires (only) to take care of and remove the special state σ_{\top} :

Definition 5.9.2.7 (Deterministic Collecting Sem.)

- ▶ The **deterministic collecting semantics** of G is defined by:

$$DCS_G : \Sigma_{Init} \rightarrow N \rightarrow \mathcal{P}(\Sigma)$$

$$\forall \sigma \in \Sigma_{Init}. \forall n \in N. DCS_G(\sigma, n) =_{df} \{ \theta_p(\sigma) \mid p \in \mathbf{P}[\mathbf{s}, n] \} \setminus \{ \sigma_{\top} \}$$

- ▶ The **deterministic collecting semantics** of G with respect to a fixed **initial state** $\sigma_s \in \Sigma_{Init}$ is defined by

$$DCS_{G, \sigma_s} : N \rightarrow \mathcal{P}(\Sigma)$$

$$\forall n \in N. DCS_{G, \sigma_s}(n) =_{df} \{ \theta_p(\sigma_s) \mid p \in \mathbf{P}[\mathbf{s}, n] \} \setminus \{ \sigma_{\top} \}$$

Unreachable Nodes

Note: If $\mathcal{DCS}_{G, \sigma_s}(n) = \emptyset$ for some node $n \in N$, this means:

- ▶ node n is **not reachable**, when branching conditions are deterministically interpreted.

Chapter 5.9.3

Deterministic Constant Propagation: Problem Definition

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

630/185

Chapter 5.9.3.1

The Deterministic CP Problem: Set View

Deterministic Constants

Let $\sigma_s \in \Sigma_{Init}$ be an initial state, let $t \in \mathbf{T}_{AB}$ be a term, and let $d \in \text{ID}_{IB} \setminus \{\perp, \top\}$ be a data value.

Definition 5.9.3.1.1 (Deterministic Constant)

t is a **deterministic constant** at node n for σ_s , i.e., the value of t at node n , $n \in N$, is a constant, if

$$\forall \sigma, \sigma' \in \mathcal{DCS}_{G, \sigma_s}(n). \mathcal{E}(t)(\sigma) = \mathcal{E}(t)(\sigma') \neq \perp$$

Definition 5.9.3.1.2 (Det. Constant of Value d)

t is a **deterministic constant of value d** for σ_s at node n , $n \in N$, if

$$\{\mathcal{E}(t)(\sigma) \mid \sigma \in \mathcal{DCS}_{G, \sigma_s}(n)\} = \{d\}$$

Deterministic Constants: Set View

Let $\sigma_s \in \Sigma_{Init}$ be an initial state, and let n be a node of G .

Definition 5.9.3.1.3 (Det. Const. Terms & Variables)

The set of terms and variables being **(deterministic) constants** of some value at n are given by the sets:

- ▶ $DC_{G,\sigma_s}^T(n) =_{df} \{(t, d) \in \mathbf{T}_{AB} \times \text{ID}_{IB} \mid t \text{ is a deterministic constant of value } d \text{ for } \sigma_s \text{ at } n\}$
- ▶ $DC_{G,\sigma_s}^V(n) =_{df} \{(v, d) \in \mathbf{V} \times \text{ID} \mid v \text{ is a deterministic constant of value } d \text{ for } \sigma_s \text{ at } n\}$

Deterministic Constant Propagation Problem

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, and let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$ be an initial state.

The deterministic constant propagation (CP) problems for terms and variables are defined by:

Definition 5.9.3.1.4 (Det. CP Prob. f. Terms, DC^T)

The deterministic term CP problem DC^T , is to determine for every node $n \in N$ of G the set $DC_{G, \sigma_{\mathbf{s}}}^T(n)$.

Definition 5.9.3.1.5 (Det. CP Prob. f. Var., DC^V)

The deterministic variable CP problem DC^V , is to determine for every node $n \in N$ of G the set $DC_{G, \sigma_{\mathbf{s}}}^V(n)$.

Chapter 5.9.3.2

The Deterministic CP Problem: Valuation View

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

635/185

Deterministic Constants: Valuation View

The sets DC_{G,σ_s}^T and DC_{G,σ_s}^V induce for every node n two (state-like) valuations $v_{DC_{G,\sigma_s}^T}(n)$ and $v_{DC_{G,\sigma_s}^V}(n)$, respectively, which are alternative representations of the sets DC_{G,σ_s}^T and DC_{G,σ_s}^V (as justified by Lemma 5.9.3.3.1)

► $v_{DC_{G,\sigma_s}^T} : N \rightarrow \mathbf{T}_{AB} \rightarrow \text{ID}_{IB}$ defined by

$$v_{DC_{G,\sigma_s}^T}(n)(t) =_{df} \begin{cases} d & \text{if } (t, d) \in DC_{G,\sigma_s}^T(n) \\ \perp & \text{otherwise} \end{cases}$$

► $v_{DC_{G,\sigma_s}^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$ defined by

$$v_{DC_{G,\sigma_s}^V}(n)(v) =_{df} \begin{cases} d & \text{if } (v, d) \in DC_{G,\sigma_s}^V(n) \\ \perp & \text{otherwise} \end{cases}$$

Chapter 5.9.3.3

Equivalence of Set and Valuation View

Equivalence of Set and Valuation View

We have:

Lemma 5.9.3.3.1 (Equivalence)

$\forall n \in N \forall t \in \mathbf{T}_{AB} \forall v \in \mathbf{V} \forall d \in \text{ID} \setminus \{\perp, \top\}$.

▶ $(t, d) \in DC_{G, \sigma_s}^T(n)$ iff $v_{DC_{G, \sigma_s}^T}(n)(t) = d$

▶ $(v, d) \in DC_{G, \sigma_s}^V(n)$ iff $v_{DC_{G, \sigma_s}^V}(n)(v) = d$

Equival't Charact'ion of the Det. CP Problem

The Equivalence Lemma 5.9.3.3.1 implies:

Lemma 5.9.3.3.2 (Problem Equivalence)

Solving the deterministic CP problems

- ▶ DC^T (i.e., det. CP for **terms**)
- ▶ DC^V (i.e., det. CP for **variables**)

is equivalent to computing the higher-order (valuation) functions

- ▶ $v_{DC_{G,\sigma_s}^T} : N \rightarrow \mathbf{T}_{AB} \rightarrow \text{ID}_{IB}$
- ▶ $v_{DC_{G,\sigma_s}^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$

respectively.

Chapter 5.9.3.4

The DC^T Approximation induced by DC^V

The DC^T Approximation induced by DC^V

The solution DC_{G,σ_s}^V of the det. CP problem DC^V wrt the initial state σ_s , and its induced higher-order valuation (or state) function

$$v_{DC_{G,\sigma_s}^V} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$$

induce a solution of the CP problem DC^T for both the valuation and set view of DC^T , respectively:

► Valuation-based:

$$v_{DC_{G,\sigma_s}^T} : N \rightarrow \mathbf{T}_{\text{AB}} \rightarrow \text{ID}_{\text{IB}}$$

$$v_{DC_{G,\sigma_s}^T}(n)(t) =_{df} \mathcal{E}(t)(v_{DC_{G,\sigma_s}^V}(n))$$

► Set-based:

$$DC_{DC_{G,\sigma_s}^V}^T(n) =_{df} \{(t, d) \in \mathbf{T}_{\text{AB}} \times \text{ID}_{\text{IB}} \mid \mathcal{E}(t)(v_{DC_{G,\sigma_s}^V}(n)) = d \neq \perp\}$$

$$\stackrel{\text{L. 5.9.3.4.1}}{=} \{(t, d) \in \mathbf{T}_{\text{AB}} \times \text{ID}_{\text{IB}} \mid v_{DC_{G,\sigma_s}^T}(n)(t) = d \neq \perp\}$$

Equivalence and Approximation

We have:

Lemma 5.9.3.4.1 (Equivalence)

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. \forall t \in \mathbf{T}_{AB}. \forall d \in ID. \\ (t, d) \in DC_{DC_{G, \sigma_s}^V}^T(n) \iff v_{DC_{G, \sigma_s}^V}^T(n)(t) = d$$

Lemma 5.9.3.4.2 (Approximation)

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. DC_{G, \sigma_s}^T(n) \supseteq DC_{DC_{G, \sigma_s}^V}^T(n)$$

In general, the inclusion is proper.

Interpretation, Finding (1)

Intuitively

- ▶ The [Approximation Lemma 5.9.3.4.2](#) states that a term can be a deterministic constant at some node n without that all of its variables are deterministic constants at n .

Hence

- ▶ Any sound algorithm for the det. CP problem DC^V is in general conservative and suboptimal for the det. CP problem DC^T .
- ▶ This would even hold for a (hypothetical) optimal algorithm (cf. [Theorem 5.1.1](#)) for DC^V .

Interpretation, Finding (2)

As a matter of fact

- ▶ The Undecidability Theorem 5.1.1 rules out the possibility and existence of optimal CP algorithms for DC^T and DC^V .

Hence

- ▶ The best we can hope for are conservative CP algorithms for DC^T and DC^V trading optimality for decidability (and efficiency, scalability).

Chapter 5.9.3.5

Deterministic CP Algorithms: Soundness and Completeness

Det. CP Algorithms: Soundness, Completeness

Let A^T (A^V) be a det. CP algorithm for DC^T (DC^V); let $A_{G,\sigma_s}^T(n) \subseteq \mathbf{T}_{AB} \times \text{ID}_{IB}$ ($A_{G,\sigma_s}^V(n) \subseteq \mathbf{V} \times \text{ID}$) denote the set of terms (variables) discovered by A^T (A^V) to be constant at node n .

Definition 5.9.3.5.1 (Soundness of det. CP Alg's)

A^T (A^V) is **sound** for DC^T (DC^V) if

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. DC_{G,\sigma_s}^T(n) \supseteq A_{G,\sigma_s}^T(n) \\ (DC_{G,\sigma_s}^V(n) \supseteq A_{G,\sigma_s}^V(n))$$

Definition 5.9.3.5.2 (Completeness of det. CP Alg's)

A^T (A^V) is **complete** for DC^T (DC^V) if

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. DC_{G,\sigma_s}^T(n) \subseteq A_{G,\sigma_s}^T(n) \\ (DC_{G,\sigma_s}^V(n) \subseteq A_{G,\sigma_s}^V(n))$$

Chapter 5.9.3.6

Deterministic CP Algorithms: Conservativity and Optimality

Det. CP Algorithms: Conservativity, Optimality

Definition 5.9.3.6.1 (Conservativity of det. CP Alg's)

A det. CP algorithm A^T (A^V) is **conservative** for DC^T (DC^V), if it is sound for DC^T (DC^V), i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. DC_{G, \sigma_s}^T(n) \supseteq A_{G, \sigma_s}^T(n) \\ (DC_{G, \sigma_s}^V(n) \supseteq A_{G, \sigma_s}^V(n))$$

Definition 5.9.3.6.2 (Optimality of det. CP Alg's)

A det. CP algorithm A^T (A^V) is **optimal** for DC^T (DC^V), if it is sound and complete for DC^T (DC^V), i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. DC_{G, \sigma_s}^T(n) = A_{G, \sigma_s}^T(n) \\ (DC_{G, \sigma_s}^V(n) = A_{G, \sigma_s}^V(n))$$

Chapter 5.9.3.7

Classifying Deterministic CP Algorithms

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

649/185

Classifying Det. CP Algorithms, Classifying A_{cc}

In principle, the classification scheme of algorithms for **non-deterministic** CP algorithms (cf. Chap. 5.3.7) applies to **deterministic** CP algorithms, too:

A	Number of Stored Values per Program Point	
	Singleton	Multiple
A^V	$v_{A^V, ID} : N \rightarrow \mathbf{V} \rightarrow ID$	$v_{A^V, \mathcal{P}(ID)} : N \rightarrow \mathbf{V} \rightarrow \mathcal{P}(ID)$
A^T	$v_{A^T, ID} : N \rightarrow \mathbf{T} \rightarrow ID$	$v_{A^T, \mathcal{P}(ID)} : N \rightarrow \mathbf{T} \rightarrow \mathcal{P}(ID)$

In practice, however, only a **singleton-valued** algorithm for the CP problem DC^V has been proposed computing as its result a **variable valuation** function

$$v : N \rightarrow \mathbf{V} \rightarrow ID$$

We denote this algorithm computing the class of so-called **conditional constants** by A_{cc} .

Chapter 5.9.3.8

The DC^T , DC^V Approximations of A_{cc}

DC^V, DC^T Valuation Approximations of A_{cc}

The **valuation approximation** of the **CP** problem DC^V of A_{cc} is given by the **variable valuation** function

$$\blacktriangleright v_{A_{cc}} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$$

computed by A_{cc} .

The **valuation approximation** of the **CP** problem DC^T of A_{cc} is given by the **term valuation** function

$$\blacktriangleright v_{A_{cc}}^T : N \rightarrow \mathbf{T}_{AB} \rightarrow \text{ID}_{IB}$$

induced by $v_{A_{cc}}$ and defined by:

$$\forall n \in N \forall t \in \mathbf{T}_{AB}. v_{A_{cc}}^T(n)(t) =_{df}$$

$$\begin{cases} d & \text{if } \mathcal{E}(t)(v_{A_{cc}}(n)) = d \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

The DC^V , DC^T Set Approximations of A_{cc}

The set approximations of the CP problems DC^V and DC^T of A_{cc} are induced by the variable and term valuation functions

- ▶ $v_{A_{cc}} : N \rightarrow \mathbf{V} \rightarrow \text{ID}$
- ▶ $v_{A_{cc}}^T : N \rightarrow \mathbf{T}_{AB} \rightarrow \text{ID}_{IB}$

of A_{cc} and defined by:

- ▶ $DC_{A_{cc}}^V(n) =_{df} \{(v, d) \in \mathbf{V} \times \text{ID} \mid v_{A_{cc}}(n)(v) = d \neq \perp\}$
- ▶ $DC_{A_{cc}}^T(n) =_{df} \{(t, d) \in \mathbf{T}_{AB} \times \text{ID}_{IB} \mid v_{A_{cc}}^T(n)(t) = d \neq \perp\}$

Chapter 5.9.4

DFA States, DFA Lattice

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

654/185

From Data Domains to DFA Lattices (1)

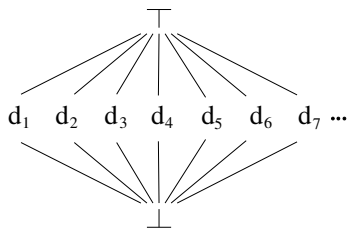
Given

- ▶ the data domain $ID_{IB} =_{df} ID \dot{\cup} IB$ of interest with ID e.g., the set of natural numbers \mathbb{N} , the set of integers \mathbb{Z} , etc., including a distinguished element \perp representing the value *undefined* and a distinguished element \top representing 'universal' information.

...it remains to arrange the elements of data domain ID_{IB} to a flat DFA lattice (as shown next).

From Data Domains to DFA Lattices (2)

Given the data domain ID_{IB} , the flat lattice $\mathcal{FL}_{ID_{IB}}$ (cf. Appendix A.4)



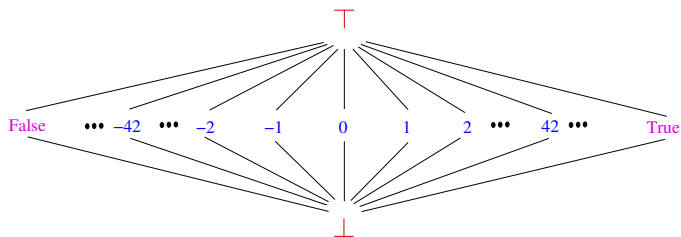
...constitutes the basic DFA lattice for conditional constant propagation.

Intuitively

- ▶ \top represents complete but inconsistent information.
- ▶ $d_i, i \geq 1$, represents precise information.
- ▶ \perp represents no information, the empty information.

The Basic DFA Lattice over \mathbb{Z} and IB

...for deterministic CP is given by $\mathcal{FL}_{\mathbb{Z}_B}$:



...leading to the class of conditional constants over \mathbb{Z} and IB.

DFA States, Semantics of Instructions, etc.

Unlike to [Chapter 5.4.1](#), we do not need to extend the notions of

- ▶ interpretation of terms
- ▶ states and initial states
- ▶ semantics of terms
- ▶ semantics of instructions

to cope with the distinguished element \top for the constant propagation analysis. This has been done to full extent in the corresponding definitions of [Chapter 5.9.2](#), which can now directly be used for defining the [conditional constants propagation analysis](#).

The DFA Lattice for Conditional Constants

The set of (DFA) states Σ together with the pointwise ordering of states, \sqsubseteq_{Σ} , constitutes a complete lattice (cf. [Appendix A.4](#)):

$$\forall \sigma, \sigma' \in \Sigma. \sigma \sqsubseteq_{\Sigma} \sigma' \text{ iff } \forall v \in \mathbf{V}. \sigma(v) \sqsubseteq_{\mathcal{FL}_{DB}} \sigma'(v)$$

Lemma 5.9.4.1 (Lattice of DFA States)

$\widehat{\Sigma} =_{df} (\Sigma, \sqcap_{\Sigma}, \sqcup_{\Sigma}, \sqsubseteq_{\Sigma}, \sigma_{\perp}, \sigma_{\top})$ is a complete lattice with

- ▶ least element σ_{\perp} , greatest element σ_{\top} ,
- ▶ pointwise meet \sqcap_{Σ} and join \sqcup_{Σ} as meet and join operation, respectively.

Chapter 5.9.5

Conditional Constants: Specification

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

660/185

Conditional Constants over \mathbb{Z}_{IB} : DFA Specific.

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\Sigma, \sqcap_{\Sigma}, \sqcup_{\Sigma}, \sqsubseteq_{\Sigma}, \sigma_{\perp}, \sigma_{\top}) = \widehat{\Sigma}$$

with Σ set of DFA states over \mathbb{Z}_{IB} .

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{cc} : E \rightarrow (\Sigma \rightarrow \Sigma) \text{ where } \forall e \in E. \llbracket e \rrbracket_{cc} =_{df} \theta_{\nu_e}$$

- ▶ Initial information: $\sigma_s \in \Sigma_{init}$
- ▶ Direction of information flow: forward

Conditional Constants Specification

- ▶ Specification: $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, \llbracket \cdot \rrbracket_{cc}, \sigma_s, fw)$

Chapter 5.9.6

Termination, Safety, and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

662/185

Towards Safety and Termination

Lemma 5.9.6.1 (Descending Chain Condition)

$\widehat{\Sigma}$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.9.6.2 (Monotonicity)

$\llbracket \cdot \rrbracket_{cc}$ is monotonic.

Lemma 5.9.6.3 (Non-Distributivity)

$\llbracket \cdot \rrbracket_{cc}$ is not distributive.

Termination and Safety/Conservativity

Theorem 5.9.6.4 (Termination)

Applied to $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, \llbracket \cdot \rrbracket_{cc}, \sigma_s, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of \mathcal{S}_G^{cc} .

Proof. Immediately with Lemma 5.9.6.1, Lemma 5.9.6.2, and Termination Theorem 3.4.4.

Theorem 5.9.6.5 (Safety/Conservativity)

Applied to $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, \llbracket \cdot \rrbracket_{cc}, \sigma_s, fw)$, Algorithm 3.4.3 is *MOP* conservative for \mathcal{S}_G^{cc} (i.e., it terminates with a lower approximation of the *MOP* solution of \mathcal{S}_G^{cc}).

Proof. Immediately with Lemma 5.9.6.2, Safety Theorem 3.5.1, and Termination Theorem 5.9.6.4.

Non-Coincidence

Theorem 5.9.6.6 (Non-Coincidence/Non-Opt.)

Applied to $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, \llbracket \rrbracket_{cc}, \sigma_s, fw)$, Algorithm 3.4.3 is in general not *MOP* optimal for \mathcal{S}_G^{cc} (i.e., it terminates with a properly lower approximation of the *MOP* solution of \mathcal{S}_G^{cc}).

Proof. Immediately with Lemma 5.9.6.3, Coincidence Theorem 3.5.2, and Termination Theorem 5.9.6.4.

Corollary 5.9.6.7 (Safety, Non-Coincidence)

The *MaxFP* solution for \mathcal{S}_G^{cc} , is always a safe approximation of the *MOP* solution of \mathcal{S}_G^{cc} . In general, the *MOP* solution and the *MaxFP* solution of \mathcal{S}_G^{cc} do not coincide.

Chapter 5.9.7

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

5.9.8

5.10

Soundness and Completeness of $MOP_{S_G^{cc}}$

Theorem 5.9.7.1 (Soundness and Completeness)

The MOP solution of S_G^{cc} is

1. sound and complete for the deterministic variable constant propagation problem DC^V , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. v_{DC_{G,\sigma_s}^V}(n) = \Sigma MOP_{S_G^{cc}, \sigma_s}(n)$$

2. sound but not complete for the deterministic term constant propagation problem DC^T , i.e.:

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. v_{DC_{G,\sigma_s}^T}(n) \sqsupseteq_{\Sigma} MOP_{S_G^{cc}, \sigma_s}^T(n)$$

In general, the inclusion is proper.

Soundness and Completeness of $MaxFP_{S_G^{cc}}$

Corollary 5.9.7.2 (Soundness and Completeness)

The $MaxFP$ solution of S_G^{cc} is

1. sound but not complete for DC^V , i.e.

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. v_{DC_G^V, \sigma_s}(n) \not\sqsupseteq_{\Sigma} MaxFP_{S_G^{cc}, \sigma_s}(n)$$

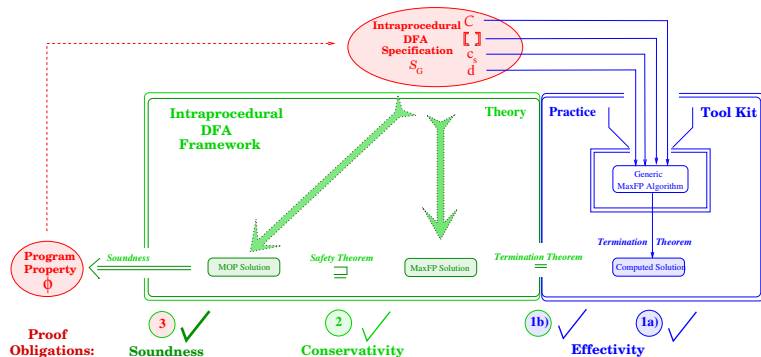
2. sound but not complete for DC^T , i.e.

$$\forall n \in N. \forall \sigma_s \in \Sigma_{Init}. v_{DC_G^T, \sigma_s}(n) \not\sqsupseteq_{\Sigma} MaxFP_{S_G^{cc}, \sigma_s}^T(n)$$

In general, both inclusions are proper.

In Summary: Soundness, *MOP*-Conservativity

...of \mathcal{S}_G^{cc} for conditional constants proven:



Chapter 5.9.8

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

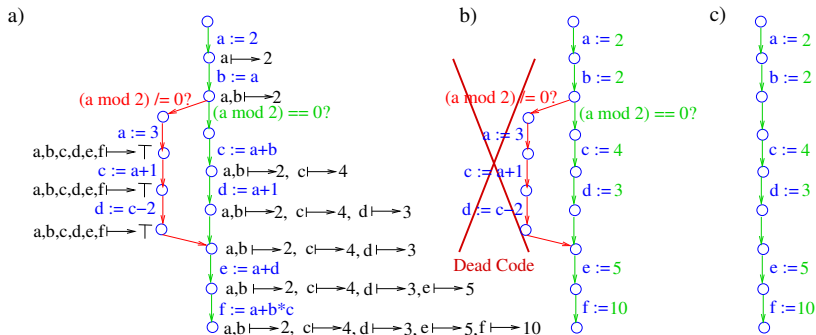
5.9.7

5.9.8

5.10

670/185

Conditional Constants over \mathbb{Z}_{IB} : Example



...all terms are conditional constants.

Remark

The original algorithm of Wegman and Zadeck for conditional constants is technically different and makes use of

- ▶ executable flags for nodes/edges

in order to filter information for propagation.

Going Beyond Conditional Constants

...is possible.

Conditional expressions like

- ▶ $x==1?$ can be treated like an assignment on the 'true' branch, even if their truth value can not be decided at analysis time.
- ▶ $x>0?$ can also be propagated along the 'true' branch and beneficially be exploited for evaluating other program terms; similarly, this holds for the negation of this expression $x\leq 0?$ along the 'false' branch.
- ▶ $(a \bmod 2)==0?$ can also be propagated along the 'true' branch and beneficially be exploited for evaluating other program terms.
- ▶ ...

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.9.1

5.9.2

5.9.3

5.9.4

5.9.5

5.9.6

5.9.7

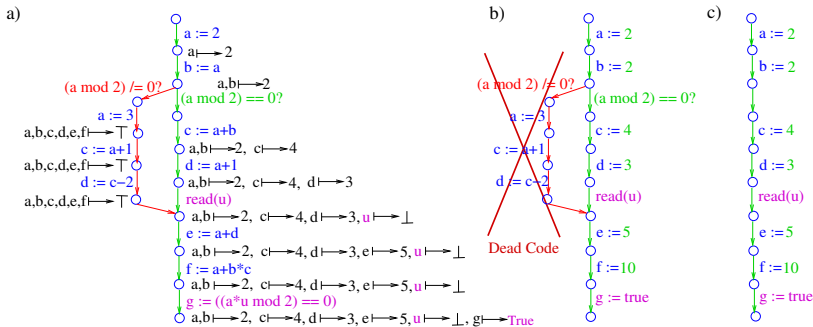
5.9.8

5.10

673/185

Beyond Conditional Constants over \mathbb{Z}_{IB}

Heuristic extensions of the conditional constants analysis as sketched before would allow to detect that the (Boolean) variable g is a constant of value **True**, even though the value of variable u can not be figured out at compile time.



Chapter 5.10

VG Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

5.12

Chap. 6

Chapter 5.10.1

Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

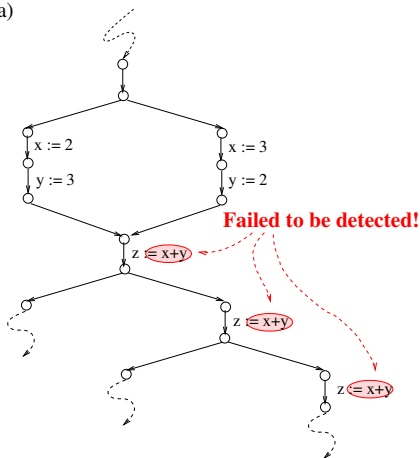
5.12

Chap. 6

Simple Constants and Q Constants

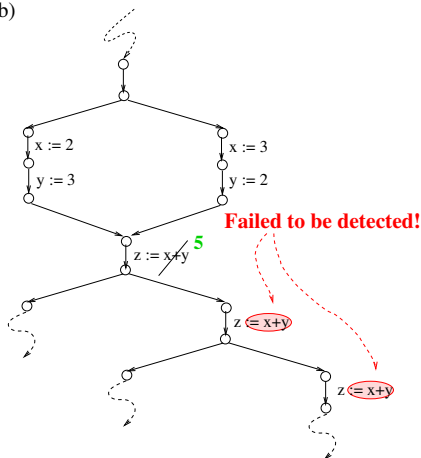
Recalling the limitations of **simple** and **Q constants**:

a)



After simple constant propagation
(No effect at all!)

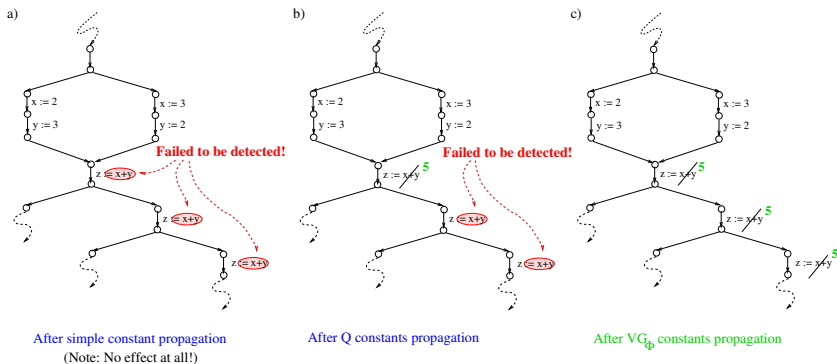
b)



After Q constants propagation

The Look-Ahead Challenge

...there is a need for a **look-ahead** of **unlimited length**:



Constant Propagation on the Value Graph (1)

In this chapter, we are going to present

- ▶ the VG_ϕ algorithm addressing this challenge.

Like the algorithm for **finite constants**, the VG_ϕ algorithm

- ▶ extends the **look-ahead of 1 heuristics** of the **Q approach** systematically.

Compared to the algorithm for **finite constants**, however,

- ▶ the VG_ϕ algorithm balances **analysis power** and **computational complexity** differently giving more weight to **performance**.

Constant Propagation on the Value Graph (2)

...comes in two variants:

- ▶ The VG_{sc} Approach: The basic algorithm
...computes simple constants.
- ▶ The VG_{ϕ} Approach: The full algorithm
...mimicks the look-ahead heuristics for a finite but long range analysis.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

5.12

Chap. 6

680/185

Constant Propagation on the Value Graph (3)

Technically, both the basic VG_{sc} and the full VG_{ϕ} algorithm

- ▶ work on the value graph of Alpern, Wegman, and Zadeck (POPL'88) of a program that is derived from the static single assignment (SSA) representation of a program.

and proceed in 5 steps:

- ▶ Construct the static single assignment form (SSA) form G_{ssa} of a program G .
- ▶ Construct the value graph (VG) of G_{ssa} .
- ▶ Analyse the value graph to detect constant terms.
- ▶ Apply the analysis results to optimize the SSA form of G_{ssa} of G , obtaining $G_{ssa_{opt}}$.
- ▶ Construct the optimized flow graph G_{opt} from $G_{ssa_{opt}}$.

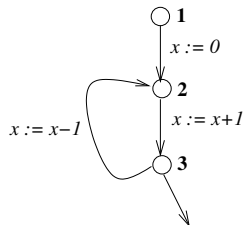
Chapter 5.10.2

$V_{G_{SC}}$ Constants: The Basic Approach

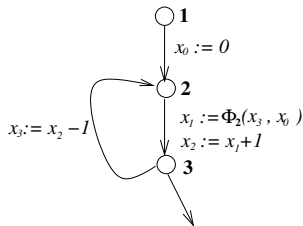
The Running Example

...constructing the SSA form and the VG of a program:

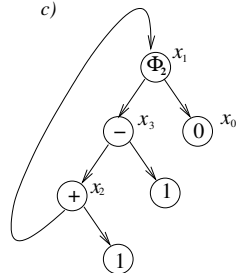
a)



b)



c)



Original Flow Graph

SSA Form

Value Graph

Defining the Value Graph Formally (1)

Definition 5.10.2.1 (Value Graph)

Let G_{SSA} be the SSA form of a flow graph G . The **value graph** $VG_{G_{SSA}} = (V, L, A)$ is a triple, where V is a set of vertices, L a labelling function of vertices, and A a set of directed edges (or arcs).

- ▶ **Vertices:** For every assignment of G_{SSA} with a nontrivial right-hand side term t (i.e., t contains at least one operator), V contains an **operator vertex**; for every occurrence of a constant in G_{SSA} , V contains a **constant vertex**.⁴

⁴For the sake of simplicity we assume that all variables are initialized.

Defining the Value Graph Formally (2)

- ▶ **Labels:** Every vertex of V is labelled with the operator or the constant of its underlying right-hand side term or constant occurrence, respectively, and the set of variables whose value is generated by the corresponding assignment or constant.
- ▶ **Arcs:** Every operator vertex of V has for every of its operands an outgoing arc pointing to the vertex of V labelled with this operand.

Every arc is labelled with a natural number denoting the position of the operand that it points to in the term it is an operand of.⁵

⁵This labelling is omitted in the examples; we assume that edges are implicitly ordered from left to right.

Value Graphs: A few Remarks

Let VG be a value graph. By construction, it is ensured that

- ▶ operator nodes of VG are always annotated with the left-hand side variable of their underlying assignment statement, also called the **generating assignment**.
- ▶ The left-hand side variable x of a trivial assignment
 - ▶ $x := y$, $y \in \mathbf{V}$, is attached to the vertex corresponding to the generating assignment of y .
 - ▶ $x := c$, $c \in \mathbf{C}$, is attached to the constant vertex corresponding to the occurrence of the constant c .

For convenience, constant and operator annotations are written inside the circle visualizing a vertex, variable annotations outside.⁶

⁶For simplicity, we assume that ordinary term operators and ϕ operators are all binary (extensions to k -ary operators are straightforward).

The VG_{sc} Algorithm

Algorithm 5.10.2.2 (Computing VG_{sc} Constants)

Let $VG = (V, L, A)$ be a value graph. Then:

Initialization Step: For every vertex $v \in V$ initialize:

$$\text{dfi}[v] = \begin{cases} I_0(c) & \text{if } v \text{ is a leaf node of } VG \text{ labelled by } c \\ \top & \text{otherwise} \end{cases}$$

Iteration Step:

1. For every vertex $v \in V$ labelled by an ord. operator op :

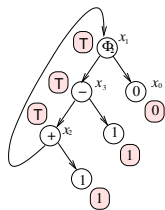
$$\text{dfi}[v] = I_0(op)(\text{dfi}[l(v)], \text{dfi}[r(v)]) \quad (\text{Evaluating terms})$$

2. For every vertex $v \in V$ labelled by a ϕ operator:

$$\text{dfi}[v] = \text{dfi}[l(v)] \sqcap \text{dfi}[r(v)] \quad (\text{Merging DFA-info's at join nodes})$$

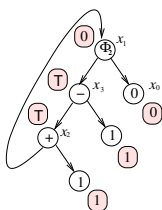
Running Example: Illustrating the VG_{SC} Alg.

a) Initialization Step

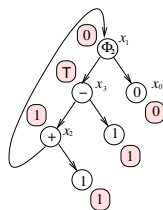


After the initialization step

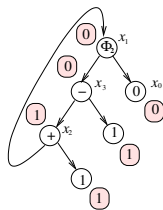
b) Iteration Steps



After the 1st iteration step



After the 2nd iteration step

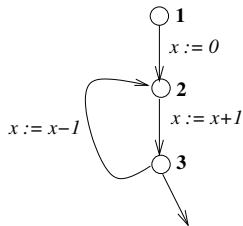


After the 3rd iteration step: Stable!

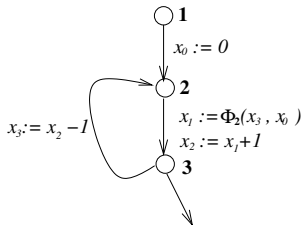
Analysis result: x_0 , x_1 , x_2 , and x_3 are VG_{SC} constants!

Running Example: The VG_{SC} Optimization

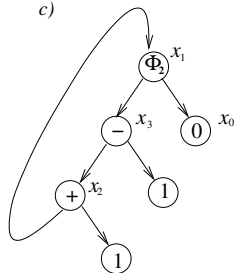
a)



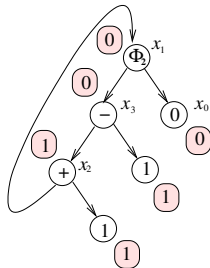
b)



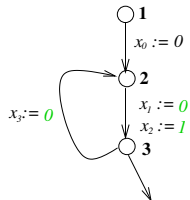
c)



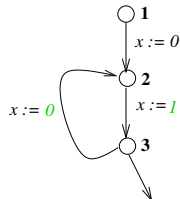
d)



e)



f)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

5.12

Chap. 6

Chapter 5.10.3

VG_{ϕ} Constants: The Full Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

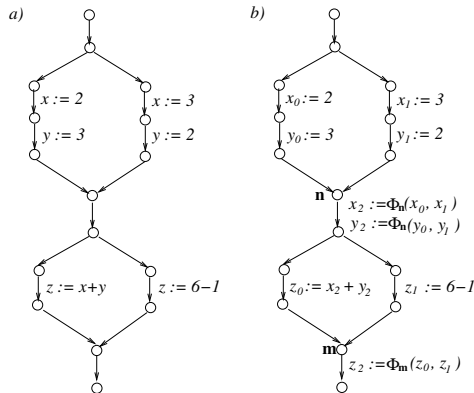
5.11

5.12

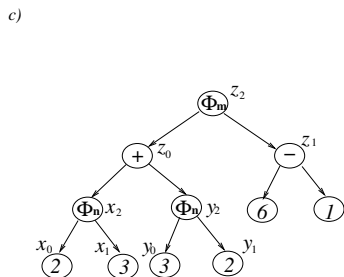
Chap. 6

The Running Example

...constructing the **SSA form** and the **VG** of a program:



Original Flow Graph SSA Form



Value Graph

Extending the Data Domain: ϕ Constants

Intuitively, ϕ constants are expressions which are composed of constants and ϕ operators.

Definition 5.10.3.1 (ϕ Constants over ID')

The set ID^ϕ of ϕ constants over ID' is inductively defined as the smallest set satisfying:

1. $ID' \subseteq ID^\phi$
2. If ϕ_n is a ϕ operator occurring in the SSA form and $d_1, d_2 \in ID^\phi$ such that neither d_1 nor d_2 contains ϕ_n , then $\phi_n(d_1, d_2) \in ID^\phi$.

Note that ID' (cf. Chapter 5.4) and hence ID^ϕ contain the distinguished elements \perp and \top .

The Complete Partial Order of ϕ Constants

Lemma 5.10.3.2 (CPO of ϕ Constants)

The pair (ID^ϕ, \sqsubseteq) , where \sqsubseteq is defined by

$$\phi_n(r_1, r_2) \sqsubseteq r \iff_{df}$$

$$(r_1 \sqsubseteq r \vee r_2 \sqsubseteq r) \vee (r = \phi_n(r_3, r_4) \wedge r_1 \sqsubseteq r_3 \wedge r_2 \sqsubseteq r_4)$$

is a **complete partial order**.

Note: (ID^ϕ, \sqsubseteq) is not a lattice since greatest lower bounds do not exist. E.g., $\phi_n(2, 3)$ and $\phi_n(3, 2)$ are incomparable lower bounds of 2 and 3, respectively.

The Evaluation Function \mathcal{E}^+

Definition 5.10.3.3 (Evaluation Function \mathcal{E}^+)

The evaluation function $\mathcal{E}^+ : V_{L(C, Op)} \rightarrow ID^\phi$ is defined by:

1. $\mathcal{E}^+(d) =_{df} d$ if $d \in ID'$

2.

$$\mathcal{E}^+(\phi_n(r_1, r_2)) =_{df} \begin{cases} \perp & \text{if } \mathcal{E}^+(r_1) \text{ or } \mathcal{E}^+(r_2) \\ & \text{contains } \phi_n \\ \mathcal{E}^+(r_1) & \text{if } \mathcal{E}^+(r_1) \sqsubseteq \mathcal{E}^+(r_2) \\ \mathcal{E}^+(r_2) & \text{if } \mathcal{E}^+(r_2) \sqsubseteq \mathcal{E}^+(r_1) \\ \phi_n(\mathcal{E}^+(r_1), \mathcal{E}^+(r_2)) & \text{otherwise} \end{cases}$$

3. $\mathcal{E}^+(op(r_1, r_2)) =_{df}$

$$\begin{cases} I_0(op)(r_1, r_2) & \text{if } r_1, r_2 \in ID' \\ \mathcal{E}^+(\phi_n(op(r_1, r_{21}), op(r_1, r_{22}))) & \text{if } r_1 \in ID', r_2 = \phi_n(r_{21}, r_{22}) \\ \mathcal{E}^+(\phi_n(op(r_{11}, r_2), op(r_{12}, r_2))) & \text{if } r_1 = \phi_n(r_{11}, r_{12}), r_2 \in ID' \\ \mathcal{E}^+(\phi_n(op(r_{11}, r_{21}), op(r_{12}, r_{22}))) & \text{if } r_1 = \phi_n(r_{11}, r_{12}), \\ & r_2 = \phi_n(r_{21}, r_{22}) \\ \perp & \text{otherwise} \end{cases}$$

Discussing \mathcal{E}^+ : Intuition

Intuitively

- ▶ the evaluation function \mathcal{E}^+ maps vertices of the value graph depending on the operator or constant symbol they are annotated with ('inside the circle') to a ϕ constant in ID^ϕ .

Discussing \mathcal{E}^+ : Power and Performance

Controlling analysis power:

- ▶ Def. 5.10.3.3, second item, the 'otherwise' case: Here, ϕ constants are constructed, which, as operands of ordinary operators, are evaluated in a distributive fashion (cf. lines two to four of the third item, $\phi_n(\mathcal{E}^+(\dots))$ and $\mathcal{E}^+(\phi_n\dots)$)

Controlling performance:

- ▶ Def. 5.10.3.3, third item, the 'otherwise' case: The evaluation of \mathcal{E}^+ yields \perp , if r_1 and r_2 are ϕ constants with different top level ϕ operators, i.e., origin from different join nodes in the program. This is in order to avoid the combinatoric explosion which reflects the co-NP-hardness of constant propagation on (even) acyclic programs.

The VG_ϕ Algorithm

Algorithm 5.10.3.4 (Computing VG_ϕ Constants)

Let $VG = (V, L, A)$ be a value graph. Then:

Initialization Step: For every vertex $v \in V$ initialize:

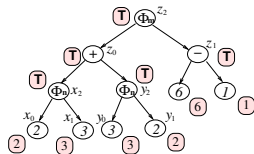
$$\text{dfi}[v] = \begin{cases} \mathcal{E}^+(\text{lab}[v]) & \text{if } v \text{ is a leaf node of } VG \\ \top & \text{otherwise} \end{cases}$$

Iteration Step: For every vertex $v \in V$ labelled by an ordinary or ϕ operator op :

$$\text{dfi}[v] = \mathcal{E}^+(op(\text{dfi}[l(v)], \text{dfi}[r(v)]))$$

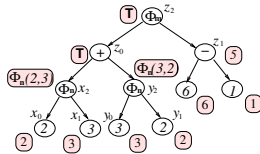
Running Example: Illustrating the VG_ϕ Alg.

a) Initialization Step

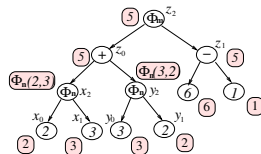


After the initialization step

b) Iteration Steps



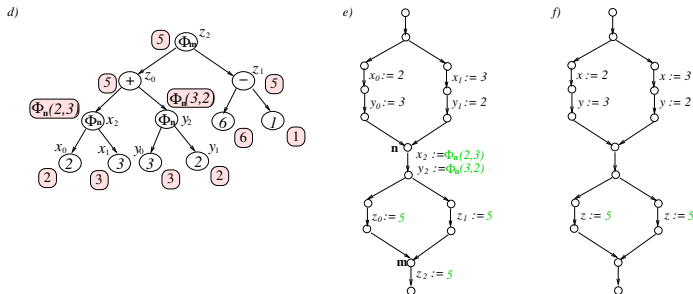
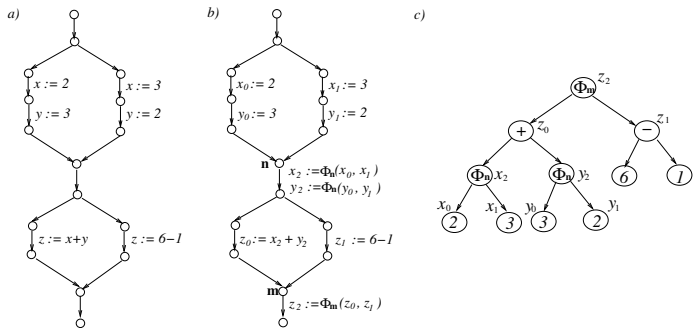
After the 1st iteration step



After the 2nd iteration step: Stable!

Analysis result: x_0 , x_1 , y_0 , y_1 , z_0 , z_1 , and z_2 are VG_ϕ constants!

Running Example: The VG_ϕ Optimization



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

5.12

Chap. 6

699 / 185

The Key

...to obtaining this result:

- ▶ Introducing ϕ constants.
- ▶ Extending the eval. function on VGs to ϕ constants, \mathcal{E}^+ .
- ▶ Defining \mathcal{E}^+ to carefully balancing power and computational complexity when evaluating it.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

5.12

Chap. 6

700/185

Chapter 5.10.4

Main Results

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

5.12

Chap. 6

701/185

Main Result for VG_{sc} and VG_{ϕ}

Theorem 5.10.4.1 (VG_{sc} Theorem)

The VG_{sc} algorithm computes the class of **simple constants**.

Theorem 5.10.4.2 (VG_{ϕ} Theorem)

The VG_{ϕ} algorithm computes

1. a **superset** of the set of **simple constants** for programs with **unrestricted control flow**.
2. the class of **injective constants**, i.e., the class of constants composed of operators, which are injective for the relevant term operands, for programs with **acyclic control flow**.

Injective Operators

Informally, an operator *op* is called

- ▶ *injective*, if the equality of the values of *op* applied to two arrays of operands implies equality of corresponding operands of these two arrays of operands.

Examples: The operators

- ▶ *plus* and *minus* are not injective.
- ▶ *square* and *abs* are not injective over \mathbb{Z} but injective over \mathbb{N} .
- ▶ *increment*, *decrement*, *plus c*, *minus c*, *c* a constant, are injective.
- ▶ ...

Overall

...the VG_ϕ algorithm

- ▶ keeps a fine balance between power and performance
- ▶ achieves a finite but long range look-ahead.

The value graph and the SSA form of a program it is derived from are fundamental for this achievement.

Note: Beyond its usage for constant propagation on the value graph in this chapter, the SSA form of a program is

- ▶ a most widely used intermediate program representation in optimizing compilers.

The SSA form of a program is attractive because

- ▶ lexical identical terms are ensured to be semantically equivalent, i.e., to always yield the same value, which is important to know for many analyses and optimizations.

Chapter 5.10.5

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

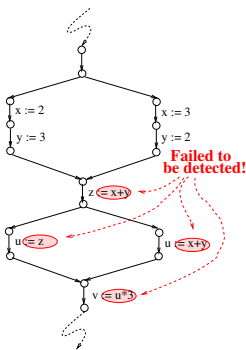
5.11

5.12

Chap. 6

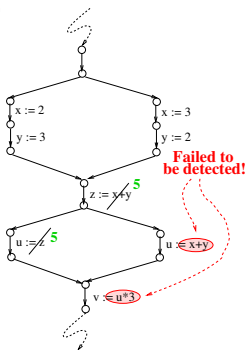
VG_{ϕ} Constants over \mathbb{Z} : Illustrating Example

a)



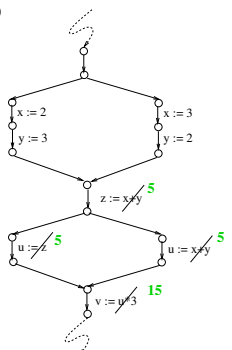
After simple constants propagation
(No effect at all!)

b)



After Q constants propagation

c)



After VG_{ϕ} constants propagation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.10.1

5.10.2

5.10.3

5.10.4

5.10.5

5.11

5.12

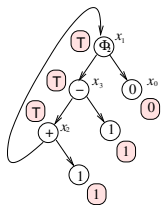
Chap. 6

706/185

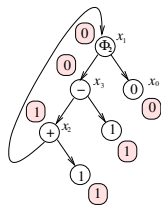
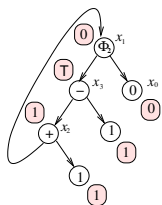
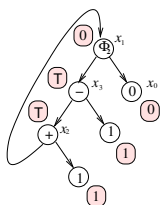
Constant Propagation on the Value Graph

...receives **Triple E** Rating: **E**xpressive, **E**fficient, **E**asy!

a) Initialization Step



b) Iteration Steps



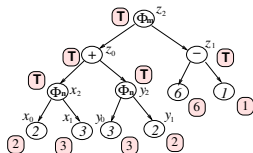
After the initialization step

After the 1st iteration step

After the 2nd iteration step

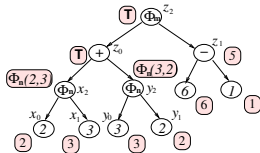
After the 3rd iteration step: Stable!

a) Initialization Step

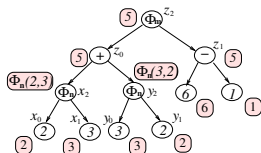


After the initialization step

b) Iteration Steps



After the 1st iteration step



After the 2nd iteration step: Stable!

Chapter 5.11

Summary, Looking Ahead

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

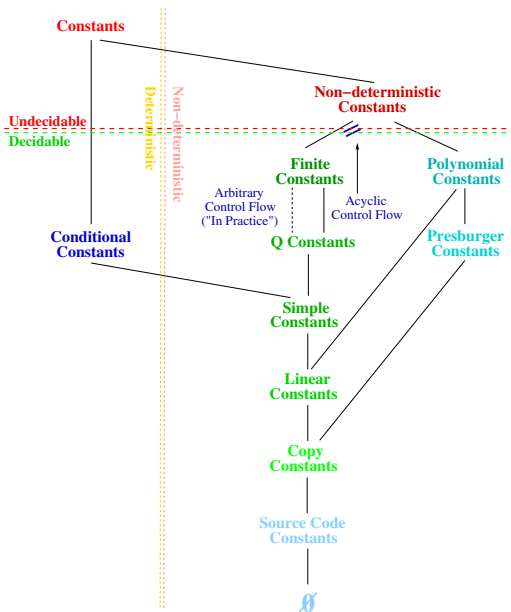
708/185

Summary

The undecidability of the general constant propagation problem inspired a quest for decidable and for efficiently decidable classes of the constant propagation problem having led (among others) to:

- ▶ Simple constants
- ▶ Linear constants
- ▶ Copy constants
- ▶ Q constants
- ▶ Conditional constants
- ▶ Finite constants
- ▶ VG_ϕ constants
- ▶ Presburger constants
- ▶ Polynomial constants
- ▶ ...

The Lattice of Constant Propagation Classes



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

710/185

Design Strategies and Achievements

- ▶ Trading generality and precision for efficiency, scalability
 - ▶ Simple constants (default for intraprocedural CP)
 - ▶ Linear constants (relevant for interprocedural CP)
 - ▶ Copy constants (relevant for interprocedural CP)
 - ▶ Q constants (modest improvement over simple const.)
 - ▶ Conditional constants (improvement over simple constants by branch evaluation, towards deterministic CP)
- ▶ Trading generality, efficiency, and scalability for precision
 - ▶ Finite constants (arbitrary term operators, decidable for arbitrary control flow, complete for acyclic control flow, intraprocedural)
 - ▶ Presburger constants (+, -: decidable and complete for arbitrary control flow, intraprocedural)
 - ▶ Polynomial constants (+, -, *: decidable and complete for arbitrary control flow, intraprocedural)

The Challenges of Constant Propagation

...are nicely illustrated by an example of Markus Müller-Olm and Helmut Seidl (SAS 2002): z at node 4 is a polynomial constant of value 0 but it is not a simple/Q/finite/conditional constant .

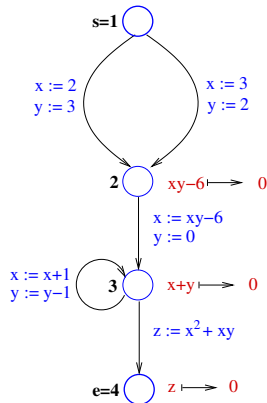


Table 2. Complexity classification of a taxonomy of CP: summarizing the results.

		Must-Constants	May-Constants	
			single value	multiple value
acyclic control flow	Copy Constants	green	green	red
	Linear Constants	green	green	red
	Pressburger Constants	green	red	red
	+,-,* Constants	red	red	red
	Full Constants	red	red	red
unrestricted control flow	Copy Constants	green	green	PSPACE-compl.
	Linear Constants	green	NP-hard	black
	Presburger Constants	green	black	black
	+,-,* Constants	PSPACE-hard	black	black
	Full Constants	black	black	black

Co-NP compl. (diagonal text in red cell)

NP complete (diagonal text in red cell)

undecidable (diagonal text in black cell)

Constant Propagat'n: More than a Commodity

Constant propagation is

- ▶ among the most important and most widely used optimizations of classical optimization.
- ▶ indispensable for designing and engineering safety-critical real-time systems, (e.g., for worst-case execution time analysis (loop bounds computation, recursion depths analysis, etc.) of such systems.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

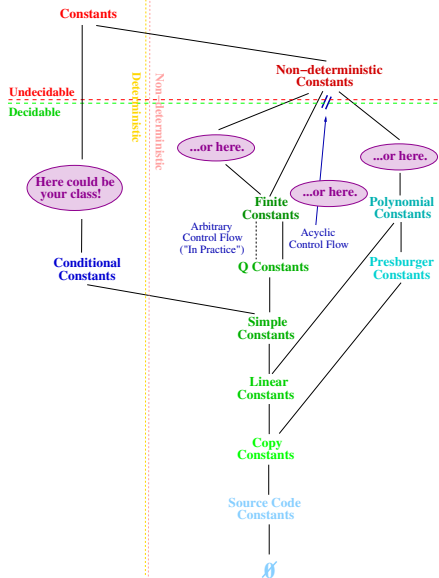
Chap. 7

Chap. 8

Chap. 9

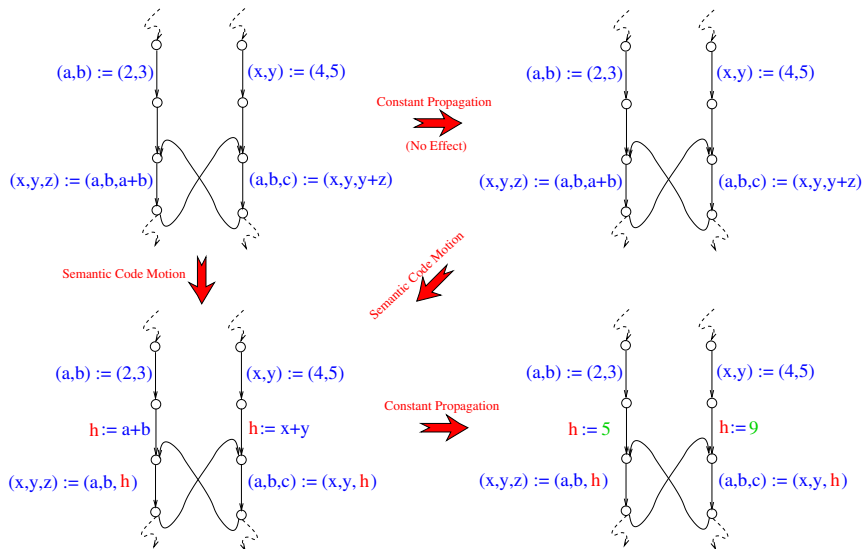
Looking Ahead: Constant Propagation

...a subject of challenging Master (and Bachelor) theses.



Looking Ahead: Topics for Theses (1)

...combining analyses and optimizations, implement, evaluate:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

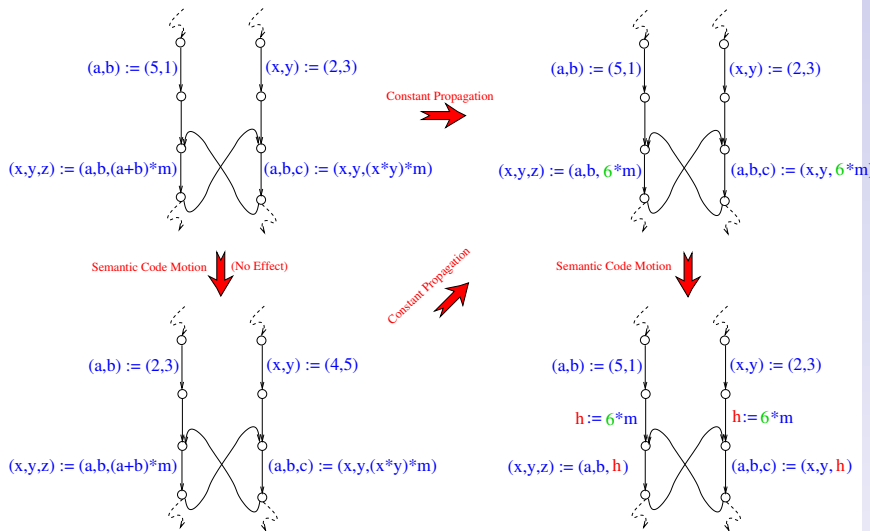
Chap. 7

Chap. 8

716/185

Looking Ahead: Topics for Theses (2)

...combining analyses and optimizations, implement, evaluate:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

717/185

Chapter 5.12

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6




Chap. 7

Chap. 8




718/185

Further Reading for Chapter 5 (1)

Constant Propagation in Textbooks

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 8.5.4, The Use of Algebraic Identities; Chapter 9.4, Constant Propagation)
-  Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 4.4.3, Constant Propagation)
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 1, Overview of Compilation; Chapter 8, Introduction to Code Optimization; Chapter 9, Data Flow Analysis)

Further Reading for Chapter 5 (2)

-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 4.2.3, Constant Propagation; Chapter 4.2.4, Variants of Constant Propagation - Conditional Constant Propagation, Copy Constant Propagation)
-  Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 8.3, Constant Propagation)
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 8.14, More Ambitious Analyses)

Further Reading for Chapter 5 (3)

Pioneering Works on Constant Propagation

Simple Constants



-  Gary A. Kildall. *A Unified Approach to Global Program Optimization*. In Conference Record of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73), 194-206, 1973.

Simple Constants and (Slightly) Beyond: Q Constants

-  John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. Acta Informatica 7:305-317, 1977.


Further Reading for Chapter 5 (4)

Undecidability of Constant Propagation, Constant Propagation on the Global Value Graph

-  John H. Reif, Harry R. Lewis. *Symbolic Evaluation and the Global Value Graph*. In Conference Record of the 4th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), 104-118, 1977.
-  John H. Reif, Harry R. Lewis. *Efficient Symbolic Analysis of Programs*. Aiken Computation Laboratory, Harvard University, TR-37-82, 1982.



Further Reading for Chapter 5 (5)

Complexity of Constant Propagation


-  Markus Müller-Olm. *The Complexity of Copy Constant Detection in Parallel Programs*. In Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2001), Springer-V., LNCS 2010, 490-501, 2001.
-  Markus Müller-Olm. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*. Springer-V., LNCS 3800, 2006.
-  Markus Müller-Olm, Oliver Rüthing. *The Complexity of Constant Propagation*. In Proceedings of the European Symposium on Programming (ESOP 2001), Springer-V., LNCS 2028, 190-205, 2001.

Further Reading for Chapter 5 (6)

Conditional Constants




-  Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. In Conference Record of the 12th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'85), 291-299, 1985.
-  Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. ACM Transactions on Programming Languages and Systems 13(2):181-210, 1991.

Linear Constants

-  Mooly Sagiv, Tom Reps, Susan Horwitz. *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation*. Theoretical Computer Science 167(1-2):131-170, 1996.




Further Reading for Chapter 5 (7)

Polynomial Constants and Finite Constants


-  Markus Müller-Olm, Helmut Seidl. *Polynomial Constants are Decidable*. In Proceedings of the 9th Static Analysis Symposium (SAS 2002), Springer-V., LNCS 2477, 4-19, 2002.
-  Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. In Proceedings of the 14th International Conference on Mathematical Foundations of Computer Science (MFCS'89), Springer-V., LNCS 379, 481-490, 1989.
-  Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. Theoretical Computer Science 80(2):303-318, 1991.

Further Reading for Chapter 5 (8)

Constant Propagation in Specific Settings

-  Kathleen Knobe, Vivek Sarkar. *Conditional Constant Propagation of Scalar and Array References Using Array SSA Form*. In Proceedings of the 5th Static Analysis Symposium (SAS'98), Springer-V., LNCS 1503, 33-56, 1998.
-  Jens Knoop. *Parallel Constant Propagation*. In Proceedings of the 4th European Conference on Parallel Processing (Euro-Par'98), Springer-V., LNCS 1470, 445-455, 1998.
-  Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. *Journal of Universal Computer Science* 9(8):829-850, 2003.

Further Reading for Chapter 5 (9)

-  Samuel P. Midkiff, José E. Moreira, Marc Snir. *A Constant Propagation Algorithm for Explicitly Parallel Programs*. International Journal of Computer Science 26(5):563-589, 1998.


Constant Propagation on the Value Graph

-  Jens Knoop, Oliver Rüthing. *Constant Propagation on the Value Graph: Simple Constants and Beyond*. In Proceedings of the 9th International Conference on Compiler Construction (CC 2000), Springer-V., LNCS 1781, 94-109, 2000.
-  Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. Journal of Universal Computer Science 9(8):829-850, 2003. (Special issue for SBLP'03).


Further Reading for Chapter 5 (10)

 Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. In Proceedings of the 7th Brazilian Symposium on Programming Languages (SBLP 2003), 135-148, 2003.


Constructing SSA Form

 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Ken Zadeck. *An Efficient Method of Computing Static Single Assignment Form*. In Conference Record of the 16th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89), 25-35, 1989.

Further Reading for Chapter 5 (11)




-  Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Ken Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. ACM Transactions on Programming Languages and Systems 13(4):451-490, 1991.

Constructing the Value Graph




-  Bowen Alpern, Mark N. Wegman, F. Ken Zadeck. *Detecting Equality of Variables in Programs*. In Conference Record of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 1-11, 1988.

Further Reading for Chapter 5 (12)



Miscellaneous

-  Martin Davis. *Hilbert's Tenth Problem is Unsolvable*. American Mathematical Monthly 80:33-269, 1973.
-  Martin Davis, Yuri Matijasevič, Julia Robinson. *Hilbert's Tenth Problem. Diophantine Equations: Positive Aspects of a Negative Solution*. In Proceedings of the Symposium on the Hilbert Problems (De Kalb, Illinois), May 1974, American Mathematical Society, Providence, R.I., 323-378, 1976.
-  William Landi. *Undecidability of Static Analysis*. ACM Letters on Programming Languages and Systems 1(4):323-337, 1992.

Further Reading for Chapter 5 (13)

-  Yuri V. Matijasevič. *Enumerable Sets are Diophantine (in Russian)*. Doklady Akademii Nauk SSSR 191:279-282, 1970 (english translation: Soviet Mathematics Doklady 11:354-357, 1970).
-  Yuri V. Matijasevič. *On Recursive Unsolvability of Hilbert's Tenth Problem*. In Proceedings of the 4th International Congress on Logic, Methodology and Philosophy of Science (Bucharest 1971), North-Holland, Amsterdam, 89-110, 1973.
-  Yuri V. Matijasevič. *What Should We Do Having Proved a Decision Problem to be Unsolvable?* In Proceedings of Algorithms in Modern Mathematics and Computer Science, Springer-V., LNCS 122, 441-448, 1979.

Further Reading for Chapter 5 (14)

-  Yuri V. Matijasevič. *Hilbert's Tenth Problem*. MIT Press, 1993.
-  Robert E. Tarjan. *Fast Algorithms for Solving Path Problems*. *Journal of the ACM* 28(3):594-614, 1981.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

Chap. 6

Chap. 7

Chap. 8

732/185

Chapter 6

Partial Redundancy Elimination

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

733/185

Chapter 6.1

Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

734/185

Motivation: Looking Back and Ahead

Looking back

- ▶ Classical Gen/Kill Data Flow Analyses
 - ▶ Focus: Proving soundness and completeness of DFAs for selected program properties (availability, liveness, very busyness, etc.); no program optimization involved (cf. Chapter 2, 3, and 4).
- ▶ Constant Propagation
 - ▶ Focus: Proving soundness and (relative) completeness of DFAs for non-deterministic and deterministic constant propagation (SCs, LCs, CpCs, QCs, FCs, CCs); program optimizations involved but trivial (cf. Chapter 3 and 5).

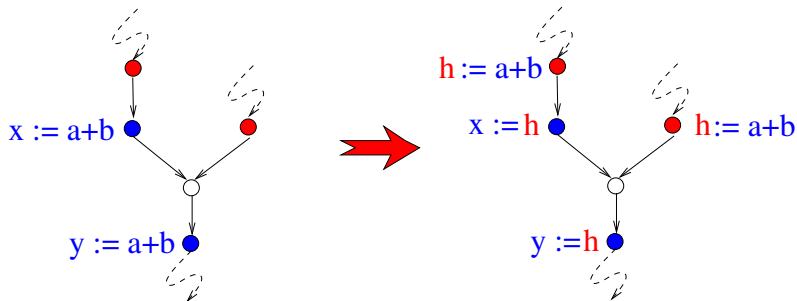
Looking ahead

- ▶ Partial Redundancy Elimination
 - ▶ Focus: Proving optimality of several non-trivial program optimizations (busy code motion, lazy code motion, sparse code motion) (cf. Chapter 7, 8, and 9).

Partial Redundancy Elimination (PRE)

What's it all about?

...avoiding multiple (re-) computations of the same value!



Chapter 6.2

PRE: Essence and Objectives

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

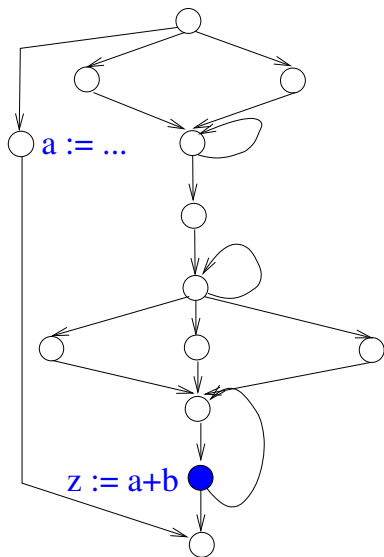
Part III

Chap. 11

Chap. 12

737/185

PRE – Particularly Powerful for Loops



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

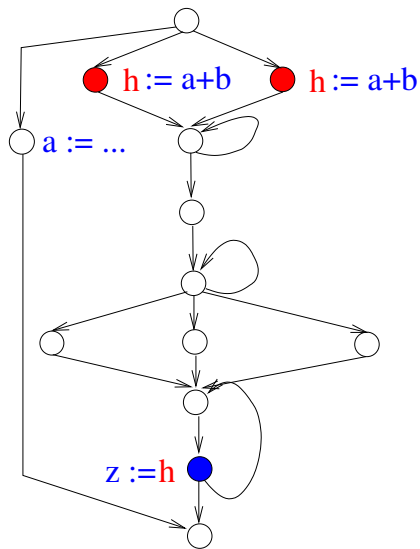
Chap. 11

Chap. 12

738/185

A Computationally Optimal Program

...free of any redundancy!



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

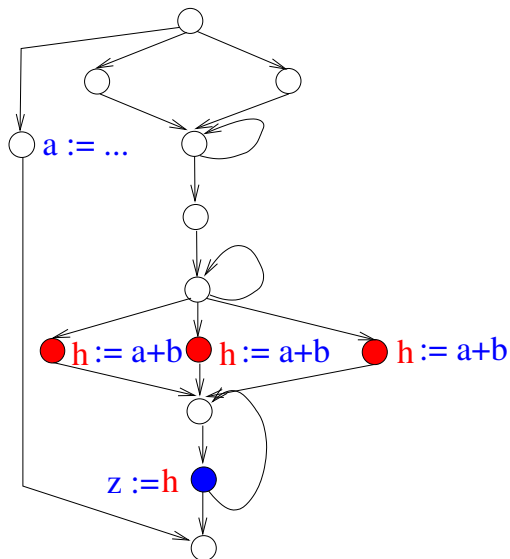
Chap. 11

Chap. 12

739/185

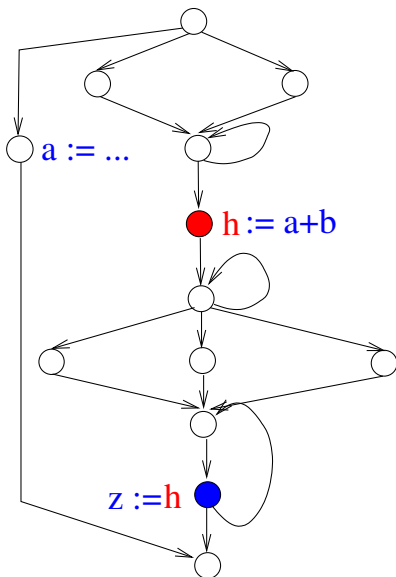
Often there is more than one

...computationally optimal program!



Which one shall PRE deliver?

...computationally optimal, too.



The (Optimization) Goals make the Difference!

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

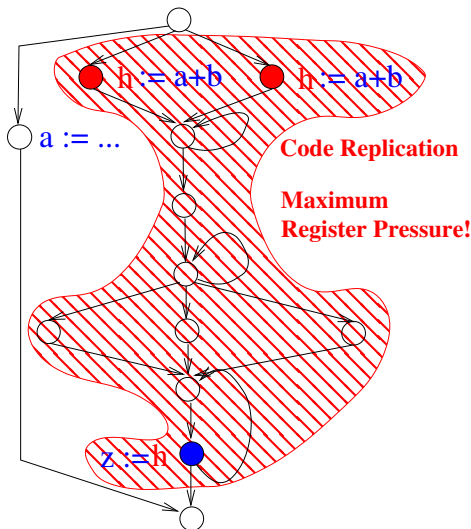
Part III

Chap. 11

Chap. 12

Transformation I: No Remaining Redundancies

...but maximum register pressure and code replication!



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

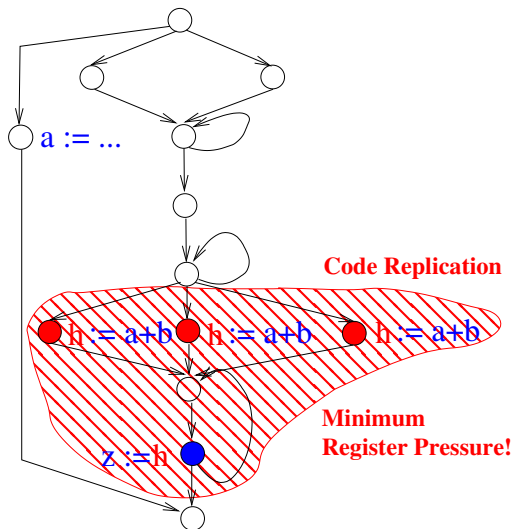
Chap. 11

Chap. 12

743/185

Transformation II: No Remaining Redundancies

...minimum register pressure but code replication.



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

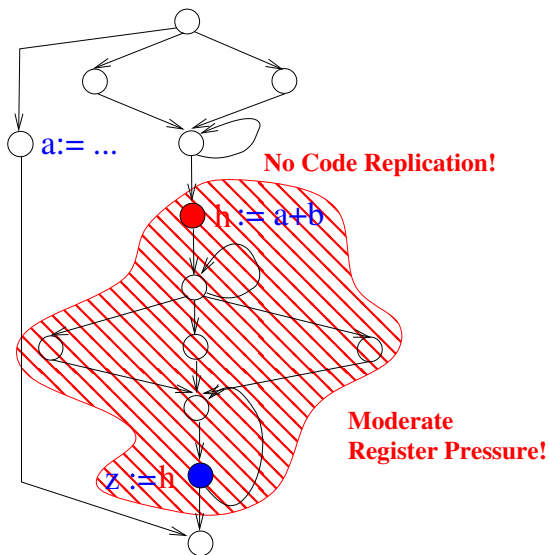
Part III

Chap. 11

Chap. 12

Transformation III: No Remaining Redundancies

...no code replication, moderate register pressure!



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

745/185

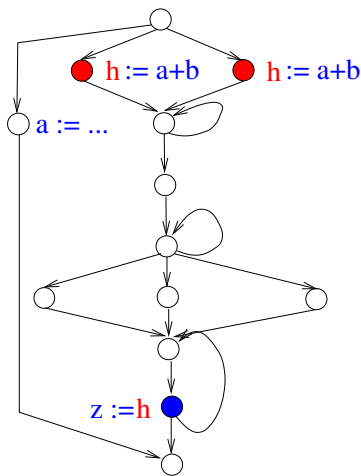
The (Optimization) Goals make the Difference!

In our running example:

- ▶ **Performance**: Avoiding unnecessary (re-) computations
~> Computational quality, **computational optimality**
- ▶ **Register pressure**: Avoiding unnecessary code motion
~> Lifetime quality, **lifetime optimality**
- ▶ **Space**: Avoiding unnecessary code replication
~> Code size quality, **code size optimality**

Transformation I: Busy Code Motion

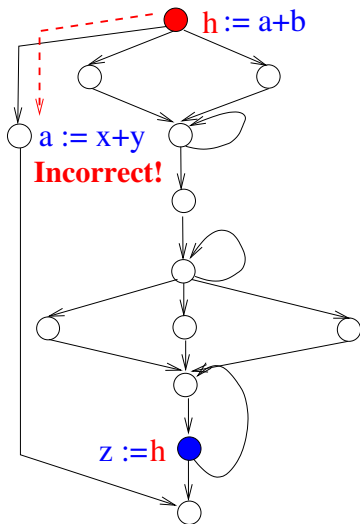
...placing computations as early as possible!



...yields computationally optimal programs.

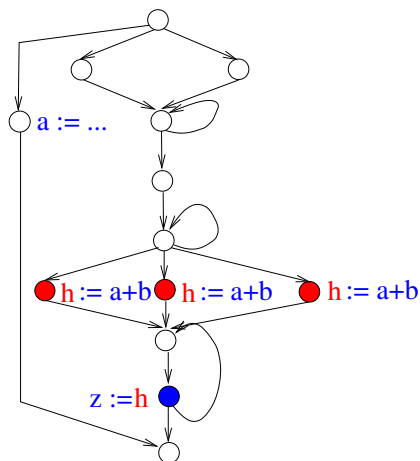
Note: As Early as Possible

...means 'earliest' but not earlier.



Transformation II: Lazy Code Motion

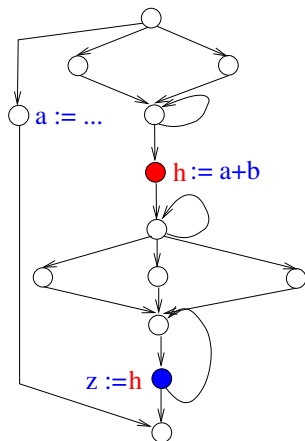
...placing computations as late as possible!



...yields computationally and lifetime optimal programs.

Transformation III: Sparse Code Motion

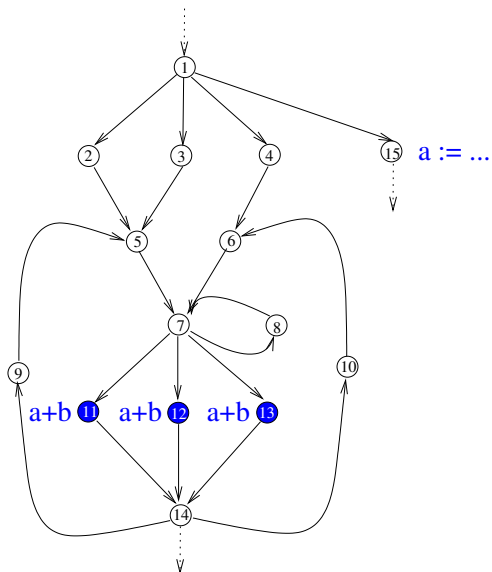
...placing computations as late as possible but as early as necessary!



...yields comp. and lifetime best **code-size optimal** programs.

A More Complex Example (1)

...for illustrating the effect of different PRE-variants.



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

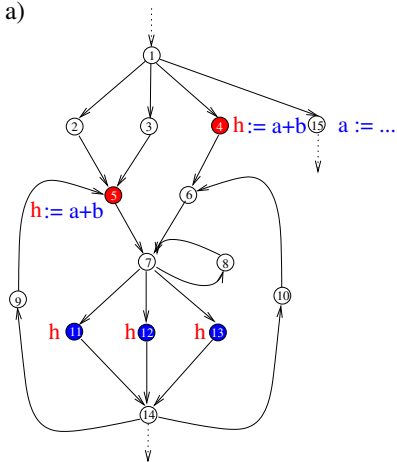
Chap. 11

Chap. 12

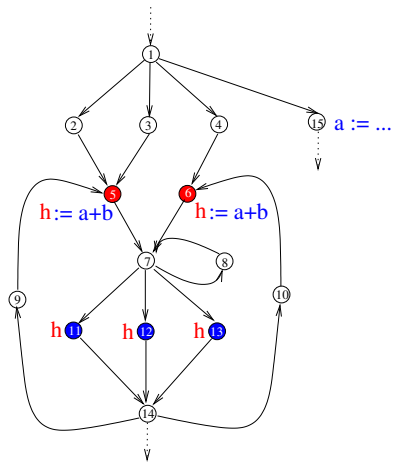
751/185

A More Complex Example (2)

a)

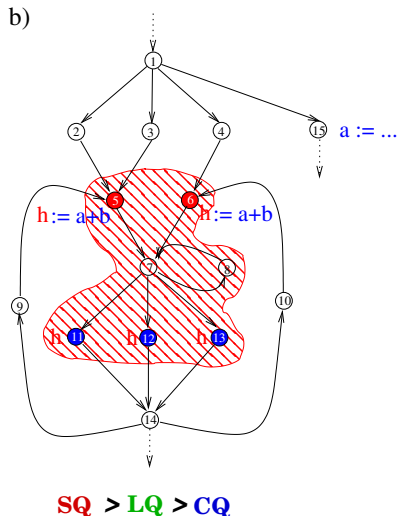
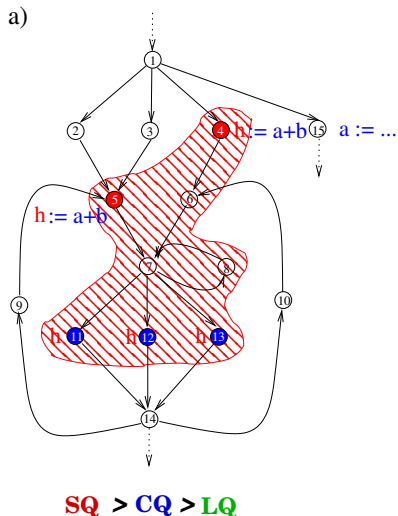


b)



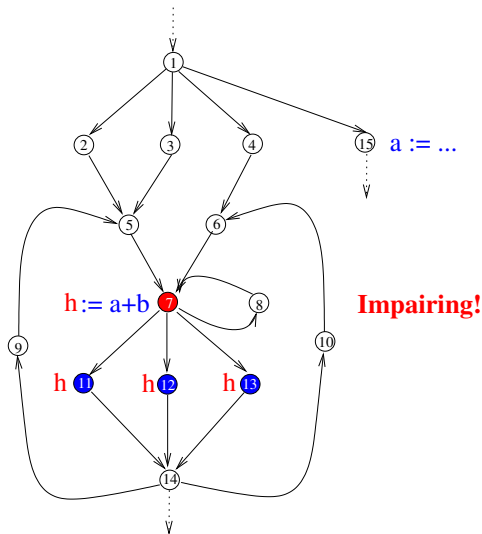
Two Code-size Optimal Programs

A More Complex Example (3)



A More Complex Example (4)

Note: The below transformation, though sound, is **not desired!**



Summing up

The previous examples demonstrate that in general we can not achieve

- ▶ computational optimality
- ▶ lifetime optimality
- ▶ space optimality

at the same time.

However, given a

- ▶ **prioritization** of computational/lifetime/space optimality

we can deliver a program that is

- ▶ **optimal** with respect to the **requested prioritization**

of these goals.

Chapter 6.3

The Groundbreaking PRE Algorithm of Morel and Renvoise

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

756/185

The Groundbreaking Algorithm for PRE

PRE (or Code Motion (CM)) is intrinsically tied to Etienne Morel and Claude Renvoise.

Conceptually

- ▶ The PRE algorithm of Morel and Renvoise presented in 1979 can be considered the *prime father* of all code motion (CM) algorithms
- ▶ continued to be the 'state of the art' CM algorithm until the early 1990s.

Technically, the PRE algorithm of Morel and Renvoise is composed of:

- ▶ 3 unidirectional bitvector analyses (AV, ANT, PAV)
- ▶ 1 bidirectional bitvector analysis (PP)

The PRE Algorithm of Morel & Renvoise (1)

The PRE Analyses

I Availability, unidirectional

$$\mathbf{AVIN}(n) = \begin{cases} \text{false} & \text{if } n = \mathbf{s} \\ \prod_{m \in \text{pred}(n)} \mathbf{AVOUT}(m) & \text{otherwise} \end{cases}$$

$$\mathbf{AVOUT}(n) = \text{TRANSP}(n) * (\text{COMP}(n) + \mathbf{AVIN}(n))$$

II Partial Availability, unidirectional

$$\mathbf{PAVIN}(n) = \begin{cases} \text{false} & \text{if } n = \mathbf{s} \\ \sum_{m \in \text{pred}(n)} \mathbf{PAVOUT}(m) & \text{otherwise} \end{cases}$$

$$\mathbf{PAVOUT}(n) = \text{TRANSP}(n) * (\text{COMP}(n) + \mathbf{PAVIN}(n))$$

The PRE Algorithm of Morel & Renvoise (2)

III Anticipability (very busyness), unidirectional

$$\mathbf{ANTIN}(n) = \text{COMP}(n) + \text{TRANSP}(n) * \mathbf{ANTOUT}(n)$$

$$\mathbf{ANTOUT}(n) = \begin{cases} \text{false} & \text{if } n = \mathbf{e} \\ \prod_{m \in \text{succ}(n)} \mathbf{ANTIN}(m) & \text{otherwise} \end{cases}$$

The PRE Algorithm of Morel & Renvoise (3)

IV Placement Possible, bidirectional

$$\mathbf{PPIN}(n) = \begin{cases} \text{false} & \text{if } n = \mathbf{s} \\ \mathbf{CONST}(n) * \\ \left(\prod_{m \in \text{pred}(n)} (\mathbf{PPOUT}(m) + \mathbf{AVOUT}(m)) \right) * \\ (\mathbf{COMP}(n) + \mathbf{TRANSP}(n) * \mathbf{PPOUT}(n)) & \\ \text{otherwise} \end{cases}$$
$$\mathbf{PPOUT}(n) = \begin{cases} \text{false} & \text{if } n = \mathbf{e} \\ \prod_{m \in \text{succ}(n)} \mathbf{PPIN}(m) & \text{otherwise} \end{cases}$$

where

$$\mathbf{CONST}(n) =_{df}$$

$$\mathbf{ANTIN}(n) * (\mathbf{PAVIN}(n) + \overline{\mathbf{COMP}(n)}) * \mathbf{TRANSP}(n)$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

760/185

The PRE Algorithm of Morel & Renvoise (4)

The PRE Transformation

Initializing temporaries

$$\begin{aligned}\mathbf{INSIN}(n) &=_{df} \textit{false} \\ \mathbf{INSOUT}(n) &=_{df} \mathbf{PPOUT}(n) * \overline{\mathbf{AVOUT}(n)} * \\ &\quad (\overline{\mathbf{PPIN}(n)} + \overline{\mathbf{TRANSP}(n)})\end{aligned}$$

Replacing original computations by references to temporaries

$$\mathbf{REPLACE}(n) =_{df} \mathbf{COMP}(n) * \mathbf{PPIN}(n)$$

Achievements, Merits

...of Morel and Renvoise's PRE algorithm:

- ▶ First algorithm for global PRE

Before 1979: PRE restricted to

- ▶ Basic blocks: Value numbering
- ▶ Program loops: Loop invariant code motion

- ▶ Computationally optimal results

- ▶ State-of-the-art algorithm for global PRE for about 15 years

Shortcomings, Limitations

...of Morel and Renvoise's PRE algorithm:

▶ Conceptually

▶ Computational optimality

↪ Achieved if **critical edges are split** (not part of the original algorithm formulation)

▶ Lifetime optimality

↪ Register pressure is heuristically dealt with, no optimality

▶ Code-size optimality

↪ Not addressed, no objective

▶ Technically

▶ Bidirectional

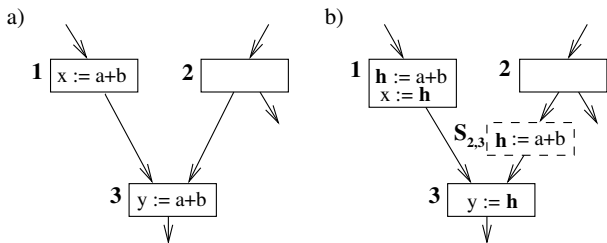
↪ conceptually and computationally more complex than unidirectional analyses

...the transformation result lies (unpredictably) between those of the **BCM** transformation and the **LCM** transformation.

Critical Edges

An edge is called **critical**, if it connects a branching node with a join node.

Illustration:



...splitting the **critical edge** from node **2** to node **3** by inserting the **synthetic node** $S_{2,3}$ allows **PRE** to **eliminate** the **partially redundant computation** of $a + b$ at node **3**, which otherwise would **not safely be possible**.

Chapter 6.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III


Chap. 11

Chap. 12



765/185

Further Reading for Chapter 6 (1)

The Groundbreaking PRE Algorithm of Morel and Renvoise




-  Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.

Variations and Improvements of Morel/Renvoise's Algorithm

-  Dhananjay M. Dhamdhere. *Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise*. ACM Transactions on Programming Languages and Systems 13(2):291-294, 1991, Technical Correspondence.
-  Karl-Heinz Drechsler, Manfred P. Stadel. *A Solution to a Problem with Morel and Renvoise's "Global Optimization by Suppression of Partial Redundancies"*. ACM Transactions on Programming Languages and Systems 10(4):635-640, 1988, Technical Correspondence.

Further Reading for Chapter 6 (2)

Textbook Presentations of PRE

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 9.5, Partial-Redundancy Elimination)
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 8.6, Global Redundancy Elimination)
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 13, Redundancy Elimination)

Further Reading for Chapter 6 (3)

Miscellaneous



Andrei P. Ershov. *On Programming of Arithmetic Operations*. Communications of the ACM 1(8):3-6, 1958. (Three figures from this article are in CACM 1(9):16).

Chapter 7

Busy Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

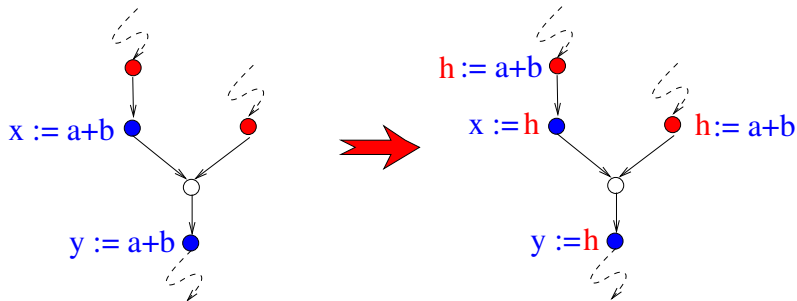
Chap. 11

Chap. 12

Code Motion: Recalling the Very Idea

Code Motion (CM) – often synonymously called Partial Redundancy Elimination (PRE) – aims at:

...avoiding multiple (re-) computations of the same value!



Chapter 7.1

Preliminaries, Problem Definition

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.1.1

7.1.2

7.1.3

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

771/185

Work Plan

In the following we will introduce and formally define:

- ▶ The set of **CM transformations** CM
- ▶ The set of **admissible CM transformations** CM_{Adm}
- ▶ The set of **computationally optimal CM transformations** CM_{CmpOpt}
- ▶ The **BCM transformation** as one specific computationally optimal CM transformation

We start recalling useful notations and common assumptions.

Useful Notations

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph. Then

- ▶ $pred(n) =_{df} \{m \mid (m, n) \in E\}$ denote the set of all predecessors.
- ▶ $succ(n) =_{df} \{m \mid (n, m) \in E\}$ denote the set of all successors.
- ▶ $source(e)$, $dest(e)$ denote the start node and end node of an edge.
- ▶ a sequence of edges $\langle e_1, \dots, e_k \rangle$ where $dest(e_i) = source(e_{i+1})$ for all $1 \leq i < k$ denotes a finite path.

Note: We also consider sequences of nodes as paths, if appropriate.

Useful Notations (Cont'd)

- ▶ $p = \langle e_1, \dots, e_k \rangle$ denotes a **path** from m to n , if $source(e_1) = m$ and $dest(e_k) = n$.
- ▶ $\mathbf{P}[m, n]$ denotes the set of **all paths** from m to n .
- ▶ λ_p denotes the **length** of p , i.e., the number of edges of p .
- ▶ ε denotes the path of length 0.
- ▶ $N_J \subseteq N$ denotes the set of **join nodes**, i.e., the set of nodes w / more than one predecessor.
- ▶ $N_B \subseteq N$ denotes the set of **branch nodes**, i.e. the set of nodes w / more than one successor.

Assumptions on Flow Graphs

Let $G = (N, E, s, e)$ be a flow graph. W/out losing generality we assume (as is common):

General assumptions in program analysis and optimization

- ▶ G is a node-labelled SI graph.
- ▶ Every node of G lies on a path from s to e .
Intuitively: Unreachable parts of G are removed.

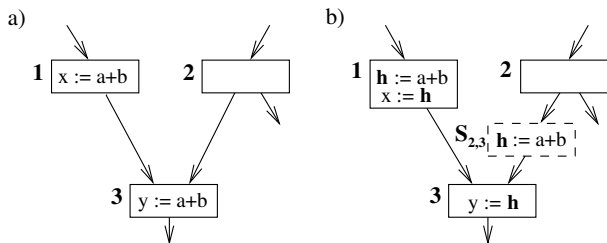
CM specific assumption

- ▶ Critical edges of G are split by inserting new so-called synthetic nodes.
Note: Splitting critical edges is required to enable computationally optimal transformation results.

Splitting Critical Edges

...edges connecting a **branch node** with a **join node** are crucial for code motion and are thus considered **critical**:

Illustration:



...the critical edge (2, 3) connecting **branch node 2** and **join node 3** is split by introducing the **synthetic node $S_{2,3}$** and allows us to remove the partially redundant computation of $a + b$ at 3.

Splitting Edges More Liberally

Splitting **critical edges**:

- ▶ **Computationally optimal CM results** can be achieved, if just **critical edges** in a flow graph are **split**.
- ▶ **CM algorithms** need to store results of computations for later reuse at both **node entries (N-initializations)** and at **node exits (X-Initializations)**.
- ▶ Algorithmically, this is not a problem at all.

Splitting (more liberally) **join edges**:

- ▶ Splitting **all edges leading to a join node** (and not just critical ones) simplifies (the presentation of) code motion.
- ▶ **Computationally optimal CM results** can be achieved by storing the results of computations for later reuse **uniformly** at **node entries (N-initializations)**.

Final Assumption: Join Edges are Split

In the following we thus **assume**:

- ▶ Every edge in G leading to a **join node** is split by inserting a synthetic node.

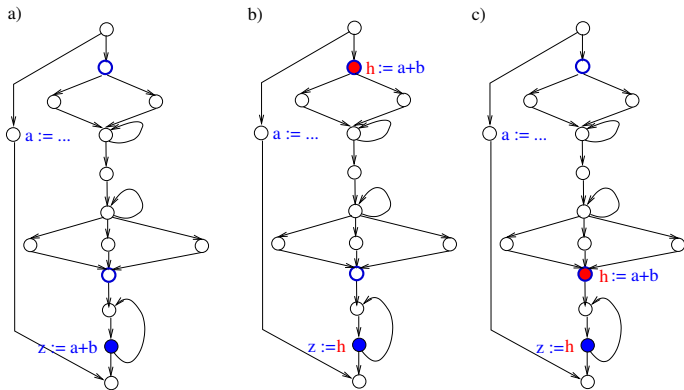
Note: Synthetic nodes, where no instruction will be placed by a code motion transformation, can be removed afterwards in a final cleaning step.

Example

- ▶ **Join edges** like the one connecting **node 1** and **node 3** in the example illustrating the splitting of critical edges are assumed to be split by inserting a **synthetic node $S_{1,3}$** .

Note

...allowing structural changes (beyond plain edge splitting) would allow even more powerful results:



...the algorithms we are going to present will not introduce new program points on their own but will make best use of program points available.

Chapter 7.1.1

Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.1.1

7.1.2

7.1.3

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

780/185

CM: The General Transformation Pattern

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a (node-labelled SI) flow graph, and let $t \in \mathbf{T}$ be a term, the so-called candidate expression for code motion.

Definition 7.1.1.1 (CM Transformation)

A CM transformation for t , CM_t , consists of two steps:

- ▶ Inserting at (the entry of) some nodes of G the instruction $h := t$, where h is a new variable.
- ▶ Replacing some of the original occurrences of t by h .

We denote the set of CM transformations for t by CM_t .

Specifying CM Transformations

A CM transformation CM_t is completely specified by means of two predicates (defined on nodes)

- ▶ $Insert_{CM_t} : N \rightarrow \text{IB}$
- ▶ $Repl_{CM_t} : N \rightarrow \text{IB}$

specifying where to

- ▶ **store** the result of a computation
- ▶ **replace** an original computation of t by a reference to a stored value

in G , respectively.

In the following we will consider an arbitrary but **fixed** expression t , the so-called **candidate expression**, which allows us to drop t as an index.

Chapter 7.1.2

Admissible Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.1.1

7.1.2

7.1.3

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

783/185

Towards Admissible CM Transformations

Obviously, CM includes transformations, which do not preserve the semantics of the original program, and are thus not acceptable.

This leads us to the notion of **admissible CM transformations**.

Informally:

- ▶ A CM transformation is **admissible**, if it is **safe** and **correct**.
- ▶ **Safe**: There is no path, on which by inserting an initialization of h a **new** value is computed, i.e., a value that has not been computed in the original program along this path.
- ▶ **Correct**: Whenever the temporary h is referenced, it stores the 'right' value, i.e., it stores the same value a recomputation of t at the use site would yield.

Towards formalising Safety and Correctness

We need to have three (local) predicates (defined on nodes):

- ▶ $Comp_t(n)$: The candidate expression t is computed at n .
- ▶ $Transp_t(n)$: Node n is transparent for t , i.e., n does not modify any operand of t .
- ▶ $Comp_{CM_t}(n) =_{df} Insert_{CM_t}(n) \vee Comp_t(n) \wedge \neg Repl_{CM_t}(n)$:
The candidate expression t is computed at node n after CM_t has been applied.

Note: In the following we will resume dropping t as an index.

Extending Predicates from Nodes to Paths

Let p be a path (in terms of sequence of nodes), let p_i denote the i -th node of p , and let *Predicate* be a predicate defined on nodes.

We define:

- ▶ $Predicate^{\forall}(p) \iff_{df} \forall 1 \leq i \leq \lambda_p. Predicate(p_i)$
- ▶ $Predicate^{\exists}(p) \iff_{df} \exists 1 \leq i \leq \lambda_p. Predicate(p_i)$

Safety and Correctness

Definition 7.1.2.1 (Safety and Correctness)

Let $n \in N$, let $CM \in \mathcal{CM}$. Then:

1. $Safe(n) \iff_{df}$

$\forall \langle n_1, \dots, n_k \rangle \in \mathbf{P}[s, e] \forall i. (n_i = n) \Rightarrow$

i) $\exists j < i. Comp(n_j) \wedge Transp^{\forall}(\langle n_j, \dots, n_{i-1} \rangle) \vee$

ii) $\exists j \geq i. Comp(n_j) \wedge Transp^{\forall}(\langle n_i, \dots, n_{j-1} \rangle)$

2. $Correct_{CM}(n) \iff_{df}$

$\forall \langle n_1, \dots, n_k \rangle \in \mathbf{P}[s, n] \exists i.$

$Insert_{CM}(n_i) \wedge Transp^{\forall}(\langle n_i, \dots, n_{k-1} \rangle)$

Up-Safety and Down-Safety

Considering the conditions (i) resp. (ii) of the definition of **safety** separately, leads us to the notions of

- ▶ **up-safety** (availability)
- ▶ **down-safety** (anticipability, very busyness)

Intuitively, a computation of t at node n is

- ▶ **up-safe**, if t is computed on all paths p from s to n and the last computation of t on p is not followed by a modification of (an operand of) t .
- ▶ **down-safe**, if t is computed on all paths p from n to e and the first computation of t on p is not preceded by a modification of (an operand of) t .

Up-Safety and Down-Safety

Definition 7.1.2.2 (Up-Safety and Down-Safety)

1. $\forall n \in \mathbf{N}. U\text{-Safe}(n) \iff_{df} \forall p \in \mathbf{P}[s, n] \exists i < \lambda_p. \text{Comp}(p_i) \wedge \text{Transp}^{\forall}(p[i, \lambda_p[)$
2. $\forall n \in \mathbf{N}. D\text{-Safe}(n) \iff_{df} \forall p \in \mathbf{P}[n, e] \exists i \leq \lambda_p. \text{Comp}(p_i) \wedge \text{Transp}^{\forall}(p[1, i[)$

Admissible CM Transformations

Now we are ready to define the set of **admissible CM transformations** formally:

Definition 7.1.2.3 (Admissible CM-Transformation)

A CM transformation $CM \in \mathcal{CM}$ is **admissible** iff for every node $n \in N$ holds:

1. $Insert_{CM}(n) \Rightarrow Safe(n)$
2. $Repl_{CM}(n) \Rightarrow Correct_{CM}(n)$

We denote the **set of admissible CM transformations** by \mathcal{CM}_{Adm} .

Important Results on Safety and Correctness

Lemma 7.1.2.4 (Safety)

$$\forall n \in N. \text{Safe}(n) \iff U\text{-Safe}(n) \vee D\text{-Safe}(n)$$

Lemma 7.1.2.5 (Correctness)

$$\forall CM \in \mathcal{CM}_{Adm} \forall n \in N. \text{Correct}_{CM}(n) \Rightarrow \text{Safe}(n)$$

Chapter 7.1.3

Computationally Optimal Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.1.1

7.1.2

7.1.3

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

792/185

Computationally Better: Higher Performance

Definition 7.1.3.1 (Computationally Better)

A CM transformation $CM \in \mathcal{CM}_{Adm}$ is **computationally better** than a CM transformation $CM' \in \mathcal{CM}_{Adm}$ iff

$$\forall p \in \mathbf{P}[s, e]. \quad |\{i \mid Comp_{CM}(p_i)\}| \leq |\{i \mid Comp_{CM'}(p_i)\}|$$

Note: The relation ‘**computationally better**’ is a quasi-order, i.e., a reflexive and transitive relation.

Comp. Optimal: Highest Performance

Definition 7.1.3.2 (Comp. Optimal Code Motion)

An admissible CM transformation $CM \in \mathcal{CM}_{Adm}$ is **computationally optimal** iff CM is computationally better than every other admissible CM transformation.

We denote the set of computationally optimal CM transformations by $\mathcal{CM}_{CompOpt}$.

Reminder: Selected Properties of Relations

Let M be a set and R be a relation on M , i.e., $R \subseteq M \times M$.

Then R is called

- ▶ **reflexive** iff $\forall m \in M. m R m$
- ▶ **transitive** iff $\forall m, n, p \in M. m R n \wedge n R p \Rightarrow m R p$
- ▶ **anti-symmetric** iff $\forall m, n \in M. m R n \wedge n R m \Rightarrow m = n$
- ▶ **quasi order** iff R is reflexive and transitive
- ▶ **partial order** iff R is reflexive, transitive and anti-symmetric

Chapter 7.2

The *BCM* Transformation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Conceptually

...code motion can be considered a **two-stage process**:

1. **Hoisting expressions**

...hoisting expressions to **'earlier'** safe computation points.

2. **Eliminating totally redundant expressions**

...eliminating computations getting totally redundant by hoisting expressions.

The Earliestness Principle

...induces an extreme placing (i.e., hoisting) strategy:

Placing computations **as early as possible**...

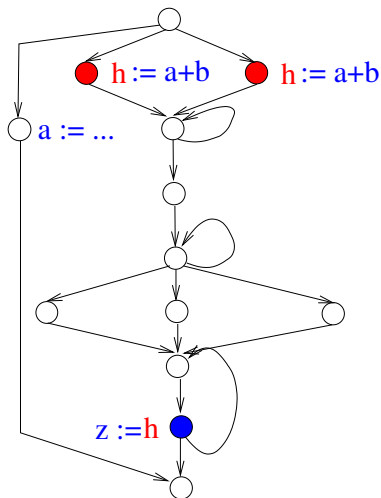
- ▶ Hoisting computations to their **earliest** safe computation points yields **computationally optimal** programs (cf. **Theorem 7.2.2**).

~> known as the **Busy Code Motion**.

Illustrating the Earliestness Principle

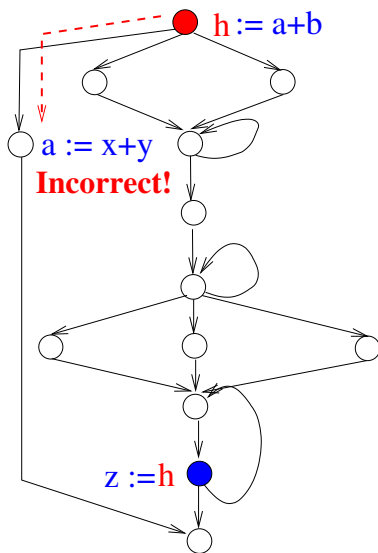
Placing computations **earliest...**

yields **computationally optimal** programs.



Recall

...earliest means as early as possible, but not earlier!



Busy Code Motion

Intuitively:

Place computations **as early as possible** in a program while preserving **safety** and **correctness**!

Note: Following this principle computations are moved as far as possible in the opposite direction of the control flow

~> ...motivates the choice of the term **busy**.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Earliest Program Points

Definition 7.2.1 (Earliestness)

$\forall n \in N. \text{Earliest}(n) =_{df}$

$$\text{Safe}(n) \wedge \begin{cases} \text{true} & \text{if } n = \mathbf{s} \\ \bigvee_{m \in \text{pred}(n)} \neg \text{Transp}(m) \vee \neg \text{Safe}(m) & \text{otherwise} \end{cases}$$

The *BCM* Transformation

...is defined by:

- ▶ $\forall n \in \mathbb{N}. \text{Insert}_{BCM}(n) =_{df} \text{Earliest}(n)$
- ▶ $\forall n \in \mathbb{N}. \text{Repl}_{BCM}(n) =_{df} \text{Comp}(n)$

The *BCM* Transf.: Computationally Optimal

Theorem 7.2.2 (*BCM* Theorem)

The *BCM* transformation is computationally optimal, i.e.:

$$BCM \in \mathcal{CM}_{CmpOpt}$$

Proof. By means of the Earliestness Lemma 7.2.3 and the *BCM* Lemma 7.2.4.

Properties of Earliest Program Points

Lemma 7.2.3 (Earliestness Lemma)

Let $n \in N$. Then we have:

1. $\text{Safe}(n) \Rightarrow$

$$\forall p \in \mathbf{P}[s, n] \exists i \leq \lambda_p. \text{Earliest}(p_i) \wedge \text{Transp}^{\forall}(p[i, \lambda_p[))$$

2. $\text{Earliest}(n) \iff$

$$D\text{-Safe}(n) \wedge \bigwedge_{m \in \text{pred}(n)} (\neg \text{Transp}(m) \vee \neg \text{Safe}(m))$$

3. $\text{Earliest}(n) \iff$

$$\text{Safe}(n) \wedge \forall CM \in \mathcal{CM}_{Adm}. \text{Correct}_{CM}(n) \Rightarrow \text{Insert}_{CM}(n)$$

Properties of the *BCM* Transformation

Lemma 7.2.4 (*BCM* Lemma)

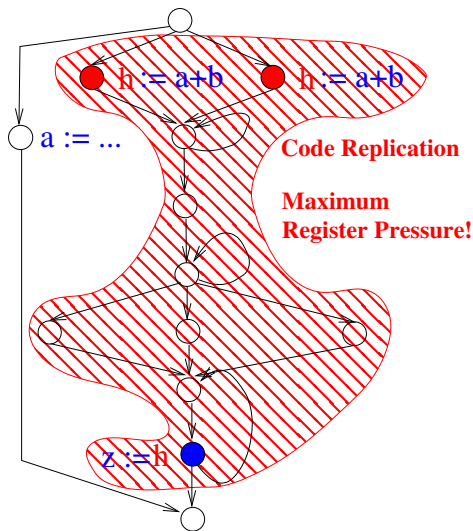
Let $p \in \mathbf{P}[s, e]$. Then we have:

- $\forall i \leq \lambda_p. \text{Insert}_{BCM}(p_i) \iff \exists j \geq i. p[i, j] \in \text{FU-LtRg}(BCM)$
- $\forall CM \in \mathcal{CM}_{Adm}. \forall i, j \leq \lambda_p. p[i, j] \in \text{LtRg}(BCM) \Rightarrow \text{Comp}_{CM}^{\exists}(p[i, j])$
- $\forall CM \in \mathcal{CM}_{CmpOpt}. \forall i \leq \lambda_p. \text{Comp}_{CM}(p_i) \Rightarrow \exists j \leq i \leq l. p[j, l] \in \text{FU-LtRg}(BCM)$

Note: Lifetime ranges (*LtRg*) and first-use lifetime ranges (*FU-LtRg*) will formally be introduced in [Definition 8.1.1.1](#).

The Result of the *BCM* Transformation

...computationally optimal but maximum register pressure.



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chapter 7.3

Up-Safety and Down-Safety: The DFA Specifications

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Note

Up-safety and down-safety

- ▶ are just synonyms for [availability](#) and [very busyness](#), respectively.

Hence, the [DFA specifications](#) for

- ▶ [availability](#) and [very busyness](#) of [Chapter 4](#) can be reused and need only be adapted from [edge](#) to [node-labelled SI flow graphs](#).

Up-Safety: The DFA Specification

...for a CM candidate expression t , $t \in \mathbf{T}$.

DFA Specification

- ▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathbf{IB}, \wedge, \vee, \leq, false, true) = \widehat{\mathbf{IB}}$$

- ▶ DFA functional

$$\llbracket \cdot \rrbracket_{us}^t : N \rightarrow (\mathbf{IB} \rightarrow \mathbf{IB}), \text{ where}$$

$$\forall n \in N \forall b \in \mathbf{IB}. \llbracket n \rrbracket_{us}^t(b) =_{df} (b \vee Comp_n^t) \wedge Transp_n^t$$

- ▶ Initial information: $b_s \in \mathbf{IB}$
- ▶ Direction of information flow: forward

Up-Safety Specification for t

- ▶ Specification: $\mathcal{S}_G^{us,t} = (\widehat{\mathbf{IB}}, \llbracket \cdot \rrbracket_{us}^t, b_s, fw)$

Down-Safety: The DFA Specification

...for a CM candidate expression t , $t \in \mathbf{T}$.

DFA Specification

▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathbf{IB}, \wedge, \vee, \leq, false, true) = \widehat{\mathbf{IB}}$$

▶ DFA functional

$$\llbracket \cdot \rrbracket_{ds}^t : N \rightarrow (\mathbf{IB} \rightarrow \mathbf{IB}), \text{ where}$$

$$\forall n \in N \forall b \in \mathbf{IB}. \llbracket n \rrbracket_{ds}^t(b) =_{df} (b \wedge Transp_n^t) \vee Comp_n^t$$

▶ Initial information: $b_e \in \mathbf{IB}$

▶ Direction of information flow: backward

Down-Safety Specification for t

▶ Specification: $\mathcal{S}_G^{ds,t} = (\widehat{\mathbf{IB}}, \llbracket \cdot \rrbracket_{ds}^t, b_e, bw)$

A Hint to Appendix C

Appendix C presents

- ▶ the specialized versions of the *MaxFP* equation systems induced by $\mathcal{S}_G^{us,t}$ and $\mathcal{S}_G^{ds,t}$, respectively, for
 - ▶ single instruction flow graphs (cf. Appendix C.1.2).
 - ▶ basic block flow graphs (cf. Appendix C.2.2).
- ▶ an illustrating example of the *BCM* transformation on basic block flow graphs.

Chapter 7.4

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

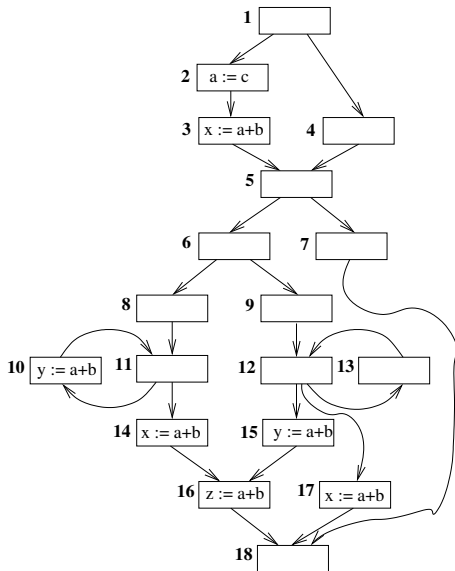
Chap. 10

Part III

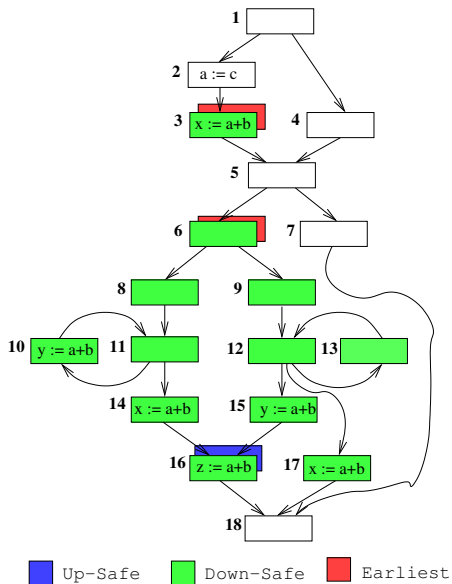
Chap. 11

Chap. 12

The Original Program



Up-Safe, Down-Safe & Earliest Program Points



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

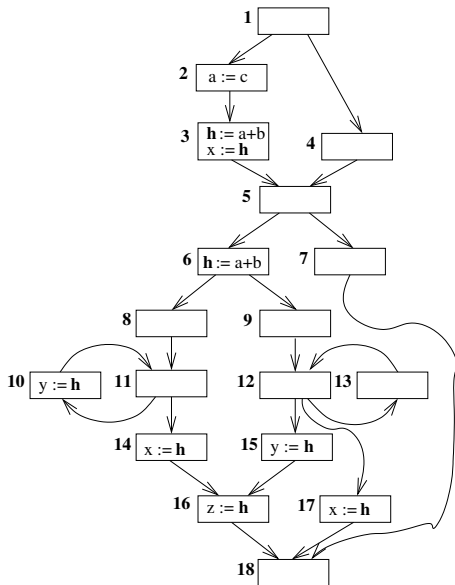
Chap. 10

Part III

Chap. 11

Chap. 12

The Result of the *BCM* Transformation



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

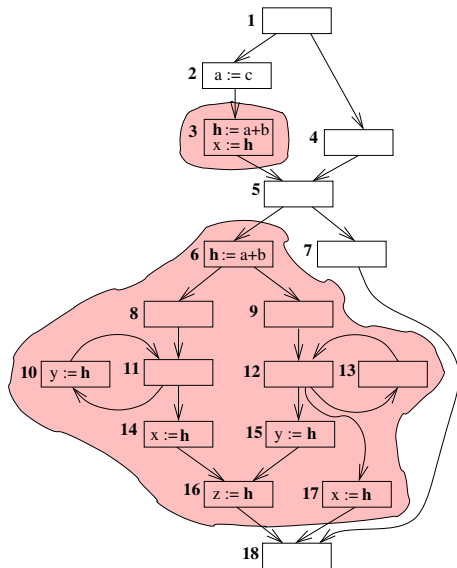
Part III

Chap. 11

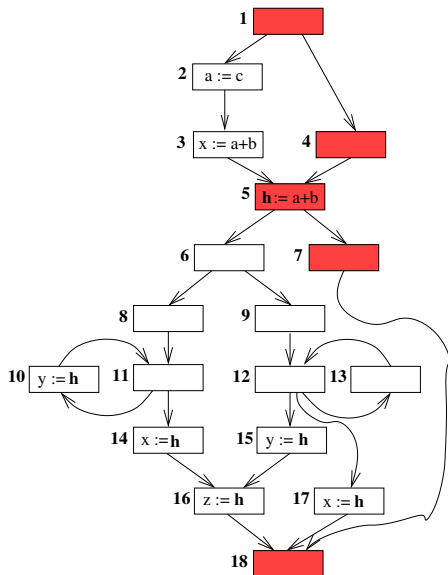
Chap. 12

BCM Transf.: Achievements & Shortcomings

Computationally optimal but **maximum register pressure.**



Note: Initializing Even Earlier is Not Correct!



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chapter 7.5

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

7.1

7.2

7.3

7.4

7.5

Chap. 8

Chap. 9




Chap. 10

Part III

Chap. 11

Chap. 12

Further Reading for Chapter 7

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retro-spective: Lazy Code Motion*. In “20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection”, ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.

Chapter 8

Lazy Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

Part III

Chap. 11

The Latestness Principle

...induces an extreme placing strategy **dual** to the **earliestness principle**:

Placing computations **as late as possible**...

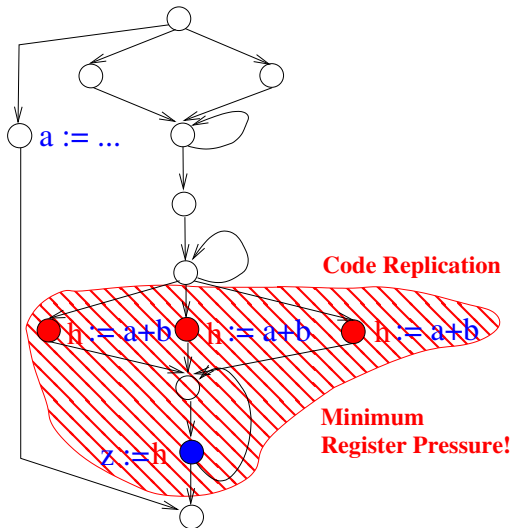
- ▶ Hoisting computations **as little as possible**, but **as far as necessary** (to achieve **computational optimality**), yields **computationally optimal programs w/ minimum register pressure** (cf. **Theorem 8.3.3** and **Corollary 8.3.4**).

↪ known as the **Lazy Code Motion**

Illustrating the Latestness Principle

Placing computations as late as possible yields...

comp. optimal programs w/ minimum register pressure!



Lazy Code Motion

Intuitively:

Place computations **as late as possible** in a program while preserving **safety, correctness and computational optimality!**

Note: Following this principle computations are moved as little as possible in the opposite direction of the control flow

~> ...motivates the choice of the term **lazy**.

Chapter 8.1

Preliminaries, Problem Definition

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.1.1

8.1.2

8.1.3

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

825/185

Work Plan

In the following we will introduce and formally define:

- ▶ The notions of **lifetime** and **first-use lifetime ranges**.
- ▶ The set of **almost lifetime optimal** CM transformations \mathcal{CM}_{ALtOpt} .
- ▶ The set of **lifetime optimal** CM transformations \mathcal{CM}_{LtOpt} .
- ▶ The **LCM transformation** as the uniquely determined sole **computationally** and **lifetime optimal** CM transformation.

Chapter 8.1.1

Lifetime Ranges

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.1.1

8.1.2

8.1.3

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

827/185

Central for Capturing Register Pressure

...formally, is the notion of a (first-use) lifetime range.

Definition 8.1.1.1 (Lifetime Ranges)

Let $CM \in \mathcal{CM}$.

► Lifetime range

$$LtRg(CM) =_{df} \{p \mid Insert_{CM}(p_1) \wedge Repl_{CM}(p_{\lambda_p}) \wedge \neg Insert_{CM}^{\exists}(p]1, \lambda_p)\}$$

► First-use lifetime range

$$FU-LtRg(CM) =_{df} \{p \in LtRg(CM) \mid \forall q \in LtRg(CM). (q \sqsubseteq p) \Rightarrow (q = p)\}$$

First-Use Lifetime Ranges do not Overlap

Lemma 8.1.1.2 (First-Use Lifetime-Range Lemma)

Let $CM \in \mathcal{CM}$, $p \in \mathbf{P}[s, e]$, and let i_1, i_2, j_1, j_2 indexes such that $p[i_1, j_1] \in FU-LtRg(CM)$ and $p[i_2, j_2] \in FU-LtRg(CM)$.

Then we have:

- ▶ either $p[i_1, j_1]$ and $p[i_2, j_2]$ coincide, i.e., $i_1 = i_2$ and $j_1 = j_2$, or
- ▶ $p[i_1, j_1]$ and $p[i_2, j_2]$ are disjoint, i.e., $j_1 < i_2$ or $j_2 < i_1$.

Lifetime Better: Less Register Pressure

Definition 8.1.1.3 (Lifetime Better)

A CM-transformation $CM \in \mathcal{CM}$ is **lifetime better** than a CM-transformation $CM' \in \mathcal{CM}$ iff

$$\forall p \in LtRg(CM). \exists q \in LtRg(CM'). p \sqsubseteq q$$

Note: The relation 'lifetime better' is a partial order, i.e., a reflexive, transitive, and antisymmetric relation.

Chapter 8.1.2

Almost Lifetime Optimal Code Motion

Almost Lifetime Optimality

...ensures almost minimum register pressure.

Definition 8.1.2.1 (Almost Lifetime Optimal CM)

A comp. optimal CM transformation $CM \in \mathcal{CM}_{CmpOpt}$ is **almost lifetime optimal** iff

$$\forall p \in LtRg(CM). \lambda_p \geq 2 \Rightarrow \\ \forall CM' \in \mathcal{CM}_{CmpOpt} \exists q \in LtRg(CM'). p \sqsubseteq q$$

We denote the set of all almost lifetime optimal CM transformations by \mathcal{CM}_{ALtOpt} .

Chapter 8.1.3

Lifetime Optimal Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.1.1

8.1.2

8.1.3

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

833/185

Lifetime Optimal: Minimum Register Pressure

Definition 8.1.3.1 (Lifetime Optimal Code Motion)

A comp. optimal CM transformation $CM \in \mathcal{CM}_{CmpOpt}$ is **lifetime optimal** iff CM is lifetime better than every other computationally optimal CM transformation.

We denote the set of all lifetime optimal CM transformations by \mathcal{CM}_{LtOpt} .

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.1.1

8.1.2

8.1.3

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

834/185

BCM Lifetime Ranges are Longest

Lemma 8.1.3.2 (*BCM* Lifetime Range Lemma)

$$\forall CM \in \mathcal{CM}_{CmpOpt}. \forall p \in LtRg(CM). \exists q \in LtRg(BCM). \\ p \sqsubseteq q$$

Intuitively

- ▶ There is no **computationally optimal CM transformation** which places computations **earlier** than the **BCM transformation**.
- ▶ The **BCM transformation** is the uniquely determined **computationally optimal CM transformation w/ maximum register pressure**.

Uniqueness of Lifetime Optimal Code Motion

Obviously, we have:

$$\mathcal{CM}_{LtOpt} \subseteq \mathcal{CM}_{CmpOpt} \subseteq \mathcal{CM}_{Adm} \subset \mathcal{CM}$$

In fact, we even have:

Theorem 8.1.3.3 (Uniqueness of Lifetime Opt. CM)

$$|\mathcal{CM}_{LtOpt}| \leq 1$$

Chapter 8.2

The *ALCM* Transformation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

Part III

Chap. 11

837/185

Delayability of Computations

Definition 8.2.1 (Delayability)

$\forall n \in \mathbf{N}. \text{Delayed}(n) \iff_{df}$

$$\forall p \in \mathbf{P}[s, n] \exists i \leq \lambda_p. \text{Earliest}(p_i) \wedge \neg \text{Comp}^\exists(p[i, \lambda_p[)$$

Lemma 8.2.2 (Delayability Lemma)

1. $\forall n \in \mathbf{N}. \text{Delayed}(n) \Rightarrow D\text{-Safe}(n)$
2. $\forall p \in \mathbf{P}[s, e]. \forall i \leq \lambda_p. \text{Delayed}(p_i) \Rightarrow$
 $\exists j \leq i \leq l. p[j, l] \in \text{FU-LtRg}(BCM)$
3. $\forall CM \in \mathcal{CM}_{\text{CompOpt}}. \forall n \in \mathbf{N}. \text{Comp}_{CM}(n) \Rightarrow \text{Delayed}(n)$

Latest Program Points

Definition 8.2.3 (Latestness)

$$\forall n \in N. \textit{Latest}(n) =_{df} \textit{Delayed}(n) \wedge (\textit{Comp}(n) \vee \bigvee_{m \in \textit{succ}(n)} \neg \textit{Delayed}(m))$$

Lemma 8.2.4 (Latestness Lemma)

1. $\forall p \in \textit{LtRg}(BCM) \exists i \leq \lambda_p. \textit{Latest}(p_i)$
2. $\forall p \in \textit{LtRg}(BCM) \forall i \leq \lambda_p. \textit{Latest}(p_i) \Rightarrow \neg \textit{Delayed}^{\exists}(p]i, \lambda_p]$

The *ALCM* Transformation

...is defined by:

- ▶ $\forall n \in N. \text{Insert}_{ALCM}(n) =_{df} \text{Latest}(n)$
- ▶ $\forall n \in N. \text{Repl}_{ALCM}(n) =_{df} \text{Comp}(n)$

The *ALCM* Transf.: Almost Lifetime Optimal

Theorem 8.2.5 (*ALCM* Theorem)

The *ALCM* transformation is almost lifetime optimal, i.e.:

$$ALCM \in \mathcal{CM}_{ALtOpt}$$

Chapter 8.3

The *LCM* Transformation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

Part III

Chap. 11

842/185

Isolated Computation Points of a CM Transf.

Definition 8.3.1 (Isolation_{CM})

$$\forall CM \in \mathcal{CM} \forall n \in N. \text{Isolated}_{CM}(n) \iff_{df} \\ \forall p \in \mathbf{P}[n, e] \forall 1 < i \leq \lambda_p. \text{Repl}_{CM}(p_i) \Rightarrow \text{Insert}_{CM}^{\exists}(p]1, i])$$

Lemma 8.3.2 (Isolation Lemma)

- $\forall CM \in \mathcal{CM} \forall n \in N. \text{Isolated}_{CM}(n) \iff \\ \forall p \in \text{LtRg}(CM). \langle n \rangle \sqsubseteq p \Rightarrow \lambda_p = 1$
- $\forall CM \in \mathcal{CM}_{\text{CmpOpt}} \forall n \in N. \text{Latest}(n) \Rightarrow \\ (\text{Isolated}_{CM}(n) \iff \text{Isolated}_{BCM}(n))$

The LCM Transformation

...is defined by:

- ▶ $\forall n \in N. \text{Insert}_{LCM}(n) =_{df} \text{Latest}(n) \wedge \neg \text{Isolated}_{BCM}(n)$
- ▶ $\forall n \in N. \text{Repl}_{LCM}(n) =_{df}$
 $\text{Comp}(n) \wedge \neg(\text{Latest}(n) \wedge \text{Isolated}_{BCM}(n))$

The *LCM* Transf.: Comp. & Lifetime Optimal

Theorem 8.3.3 (*LCM* Theorem)

The *LCM* transformation is lifetime optimal, i.e.:

$$LCM \in \mathcal{CM}_{LtOpt}$$

Corollary 8.3.4 (*LCM* Corollary)

The *LCM* transformation is computationally optimal, i.e.:

$$LCM \in \mathcal{CM}_{CmpOpt}$$

Chapter 8.4

Delayability and Isolation: The DFA Specifications

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

Chap. 9

Chap. 10

Part III

Chap. 11

Delayability: The DFA Specification

...for a CM candidate expression t , $t \in \mathbf{T}$.

DFA Specification

▶ DFA lattice

$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathbf{IB}, \wedge, \vee, \leq, \text{false}, \text{true}) = \widehat{\mathbf{IB}}$$

▶ DFA functional

$$\llbracket \cdot \rrbracket_{dl}^t : N \rightarrow (\mathbf{IB} \rightarrow \mathbf{IB}), \text{ where}$$

$$\forall n \in N \forall b \in \mathbf{IB}. \llbracket n \rrbracket_{dl}^t(b) =_{df} (b \vee \text{Earliest}^t(n)) \wedge \neg \text{Comp}_n^t$$

▶ Initial information: $\text{Earliest}^t(\mathbf{s}) \in \mathbf{IB}$

▶ Direction of information flow: forward

Delayability Specification for t

▶ Specification: $\mathcal{S}_G^{dl,t} = (\widehat{\mathbf{IB}}, \llbracket \cdot \rrbracket_{dl}^t, \text{Earliest}^t(\mathbf{s}), fw)$

The Property *Delayed*

...of [Definition 8.2.1](#) is now given by:

Lemma 8.4.1 (Delayability Implementation Lemma)

$$\forall n \in N. \text{Delayed}(n) = \nu\text{-inf}_{S_G^{dl,t}}(n) \vee \text{Earliest}^t(n)$$

where $\nu\text{-inf}_{S_G^{dl,t}}$ denotes the *MaxFP* solution of the *delayability* DFA problem $S_G^{dl,t}$ (coinciding with its *MOP* solution).

Isolation: The DFA Specification

...in terms of a *MaxFP*-Equation System:

The *MaxFP*-Equation System for Isolation:

$$\text{N-ISO}[n] = (\overline{\text{COMP}_n^t} \vee \text{Earliest}^t(n)) \wedge \text{X-ISO}[n]$$

$$\text{X-ISO}[n] = \prod_{m \in \text{succ}(n)} \text{N-ISO}[m] \vee \text{Earliest}^t(m)$$

Exercise: Does the above equation system match the standard pattern of the *MaxFP* equation system of a unidirectional DFA backward analysis? Provide the reasoning underlying your answer.

A Hint to Appendix C

Appendix C presents

- ▶ the specialized versions of the *MaxFP* equation systems induced by $\mathcal{S}_G^{dl,t}$ and $\mathcal{S}_G^{iso,t}$, respectively, for
 - ▶ single instruction flow graphs (cf. Appendix C.1.3)
 - ▶ basic block flow graphs (cf. Appendix C.2.3).
- ▶ an illustrating example of the *ALCM* transformation and the *LCM* transformation on basic block flow graphs.

Chapter 8.5

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

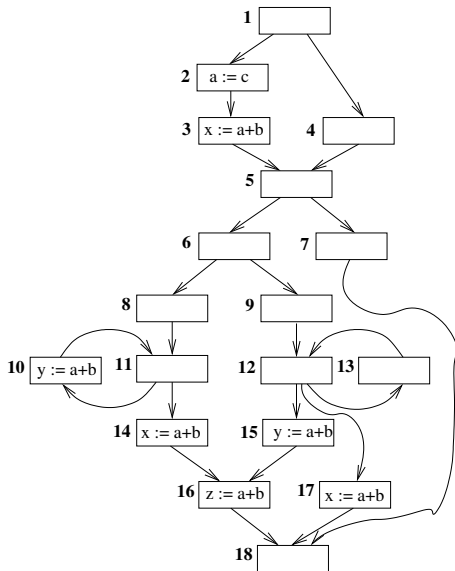
Chap. 9

Chap. 10

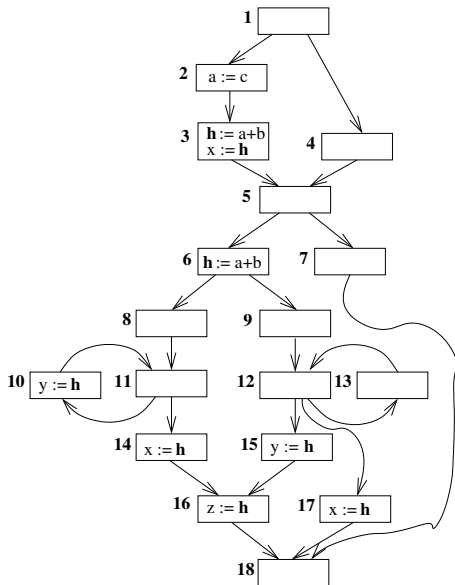
Part III

Chap. 11

The Original Program

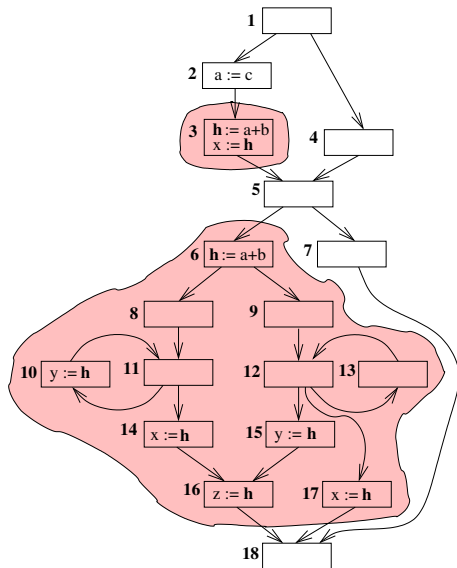


The Result of the *BCM* Transformation

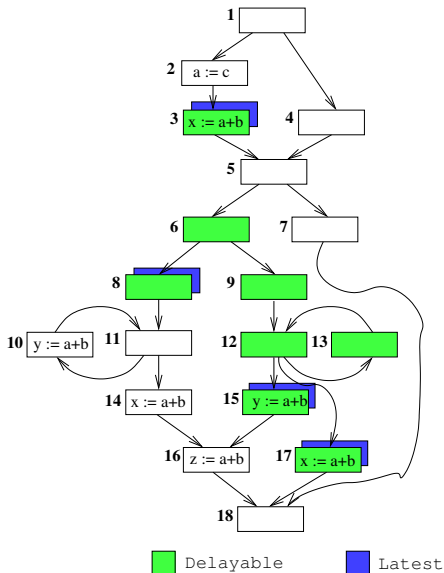


BCM Transf.: Achievements & Shortcomings

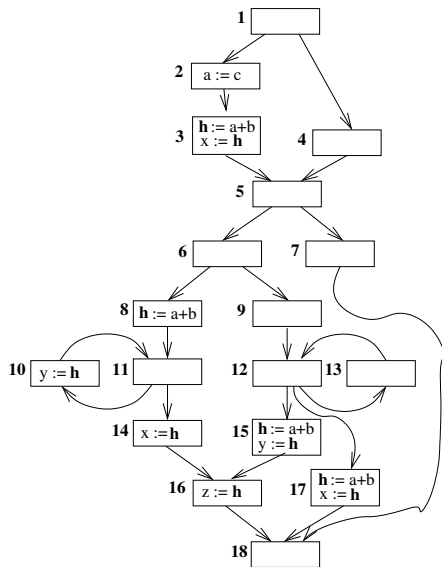
Computationally optimal but **maximum register pressure.**



Delayed and Latest Computation Points

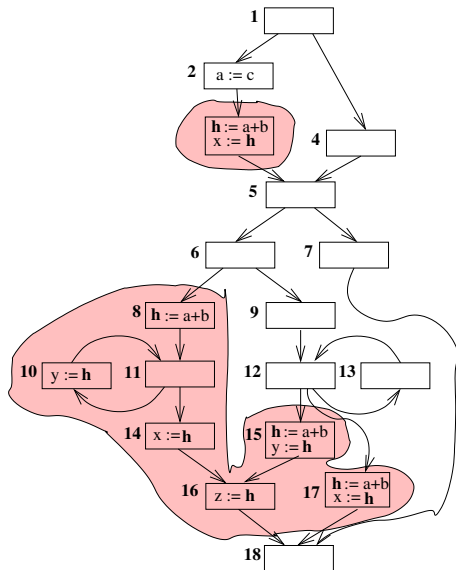


The Result of the *ALCM* Transformation

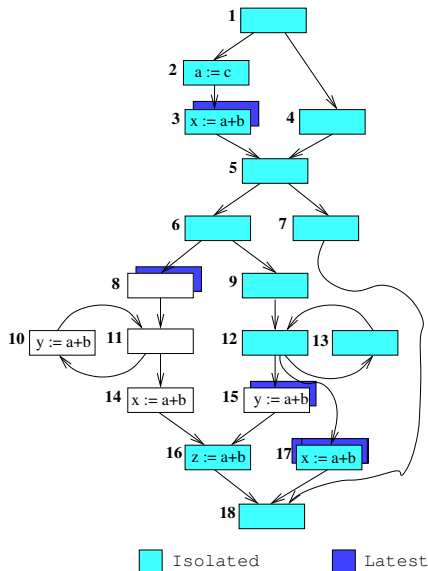


The *ALCM* Transformation: Achievements

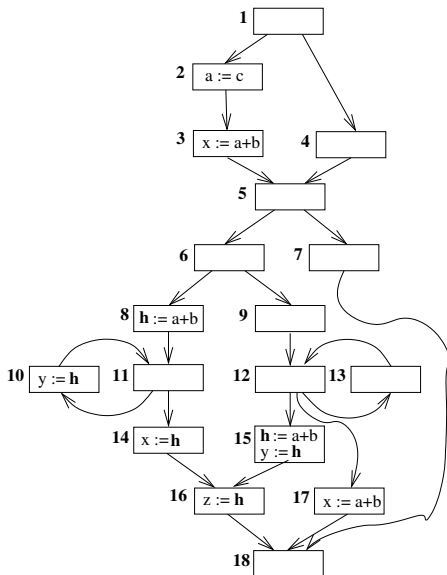
Comp. optimal with **almost** minimum register pressure.



Latest and Isolated Computation Points

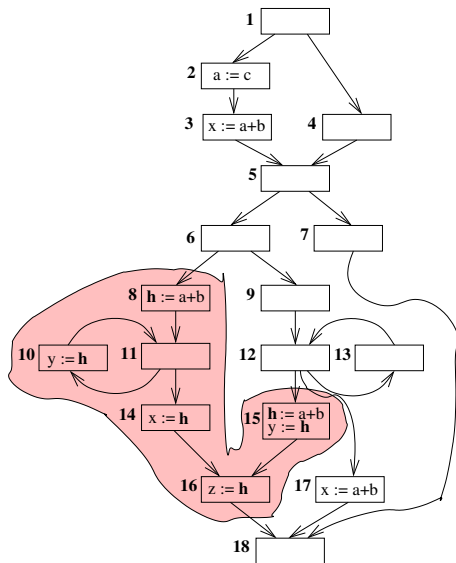


The Result of the *LCM* Transformation



The *LCM* Transformation: Achievements

Computationally optimal with minimum register pressure.



Chapter 8.6

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

Chap. 9




Chap. 10

Part III



Chap. 11

861/185



Further Reading for Chapter 8 (1)

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 9.5.3, The Lazy-Code-Motion Problem; Chapter 9.5.5, The Lazy-Code-Motion Algorithm)
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 10.3.2, Code Motion – Lazy Code Motion)
-  Karl-Heinz Drechsler, Manfred P. Stadel. *A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION*. ACM SIGPLAN Notices 28(5):29-38, 1993.

Further Reading for Chapter 8 (2)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.

Further Reading for Chapter 8 (3)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retrospective: Lazy Code Motion*. In “20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection”, ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 13.3, Partial-Redundancy Elimination – Lazy Code Motion)
-  Jean-Baptiste Tristan, Xavier Leroy. *Verified Validation of Lazy Code Motion*. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), 316-326, 2009.

Chapter 9

Sparse Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

Chap. 10

Part III

865/185

Chapter 9.1

Background and Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.1.1

9.2

9.3

9.4

9.5

9.6

9.7

Chap. 10

Part III

866/185

Recall

Code motion aims at

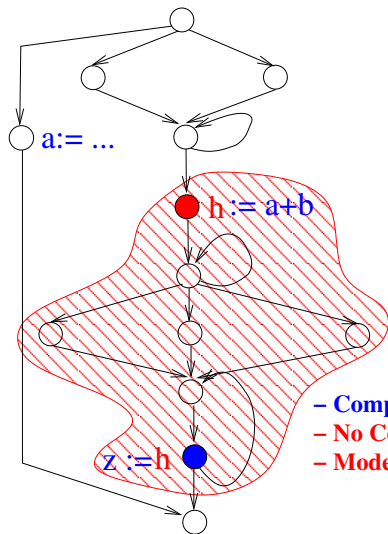
- ▶ eliminating unnecessary recomputations of values (e.g., *BCM*, *ALCM*, *LCM*)
- ▶ while simultaneously avoiding introducing unnecessary register pressure (e.g., *ALCM*, *LCM*)

Overall, code motion thus primarily aims at

- ▶ improving the runtime performance of a program.

However, there is more than Speed!

...code size, for example.



- **Computationally Optimal.**
- **No Code Replication.**
- **Moderate Register Pressure!**

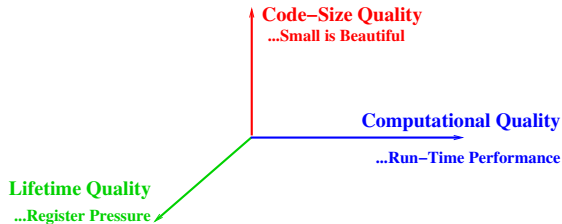
Prioritization of Optimization Goals

Recall that

▶ number of computations, register pressure, code size can not be fully optimized at the same time (cf. Chapter 6).

In this chapter

▶ we present a CM algorithm taking user priorities into account!

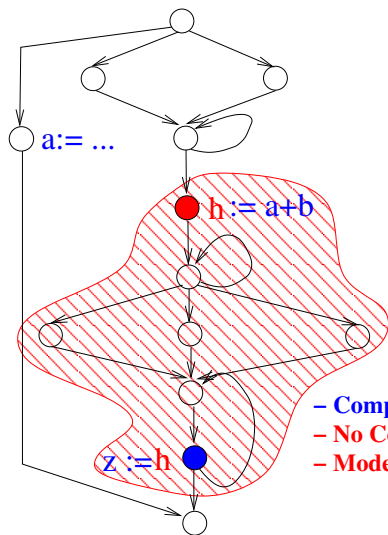


This algorithm, called Sparse Code Motion (*SpCM*)

▶ evolves as a modular extension of the *LCM* transf.

Sparse Code Motion

...can achieve the below result, if so desired:



- **Computationally Optimal.**
- **No Code Replication.**
- **Moderate Register Pressure!**

Chapter 9.1.1

The Embedded Systems Market

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.1.1

9.2

9.3

9.4

9.5

9.6

9.7

Chap. 10

Part III

871/185

The World Market for Microprocessors in 1999

Chip Category	Sold Processors
Embedded 4-bit	2000 Millions
Embedded 8-bit	4700 Millions
Embedded 16-bit	700 Millions
Embedded 32-bit	400 Millions
DSP	600 Millions
Desktop 32/64-bit	150 Millions

...[David Tennenhouse](#) (Intel Director of Research), key note lecture at the *20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, Arizona, December 1999.

The World Market for Microprocessors in 1999

Chip Category	Sold Processors
Embedded 4-bit	2000 Millions
Embedded 8-bit	4700 Millions
Embedded 16-bit	700 Millions
Embedded 32-bit	400 Millions
DSP	600 Millions
Desktop 32/64-bit	150 Millions

~ 2%

...[David Tennenhouse](#) (Intel Director of Research), key note lecture at the *20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, Arizona, December 1999.

Think about

...domain-specific processors used in embedded systems:

- ▶ Telecommunication
 - ▶ Cellular phones, pagers,...
- ▶ Consumer electronics
 - ▶ MP3-players, cameras, game consoles, TVs,...
- ▶ Automotive field
 - ▶ GPS navigation, airbags,...
- ▶ ...

Code for Embedded Systems (1)

...has **high demands** on

- ▶ **Speed** (often real-time demands)
- ▶ **Code size** (system-on-chip, on-chip RAM/ROM)
- ▶ **Power consumption** (batteries)
- ▶ ...

For **embedded systems**

- ▶ **Code size** is often more critical than **speed!**

Code for Embedded Systems (2)

Typically, these demands are still addressed by

- ▶ Assembler programming
- ▶ Manual post-optimization

Shortcomings

- ▶ Error prone
- ▶ Delayed time-to-market

...problems getting more severe with increasing complexity.

Generally, there is

- ▶ a trend towards using high-level languages programming, particularly C, C++.

In View of this Trend

...how do **classical** compiler and optimizer technologies support the specific demands of code for embedded systems?



...only little, unfortunately.

As a Matter of Fact

Classical optimizations

- ▶ are tuned towards performance optimization.
- ▶ are not code-size sensitive.
- ▶ do not allow any control on their impact on the code size.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.1.1

9.2

9.3

9.4

9.5

9.6

9.7

Chap. 10

Part III

878/185

This holds

...for **code motion** based optimizations, too, such as

- ▶ **Partial redundancy elimination** (cf. [Chapter 6](#))
- ▶ **Partial dead-code elimination**
(cf. Lecture Course 185.276 Analysis and Verification)
- ▶ **Partial redundant-assignment elimination**
(cf. Lecture Course 185.276 Analysis and Verification)
- ▶ **Strength reduction** (cf. [Appendix D](#))
- ▶ ...

Recalling the Essence of CM in General

CM can conceptually be considered a **two-stage process**:

1. **Expression Hoisting**

...hoisting computations to 'earlier' safe computation points.

2. **Totally Redundant Expression Elimination**

...eliminating computations, which become totally redundant by expression hoisting

Recalling the Essence of *LCM*

LCM can conceptually be considered the result of a **two-stage process**, too:

1. Hoisting Expressions

...to their 'earliest' safe computation points.

2. Sinking Expressions

...from their 'earliest' safe computation points to their 'latest' safe still computationally optimal computation points.

Chapter 9.2

Running Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

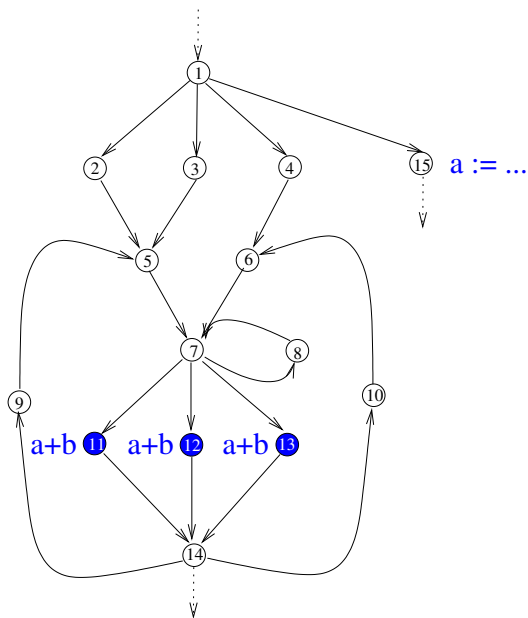
9.7

Chap. 10

Part III

882/185

Running Example: The Original Program



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

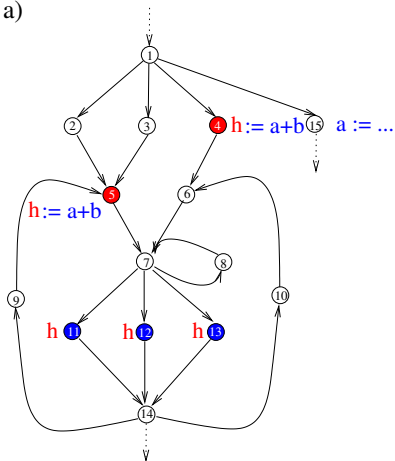
Chap. 10

Part III

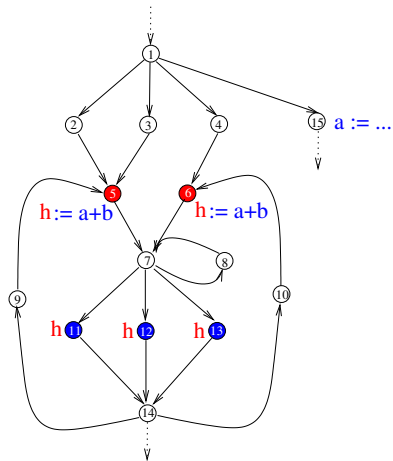
883/185

Running Example: Two Optimization Variants

a)



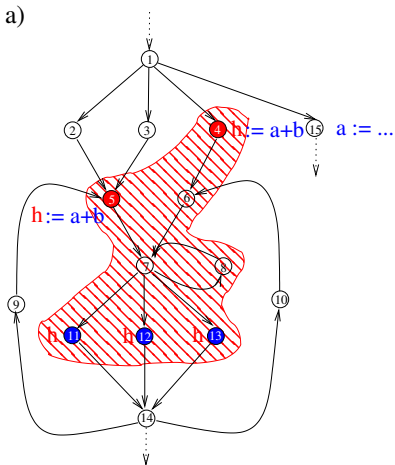
b)



Two Code-size Optimal Programs

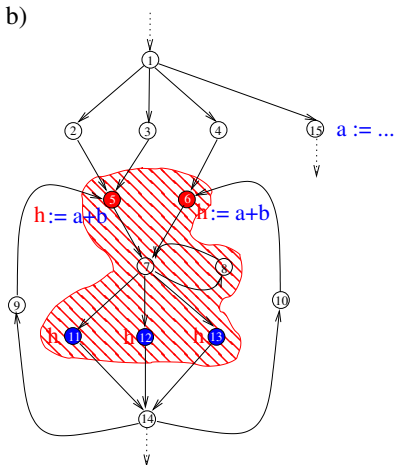
Running Example: Optimization Priorities

a)



SQ > **CQ** > **LQ**

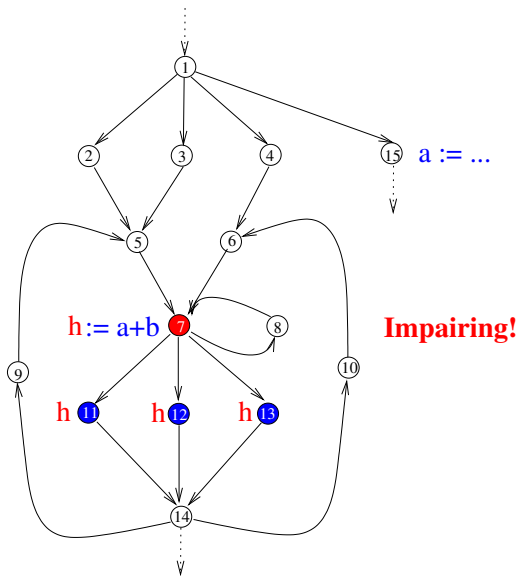
b)



SQ > **LQ** > **CQ**

Running Example: Undesired Transformation

Recall: The below transformation is not desired!



Chapter 9.3

Code-size Sensitive Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.4

9.5

9.6

9.7

Chap. 10

887/185

Code-Size Sensitive Code Motion

~> Problem:

...how do we get **code-size minimal** placement of the computations, i.e., a placement that

- ▶ is **admissible** (semantics & performance preserving).
- ▶ requires **at most as many static computations** as any other admissible code motion transformation.

~> Solution: A new View to Code Motion

...consider **code motion** as a **trade-off** problem: Exchange original computations for newly inserted ones!

~> Clou: Use Graph Theory!

...reduce the **trade-off** problem to the computation of **tight sets** in **bipartite graphs** based on **maximum matchings**!

We postpone but keep in mind

...that we have to **answer**:

- ▶ Where are computations to be inserted and where are original computations to be replaced?

...and to **prove**:

- ▶ Why is this correct (i.e., semantics preserving)?
- ▶ What is the impact on the code size?
- ▶ Why is this 'optimal' wrt a given prioritization of goals?

For each of these questions we will provide a specific theorem that yields the corresponding answer!

Chapter 9.3.1

Graph-theoretical Preliminaries

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.4

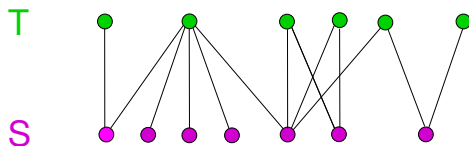
9.5

9.6

9.7

Chap. 10

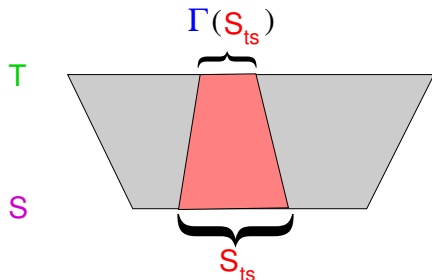
Bipartite Graphs



Tight Set

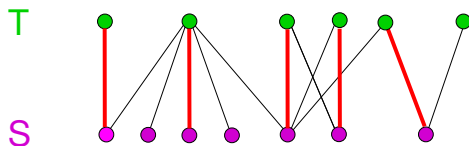
...of a bipartite graph $(S \cup T, E)$: Subset $S_{ts} \subseteq S$ w/

$$\forall S' \subseteq S. |S_{ts}| - |\Gamma(S_{ts})| \geq |S'| - |\Gamma(S')|$$



Two Variants: (1) Largest tight sets (2) Smallest tight sets

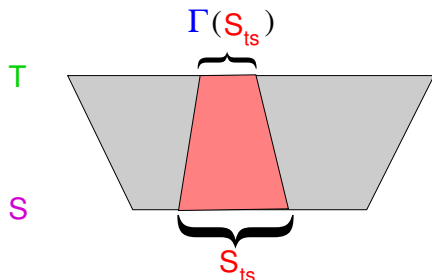
Bipartite Graphs / Maximum Matching



Tight Set

...of a bipartite graph $(S \cup T, E)$: Subset $S_{ts} \subseteq S$ w/

$$\forall S' \subseteq S. |S_{ts}| - |\Gamma(S_{ts})| \geq |S'| - |\Gamma(S')|$$



Two Variants: (1) Largest tight sets (2) Smallest tight sets

Obviously

...we can make use of **off-the-shelf algorithms** from graph theory in order to compute

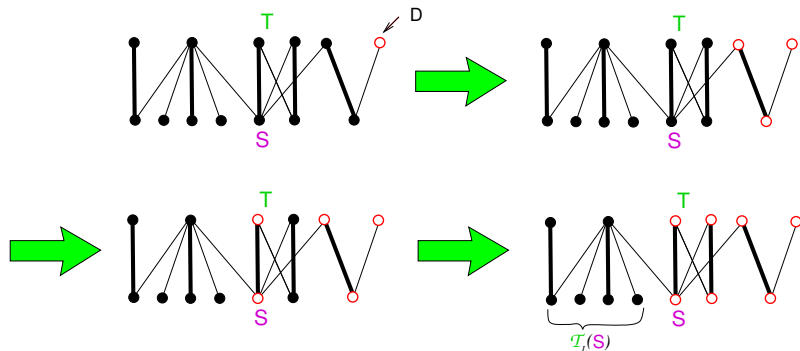
- ▶ **Maximum matchings** and
- ▶ **Tight sets**

This way the **code motion** problem boils down to

- ▶ **constructing** the **bipartite graph** that models the problem!

Computing Largest/Smallest Tight Sets

...based on maximum matchings:



Computing Largest Tight Sets

Algorithm 9.3.1.1 (Computing Largest Tight Sets)

Input: A bipartite graph $(S \dot{\cup} T, E)$, a maximum matching M .

Output: The largest tight set $\mathcal{T}_{LaTS}(S) \subseteq S$.

$S_M := S$; $D := \{t \in T \mid t \text{ is unmatched}\}$;

WHILE $D \neq \emptyset$ DO

 choose some $x \in D$; $D := D \setminus \{x\}$;

 IF $x \in S$

 THEN $S_M := S_M \setminus \{x\}$;

$D := D \cup \{y \mid \{x, y\} \in M\}$

 ELSE $D := D \cup (\Gamma(x) \cap S_M)$

 FI

OD;

$\mathcal{T}_{LaTS}(S) := S_M$

Computing Smallest Tight Sets

Algorithm 9.3.1.2 (Computing Smallest Tight Sets)

Input: A bipartite graph $(S \dot{\cup} T, E)$, a maximum matching M .

Output: The smallest tight set $\mathcal{T}_{SmTS}(S) \subseteq S$.

$S_M := \emptyset$; $A := \{s \in S \mid s \text{ is unmatched}\}$;

WHILE $A \neq \emptyset$ DO

 choose some $x \in A$; $A := A \setminus \{x\}$;

 IF $x \in S$

 THEN $S_M := S_M \cup \{x\}$;

$A := A \cup (\Gamma(x) \setminus S_M)$

 ELSE $A := A \cup \{y \mid \{x, y\} \in M\}$

 FI

OD;

$\mathcal{T}_{SmTS}(S) := S_M$

Chapter 9.3.2

Modelling the Problem

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.4

9.5

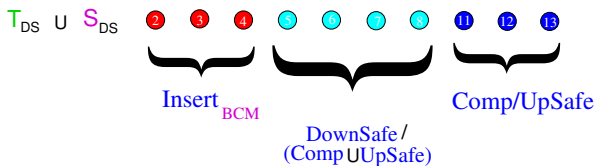
9.6

9.7

Chap. 10

Modelling the Trade-off Problem

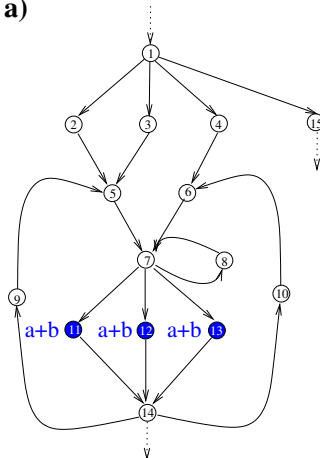
The set of nodes



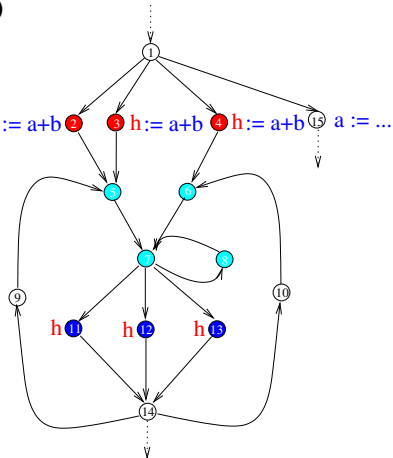
The set of edges...

Illustrating the Set of Nodes

a)



b)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.4

9.5

9.6

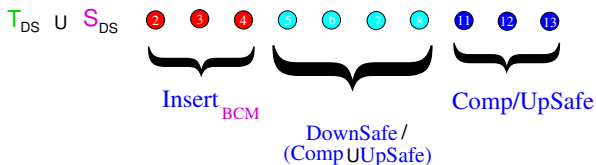
9.7

Chap. 10

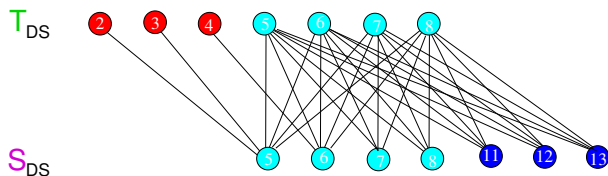
899/185

Modelling the Trade-off Problem

The set of nodes



The bipartite graph



The set of edges

$$\dots \forall n \in S_{DS} \forall m \in T_{DS}. \\ \{n, m\} \in E_{DS} \iff_{df} m \in \text{Closure}(\text{pred}(n))$$

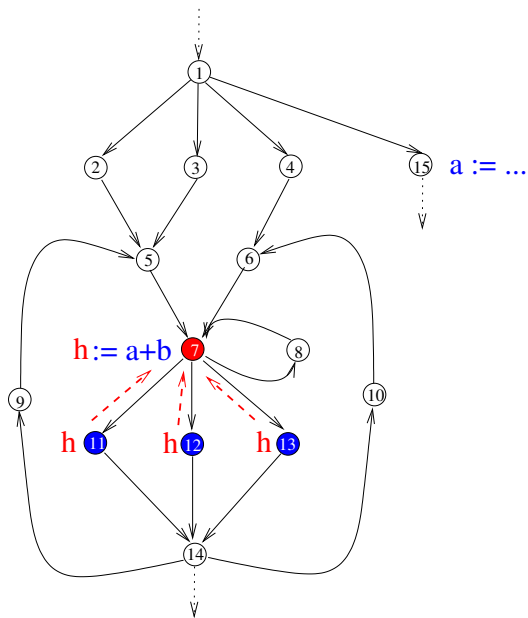
Down-Safety Closures

Definition 9.3.2.1 (Down-Safety Closure)

Let $n \in \text{DownSafe}/\text{UpSafe}$. Then the **down-safety closure** $\text{Closure}(n)$ is the smallest set of nodes such that

1. $n \in \text{Closure}(n)$
2. $\forall m \in \text{Closure}(n) \setminus \text{Comp. succ}(m) \subseteq \text{Closure}(n)$
3. $\forall m \in \text{Closure}(n). \text{pred}(m) \cap \text{Closure}(n) \neq \emptyset \Rightarrow \text{pred}(m) \setminus \text{UpSafe} \subseteq \text{Closure}(n)$

Down-Safety Closures: The Intuition (1)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.4

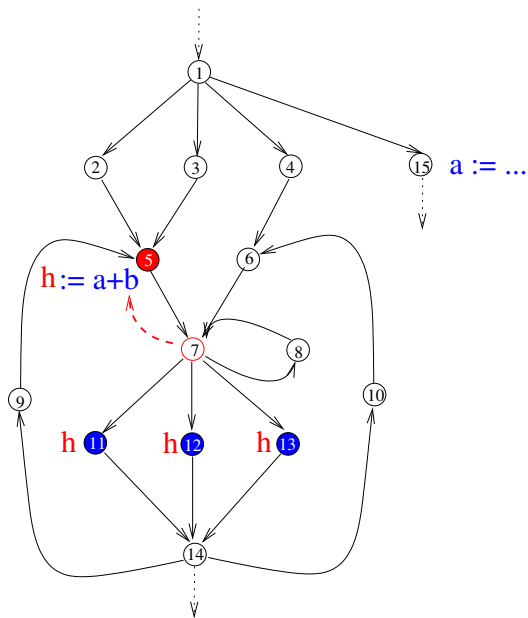
9.5

9.6

9.7

Chap. 10

Down-Safety Closures: The Intuition (2)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.4

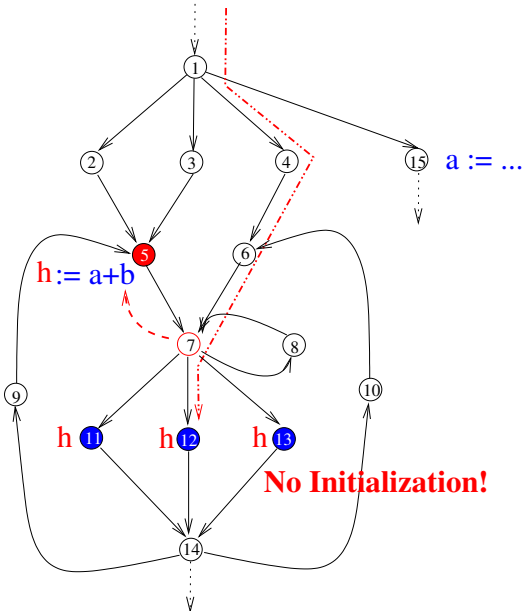
9.5

9.6

9.7

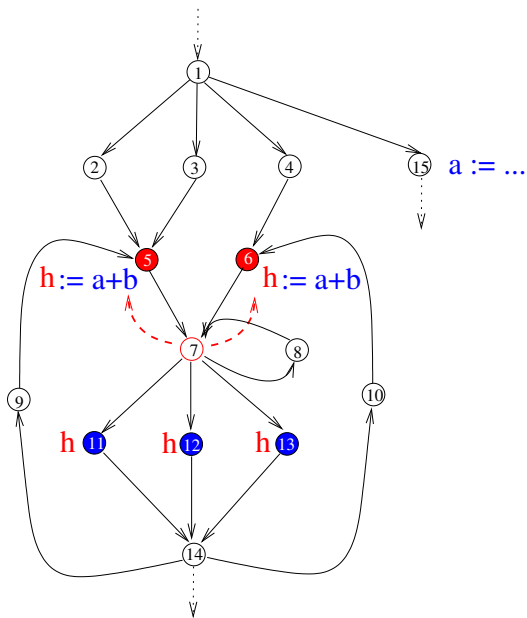
Chap. 10

Down-Safety Closures: The Intuition (3)



- Contents
- Part I
 - Chap. 1
 - Chap. 2
- Part II
 - Chap. 3
 - Chap. 4
 - Chap. 5
 - Chap. 6
 - Chap. 7
 - Chap. 8
 - Chap. 9
 - 9.1
 - 9.2
 - 9.3
 - 9.3.1
 - 9.3.2**
 - 9.3.3
 - 9.4
 - 9.5
 - 9.6
 - 9.7
- Chap. 10

Down-Safety Closures: The Intuition (4)



This intuition

...is condensed in the notion of **down-safety closures**:

Definition 9.3.2.1 (Down-Safety Closure) – recalled

Let $n \in \text{DownSafe/UpSafe}$. Then the **down-safety closure** $\text{Closure}(n)$ is the smallest set of nodes such that

1. $n \in \text{Closure}(n)$
2. $\forall m \in \text{Closure}(n) \setminus \text{Comp}. \text{succ}(m) \subseteq \text{Closure}(n)$
3. $\forall m \in \text{Closure}(n). \text{pred}(m) \cap \text{Closure}(n) \neq \emptyset \Rightarrow \text{pred}(m) \setminus \text{UpSafe} \subseteq \text{Closure}(n)$

Down-Safety Regions

...lead to a characterization of semantics-preserving code motion transformations via their insertion points.

Definition 9.3.2.2 (Down-Safety Region)

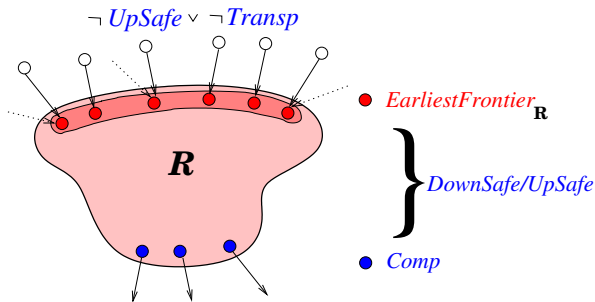
A set $\mathcal{R} \subseteq N$ of nodes is a **down-safety region** iff

1. $Comp \setminus UpSafe \subseteq \mathcal{R} \subseteq DownSafe \setminus UpSafe$
2. $Closure(\mathcal{R}) = \mathcal{R}$

Fundamental

Theorem 9.3.2.3 (Initialization Theorem)

Initializations of **admissible** code motion transformations are always at the **earliestness frontiers** of **down-safety regions**.



...characterizes exactly the set of **semantics preserving code motion** transformations.

Chapter 9.3.3

Main Results: Correctness and Optimality

The Key Questions

...regarding **correctness** and **optimality**:

1. Where to insert computations, why is it correct?
2. What is the impact on the code size?
3. Why is the result optimal, i.e., code-size minimal?

...**three theorems** will answer one of these questions each.

Main Results / 1st Key Question

Question: Where to insert computations, why is it correct?

Answer: At the earliestness frontier of the DS-region induced by the tight set.

Theorem 9.3.3.1 (Tight Sets: Insertion Points)

Let $TS \subseteq S_{DS}$ be a **tight set**. Then we have:

$$\mathcal{R}_{TS} =_{df} \Gamma(TS) \cup (Comp \setminus UpSafe)$$

is a **down-safety region** w/ $Body_{\mathcal{R}_{TS}} = TS$.

Correctness of the *SpCM* Transformation

- ▶ ...is an immediate corollary of **Theorem 9.3.3.1** and the **Initialization Theorem 9.3.2.3**.

Main Results / 2nd Key Question

Question: What is the impact on the code size?

Answer: The difference between the number of inserted and replaced computations.

Theorem 9.3.3.2 (Down-Safety Reg.: Space Gain)

Let \mathcal{R} be a down-safety region with

$$Body_{\mathcal{R}} =_{df} \mathcal{R} \setminus EarliestFrontier_{\mathcal{R}}$$

Then we have:

- ▶ Space Gain by Inserting at $EarliestFrontier_{\mathcal{R}}$:

$$\begin{aligned} & |Comp \setminus UpSafe| - |EarliestFrontier_{\mathcal{R}}| \\ = & |Body_{\mathcal{R}}| - |\Gamma(Body_{\mathcal{R}})| \\ = & defic(Body_{\mathcal{R}}) \end{aligned}$$

Main Results / 3rd Key Question

Question: Why is the result optimal, i.e., code-size minimal?

Answer: Due to a property inherent to tight sets (non-negative deficiency!).

Theorem 9.3.3.3 (Optimality: Transformation)

Let $TS \subseteq S_{DS}$ be a tight set.

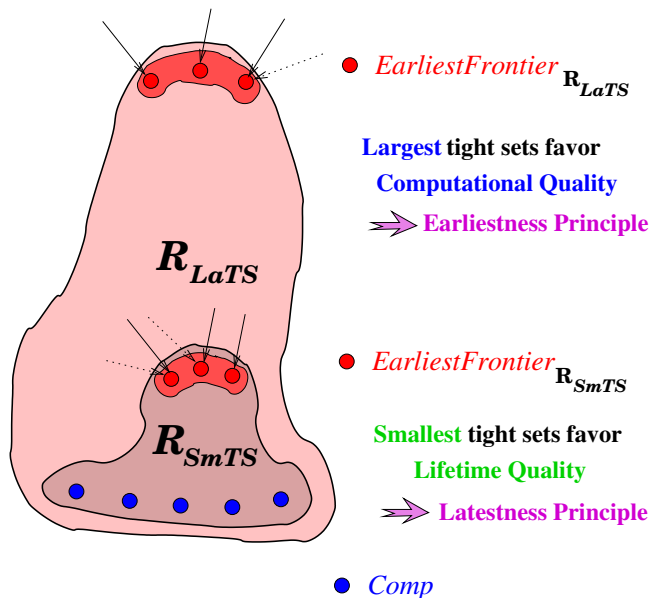
► **Insertion Points:**

$$\text{Insert}_{SpCM} =_{df} \text{EarliestFrontier}_{\mathcal{R}_{TS}} = \mathcal{R}_{TS} \setminus TS$$

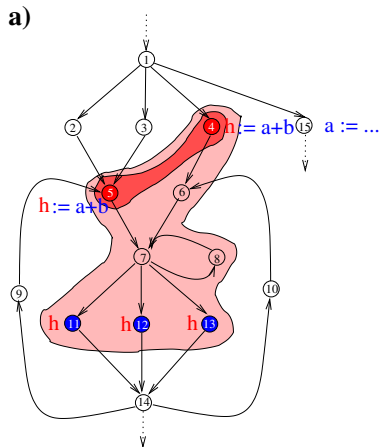
► **Space Gain:**

$$\text{defic}(TS) =_{df} |TS| - |\Gamma(TS)| \geq 0 \text{ max.}$$

Largest vs. Smallest Tight Sets: The Impact



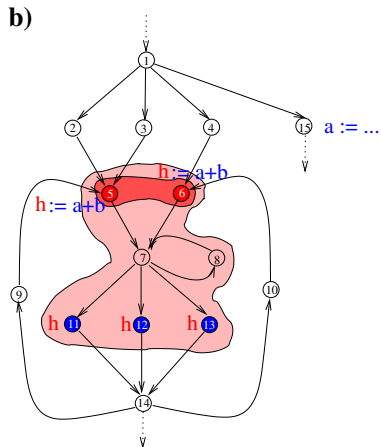
The Impact illustrated on the Running Exam.



Largest Tight Set

(SQ > CQ)

Earliestness Principle



Smallest Tight Set

(SQ > LQ)

Latestness Principle

Chapter 9.4

The *SpCM* Transformation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

Chap. 10

Part III

916/185

The *SpCM* Transformation at a Glance

Preprocess

- **Optional:** Perform **LCM** (3 GEN/KILL-DFAs)
- Compute Predicates of **BCM** for **G** resp. **LCM(G)** (2 GEN/KILL-DFAs)



Main Process

Reduction Phase

- **Construct Bipartite Graph**
- **Compute Maximum Matching**



Optimization Phase

- **Compute Largest/Smallest Tight Set**
- **Determine Insertion Points**

Chapter 9.5

The Cookbook: Recipes for Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

Chap. 10

Part III

918/185

The Cookbook: CM Recipes for Prioritization

Choice of Priority	Apply	To	Using	Yields	Auxiliary Information Required
$\mathcal{L}Q$	Not meaningful: The identity, i.e., G itself is optimal!				
SQ	Subsumed by $SQ > \mathcal{C}Q$ and $SQ > \mathcal{L}Q$!				
$\mathcal{C}Q$	BCM	G			UpSafe(G), DownSafe(G)
$\mathcal{C}Q > \mathcal{L}Q$	LCM	G		LCM(G)	UpSafe(G), DownSafe(G), Delay(G)
$SQ > \mathcal{C}Q$	SpCM	G	Largest tight set	SpCM _{LTS} (G)	UpSafe(G), DownSafe(G)
$SQ > \mathcal{L}Q$	SpCM	G	Smallest tight set		UpSafe(G), DownSafe(G)
$\mathcal{C}Q > SQ$	SpCM	LCM(G)	Largest tight set		UpSafe(G), DownSafe(G), Delay(G) UpSafe(LCM(G)), DownSafe(LCM(G))
$\mathcal{C}Q > SQ > \mathcal{L}Q$	SpCM	LCM(G)	Smallest tight set		UpSafe(G), DownSafe(G), Delay(G) UpSafe(LCM(G)), DownSafe(LCM(G))
$SQ > \mathcal{C}Q > \mathcal{L}Q$	SpCM	DL(SpCM _{LTS} (G))	Smallest tight set		UpSafe(G), DownSafe(G), Delay(SpCM _{LTS} (G)), UpSafe(DL(SpCM _{LTS} (G))), DownSafe(DL(SpCM _{LTS} (G)))

Chapter 9.6

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

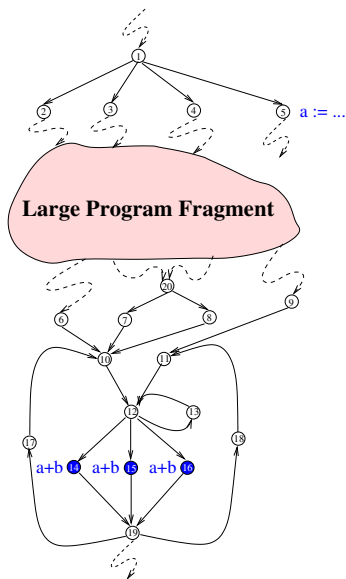
Chap. 10

Part III

920/185

Sparse Code Motion: Flexible and Powerful

The original program:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

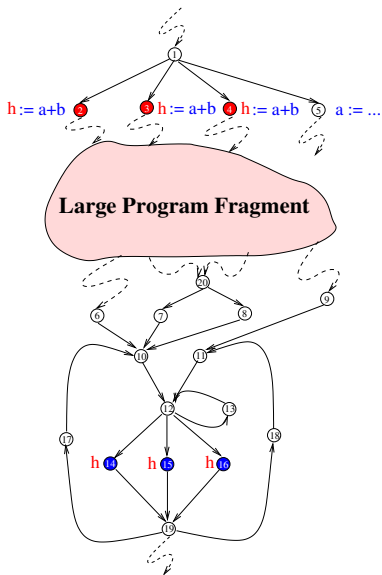
Chap. 10

Part III

921/185

SpCM: Computationally Optimal, 2 DFAs

BCM: A computationally optimal program (\mathcal{CQ})



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

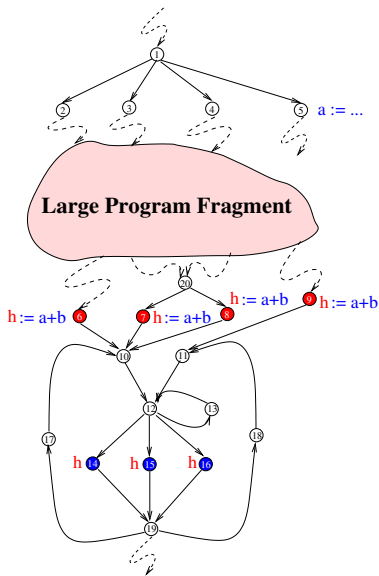
Chap. 10

Part III

922/185

SpCM: Comp. and Lifetime Optimal, 4 DFAs

LCM: A computationally & lifetime opt. program ($CQ > LQ$)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

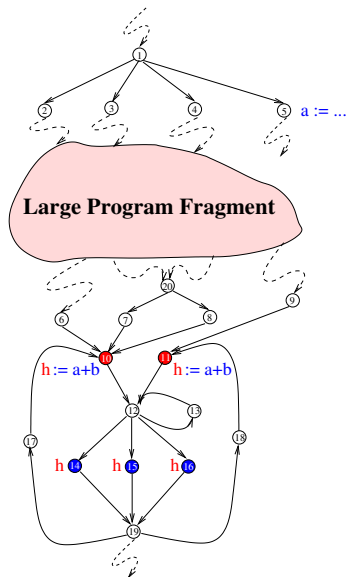
Chap. 10

Part III

923/185

SpCM: Lifetime-Best Code-Size Optimal

SpCM: A code-size & lifetime opt. program ($\mathcal{SQ} > \mathcal{LQ}$)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

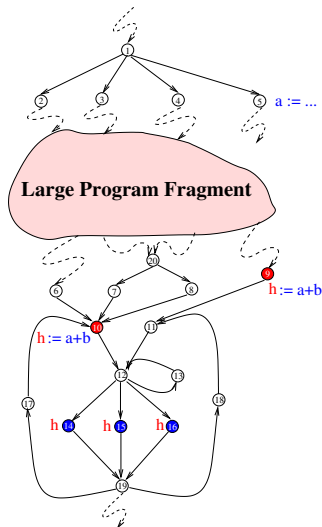
Chap. 10

Part III

924/185

SpCM: Lifet.-Best Comp. Code-Size Optimal

SpCM: A computationally & lifetime best code-size optimal program ($SQ > CQ > LQ$)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

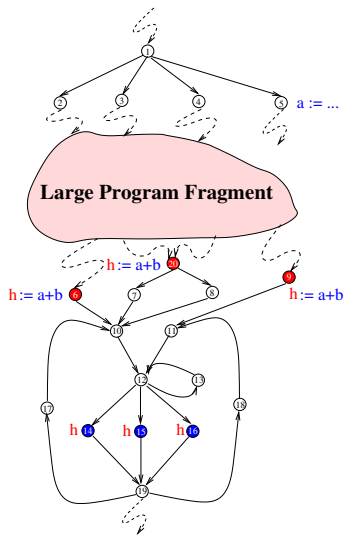
Chap. 10

Part III

925/185

SpCM: Lifet.&Code-Size Best Comp. Optimal

SpCM: A lifetime & code-size best computationally optimal program ($CQ > SQ > LQ$)



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6

9.7

Chap. 10

Part III

926/185

Chapter 9.7

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

9.6



9.7

Chap. 10

Part III

©27/185

Further Reading for Chapter 9

-  Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Sparse Code Motion*. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), 170-183, 2000.
-  Bernhard Scholz, R. Nigel Horspool, Jens Knoop. *Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination*. Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), ACM SIGPLAN Notices 39(7):221-230, 2004.

Chapter 10

Code Motion: Summary, Looking Ahead

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.4

Part III

Chap. 11

Chap. 12

Chapter 10.1

Summary: Roots and Relevance of Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.1.1

10.1.2

10.2

10.3

10.4

Part III

Chap. 11

930/185

Chapter 10.1.1

On the Roots and History of Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.1.1

10.1.2

10.2

10.3

10.4

Part III

Chap. 11

931/185

On the Origins & History of CM (\equiv PRE) (1)

- ▶ 1958: A first glimpse of PRE
 - ↪ Ershov's work on 'On Programming of Arithmetic Operations.'
- ▶ < 1979: Structurally Restricted PRE Techniques
 - ↪ Totally redundant expression elimination (TRE), loop invariant code motion (LICM).
- ▶ 1979: The origin of modern PRE
 - ↪ Morel and Renvoise's groundbreaking work on PRE.
- ▶ < ca. 1992: Heuristic improvements of the PRE algorithm of Morel and Renvoise
 - ↪ Dhamdhere [1988, 1991]; Drechsler, Stadel [1988]; Sorkin [1989]; Dhamdhere, Rosen, Zadeck [1992], Briggs, Cooper [1994],...

On the Origins & History of CM (\equiv PRE) (2)

- ▶ 1992: *BCM* and *LCM* [Knoop Rüthing, Steffen (PLDI'92)]
 - ↪ *BCM* first to achieve computational optimality based on the earliestness principle.
 - ↪ *LCM* first to achieve computational optimality with minimum register. pressure based on the latestness principle
 - ↪ *BCM, LCM* first to be purely unidirectional
 - ↪ first to be rigorously proven correct and optimal.
- ▶ 2000: *SpCM*: The origin of code-size sensitive PRE [Knoop, Rüthing, Steffen (POPL 2000)]
 - ↪ first to be code-size sensitive.
 - ↪ first to allow users prioritization of optimization goals.
 - ↪ rigorously be proven correct and optimal.
 - ↪ first to bridge the gap between compilation for general purpose processors and embedded systems.

On the Origins & History of CM (\equiv PRE) (3)

- ▶ Since ca. 1997: A new strand of research on PRE
 - ↪ Speculative PRE: Gupta, Horspool, Soffa, Xue, Scholz, Knoop,...
- ▶ 2006: A fresh look at PRE (as maximum flow problem)
 - ↪ Unifying PRE and Speculative PRE [Xue, Knoop (CC 2006)]

These days, *lazy code motion* is the

- ▶ *de facto* standard algorithm for PRE used in current state-of-the-art compilers
 - ▶ Gnu compiler family
 - ▶ Sun Sparc compiler family
 - ▶ LLVM
 - ▶ ...

Chapter 10.1.2

On the Relevance of Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.1.1

10.1.2

10.2

10.3

10.4

Part III

Chap. 11

935/185

Code Motion is Relevant and Challenging (1)

Why is [investigating code motion worthwhile](#) and [rewarding](#)?

Because [code motion](#) is

- ▶ [general](#): A family of optimizations rather than a single optimization.
- ▶ [well understood](#): Algorithms, which are [proven correct](#) and [optimal](#).
- ▶ [truly classical](#): Looks back to a long history originated by
 - ▶ Etienne Morel, Claude Renvoise. [Global Optimization by Suppression of Partial Redundancies](#). Communications of the ACM 22(2):96-103, 1979.
 - ▶ Ken Kennedy. [Safety of Code Motion](#). International Journal of Computer Mathematics 3(2-3):117-130, 1972.
 - ▶ Andrei P. Ershov. [On Programming of Arithmetic Operations](#). Communications of the ACM 1(8):3-6, 1958.

Code Motion is Relevant and Challenging (2)

In particular, [code motion](#) is

- ▶ [relevant](#): Widely used in practice because of its power.

Last but not least, [code motion](#) is

- ▶ [challenging](#): Conceptually simple but exhibits a variety of thought provoking phenomenons and pitfalls.

Some of these [challenges](#) we are going to illustrate next.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.1.1

10.1.2

10.2

10.3

10.4

Part III

Chap. 11

937/185

Code Motion Reconsidered

Traditionally:

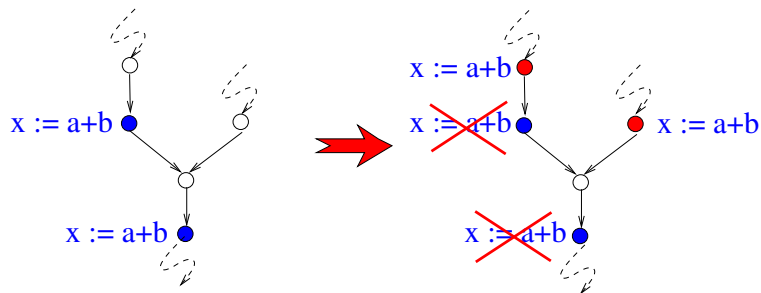
- ▶ Code (C) means expressions.
- ▶ Motion (M) means hoisting.
- ▶ CM means partially redundant expression elimination.

But:

- ▶ CM is more than hoisting of expressions and PR(E)E!

Assignments

...are `code`, too, of course.

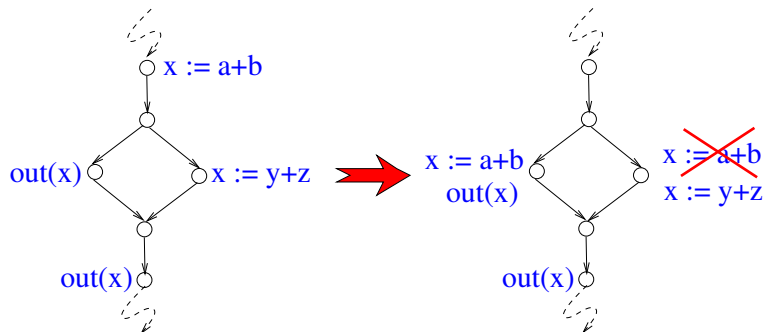


In this example, `code motion` means

- ▶ partially redundant assignment elimination (PRAE).

Assignments

...can be **hoisted** like expressions but conversely, also be **sunk**!



In this example, **code motion** means

- ▶ partially dead assignment (or code) elimination (PDCE).

Design Space of CM Algorithms (1)

In general

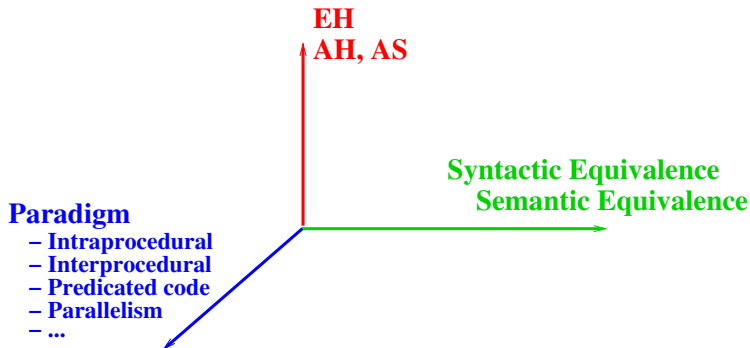
- ▶ Code means expressions/assignments.
- ▶ Motion means hoisting/sinking.

Code / Motion	Hoisting	Sinking
Expressions	EH	n.a.
Assignments	AH	AS

....which spans a first set of dimensions for designing **code** motion algorithms.

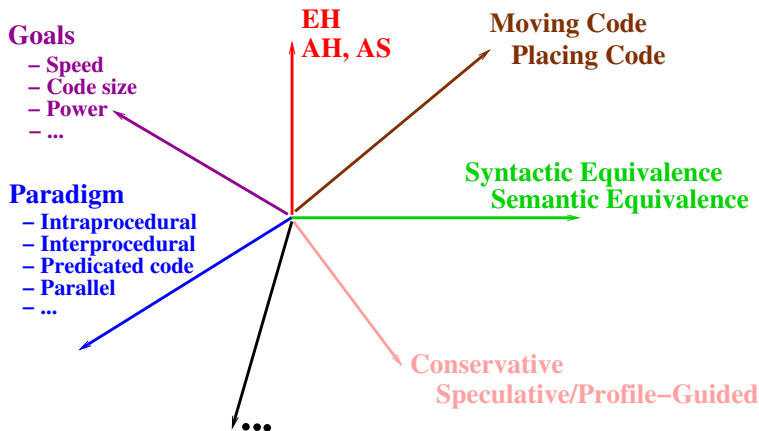
Design Space of CM Algorithms (2)

...but there are more dimensions for the design of code motion algorithms:



Design Space of CM Algorithms (3)

...and even more:



Chapter 10.2

Looking Ahead: Value Numbering

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.2.1

10.2.2

10.3

10.4

Part III

Chap. 11

944/185

Chapter 10.2.1

(Local) Value Numbering

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.2.1

10.2.2

10.3

10.4

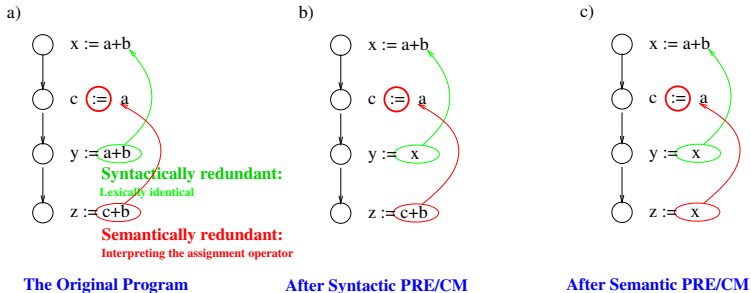
Part III

Chap. 11

(Local) Value Numbering

...eliminating **semantically redundant** computations in basic blocks.

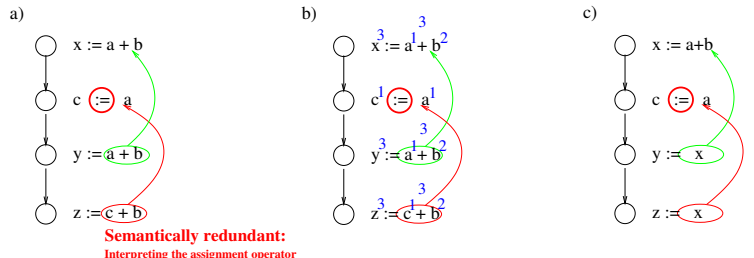
Illustrating Example:



Local Value Numbering

Intuitively

- ▶ value numbering works by assigning terms so-called **value numbers** representing symbolically their values.
- ▶ same value number implies **same value**.



The Original Program

After Value Numbering

After Semantic PRE/CM

...has been described early by

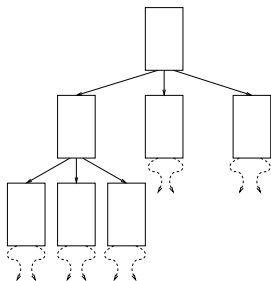
- ▶ John Cocke and Jacob T. Schwartz in 1970.

Local Value Numbering

...can straightforward be extended to

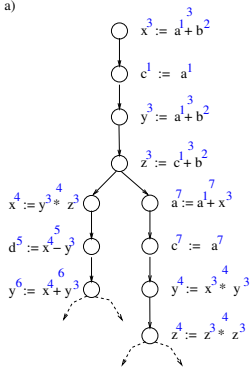
- ▶ extended basic blocks (i.e., trees of basic blocks).

Illustrating Example:



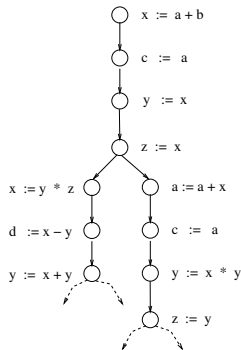
Extended Basic Block – A Tree of Basic Blocks

a)



After Value Numbering

b)



After Semantic PRE/CM

References for Chapter 10.2.1



John Cocke, Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 2nd Revised Version, 771 pages, 1970. (Chapter 6, Optimization Methods for Algebraic Languages)

Chapter 10.2.2

Global Value Numbering: Semantic Code Motion

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.2.1

10.2.2

10.3

10.4

Part III

Chap. 11

950/185

Motivation

Syntactic PRE (or Syntactic Code Motion), e.g., *BCM*, *LCM*

- + **Global**: Works for **whole programs**.
- **Equivalence**: Limited to **lexical identity**.

(Local) Value Numbering

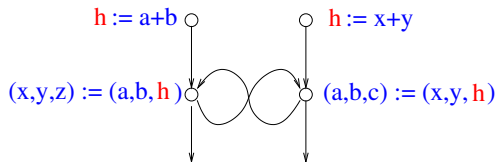
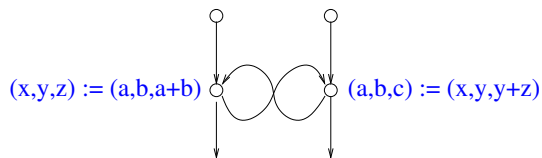
- + **Equivalence**: Captures **semantic equivalence** of terms.
- **Local**: Limited to **basic blocks** (or **basic blocks**)

Global Value Numbering (or Semantic Code Motion)

- + combines the **best features** of **syntactic PRE** and **value numbering**
- + while **avoiding** their **weaknesses**.

Global Value Numbering: Semantic CM

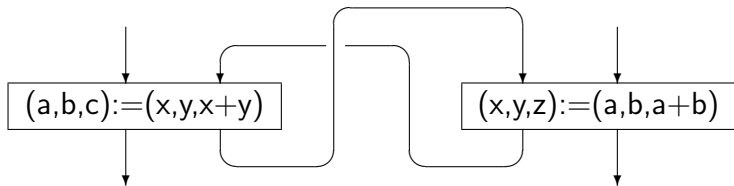
...extending the idea of value numbering to whole programs.



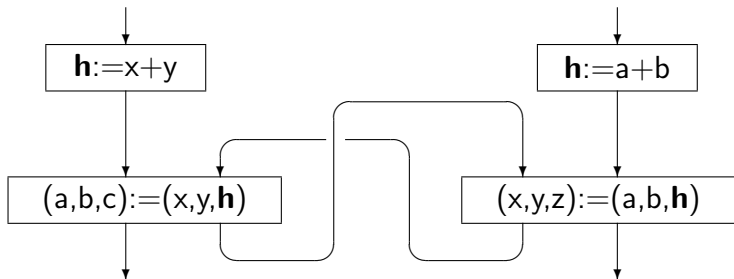
Bernhard Steffen (TAPSOFT'87)

Semantic CM: Illustrating the Essence (1)

The running example:



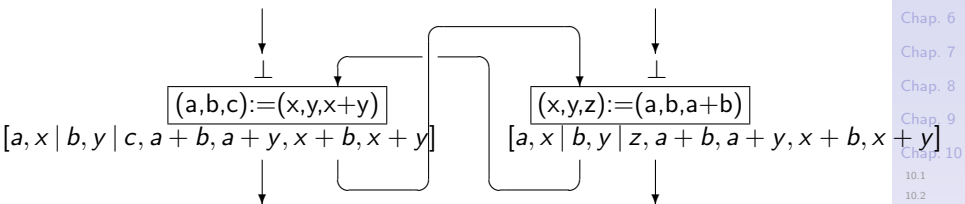
The optimized program:



Semantic CM: Illustrating the Essence (2)

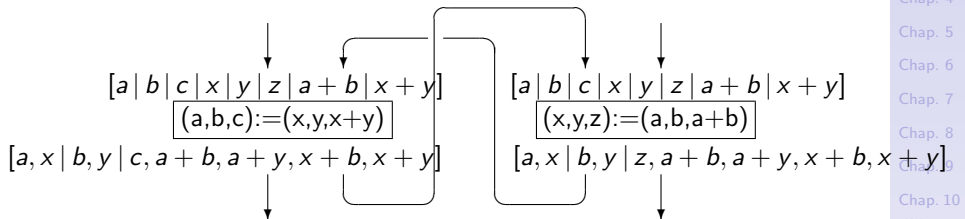
Stage 1: The Analysis Phase

Step 1.1: Determining semantically equivalent terms (wrt the Herbrand interpretation).



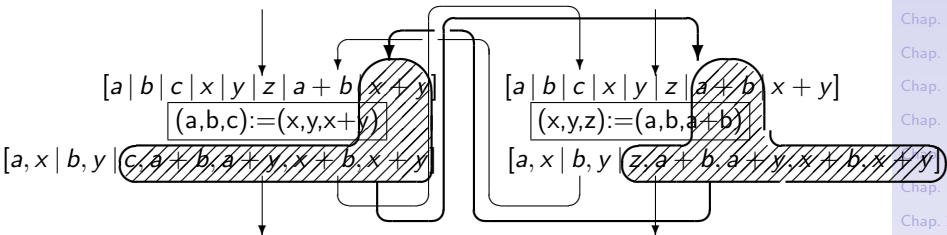
Semantic CM: Illustrating the Essence (3)

Step 1.2: Enlarging the set of term equivalences syntactically represented.



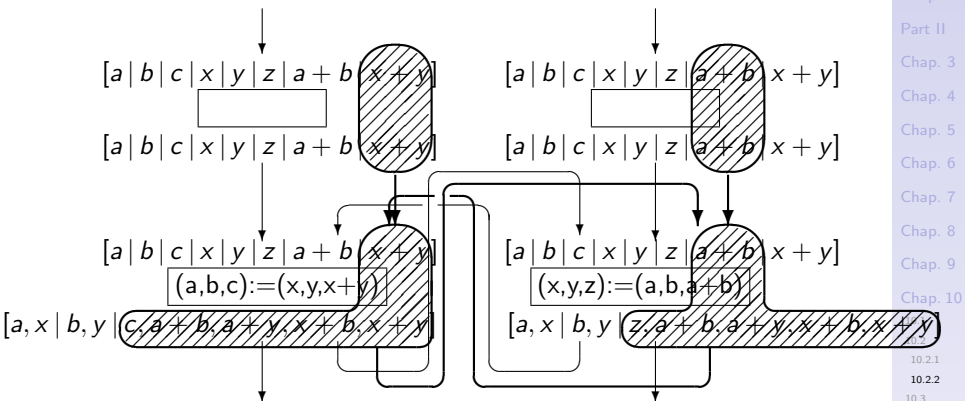
Semantic CM: Illustrating the Essence (4)

Step 1.3: Constructing the **value flow graph**.



Semantic CM: Illustrating the Essence (5)

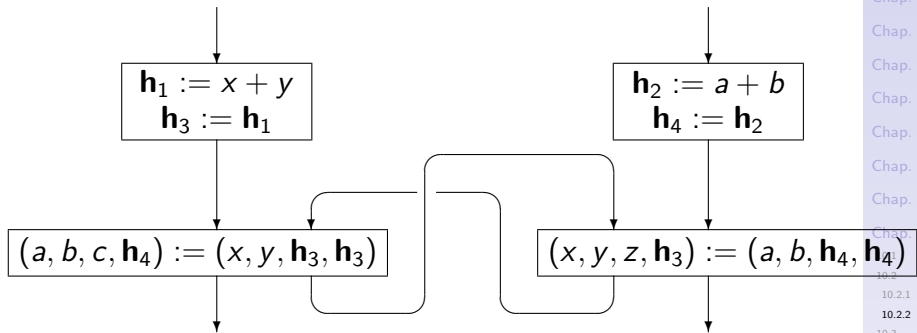
Step 1.3: The **value flow graph**, displaying a larger fragment.



Semantic CM: Illustrating the Essence (6)

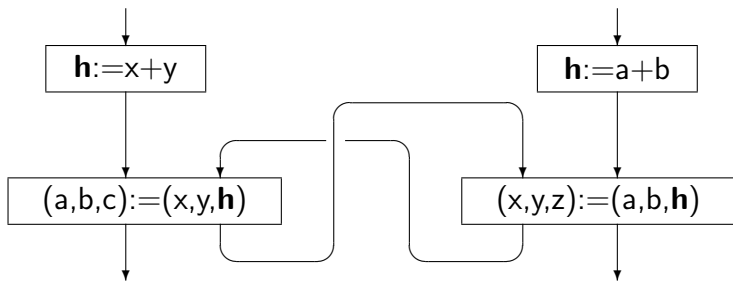
Stage 2: The Transformation Phase

Step 2.1: The Semantic Code Motion Optimization.



Semantic CM: Illustrating the Essence (7)

Step 2.1: Variable subsumption yields the final optimization.



...for details see: Steffen, Knoop, Rütting (ESOP'90).

Semantic CM: Computing Insertion Points (1)

...by analysing the **value flow graph**.

Algorithm 10.2.2.1 (Computing Insertion Points)

The Frame Conditions (Local Properties):

$$\mathbf{ANTLOC}(\nu) \iff \nu \downarrow_1 \cap \mathit{Terms}(\mathcal{N}(\nu)) \neq \emptyset$$

$$\mathbf{AVIN}(\nu) = \mathbf{PPIN}(\nu) = \mathit{false} \text{ if } \nu \in \mathit{VFN}_s$$

$$\mathbf{PPOUT}(\nu) = \mathit{false} \text{ if } \nu \in \mathit{VFN}_e$$

where

- ▶ $\nu \downarrow_1$: the projection of ν to its first component.
- ▶ $\mathcal{N}(\nu)$: the node of the flow graph ν is associated with.
- ▶ $\mathit{VFN}_s, \mathit{VFN}_e$: the set of start and end nodes of the VFG.
- ▶ $\mathit{Terms}(n)$: the set of terms of the assignment at node n .

Semantic CM: Computing Insertion Points (2)

The Fixed Point Equations (Global Properties):

$$\mathbf{AVIN}(\nu) \iff \prod_{\nu' \in \text{pred}(\nu)} \mathbf{AVOUT}(\nu')$$

$$\mathbf{AVOUT}(\nu) \iff \mathbf{AVIN}(\nu) \vee \mathbf{PPOUT}(\nu)$$

$$\mathbf{PPIN}(\nu) \iff \mathbf{AVIN}(\nu) \wedge (\mathbf{ANTLOC}(\nu) \vee \mathbf{PPOUT}(\nu))$$

$$\mathbf{PPOUT}(\nu) \iff \prod_{m \in \text{succ}(\mathcal{N}(\nu))} \sum_{\substack{\nu' \in \text{succ}(\nu) \\ \mathcal{N}(\nu') = m}} \mathbf{PPIN}(\nu')$$

The Insertion Points:

$$\mathbf{INSERT}(\nu) =_{df} \mathbf{PPOUT}(\nu) \wedge \neg \mathbf{PPIN}(\nu)$$

Semantic CM: Main Results

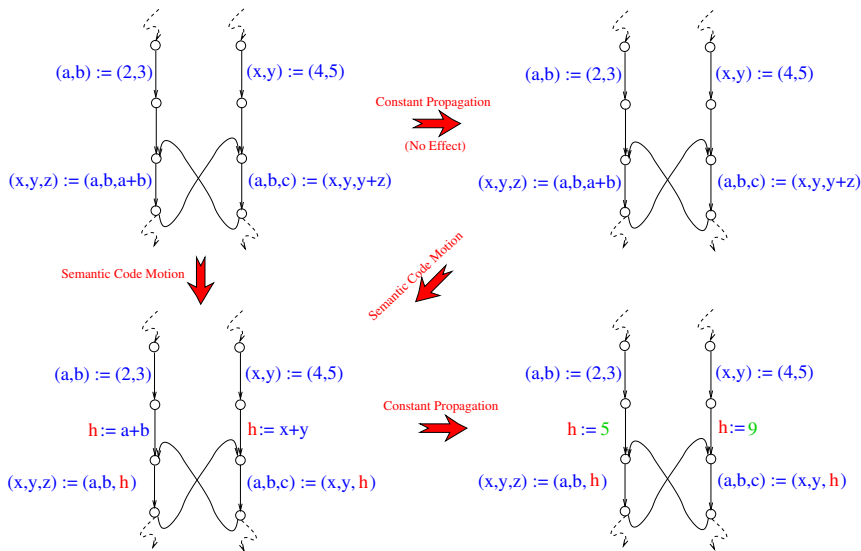
Theorem 10.2.2.2 (Optimality of Analysis)

Given an arbitrary flow graph, the analysis stage terminates with a flow graph annotation which exactly characterizes all equivalences of program terms wrt the [Herbrand interpretation](#).

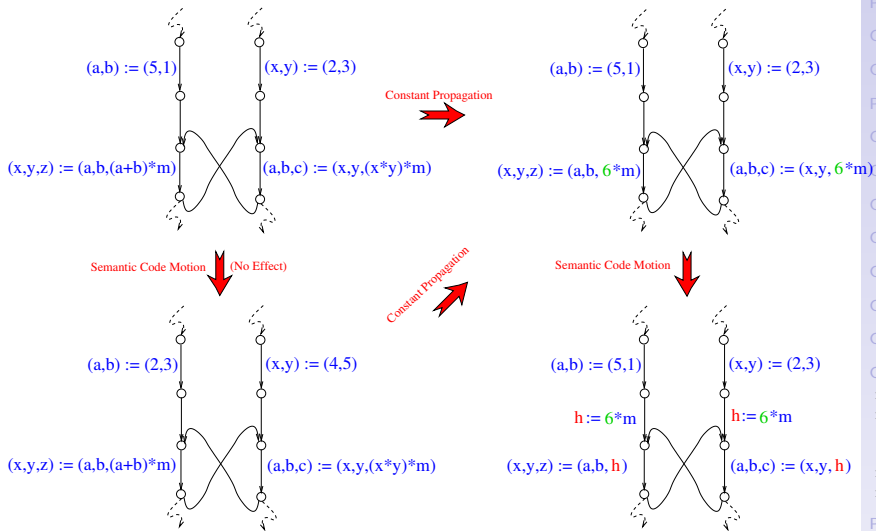
Theorem 10.2.2.3 (Optimality of Transformation)

Every flow graph transformed by the two stage algorithm (in the full variant) is [Herbrand optimal](#).



Recall: Interdependencies of SCM and CP (1)



Recall: Interdependencies of SCM and CP (2)



References for Chapter 10.2.2

-  Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Proceedings of the 2nd Joint Conference on Theory and Practice of Software Development (TAPSOFT'87), Springer-V., LNCS 249, 52-68, 1987.
-  Bernhard Steffen, Jens Knoop, Oliver Rüthing. *The Value Flow Graph: A Program Representation for Optimal Program Transformations*. In Proceedings of the 3rd European Symposium on Programming (ESOP'90), Springer-V., LNCS 432, 389-405, 1990.

Chapter 10.3

Looking Ahead: Challenges and Pitfalls

The Impact of Setting Changes

...on **safety** and **optimality**.

Safety and **optimality statements** are quite sensitive for **setting changes**!

Three **examples** shall provide evidence for this:

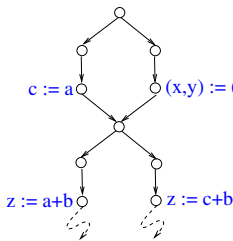
- ▶ **Code motion** vs. **code placement**.
- ▶ **Interdependencies** of elementary transformations.
- ▶ **Paradigm** dependencies.

Chapter 10.3.1

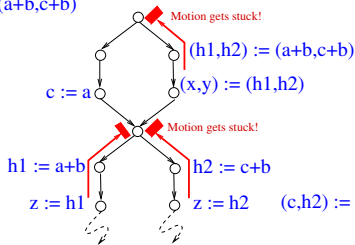
The Impact of Moving or Placing Code

Code Motion (CM) vs. Code Placement (CP)

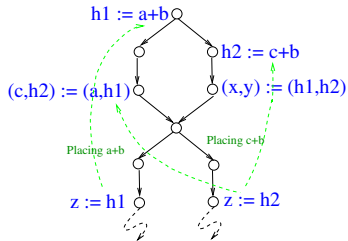
...CM and CP are no synonyms!



Original Program



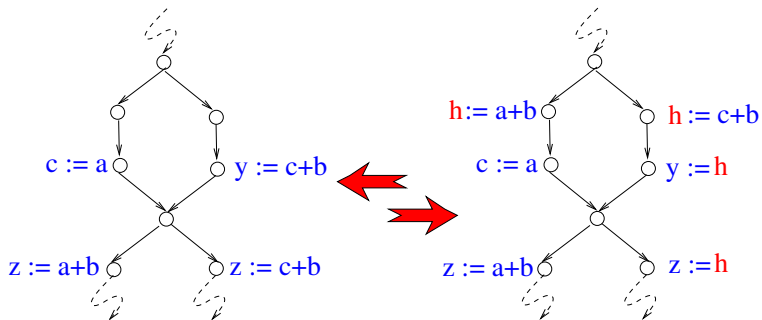
After Sem. Code Motion



After Sem. Code Placement

Even worse

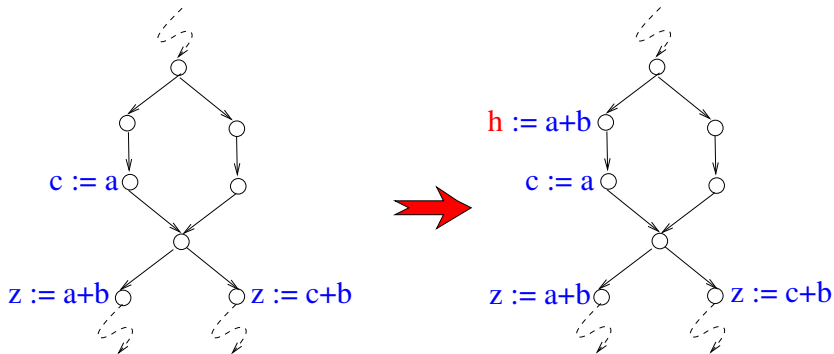
Optimality is lost!



Incomparable!

Even worse

The **performance** can be **impaired**, when applied naively!



References for Chapter 10.3.1

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Code Motion and Code Placement: Just Synonyms?* In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 154-169, 1998.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Expansion-based Removal of Semantic Partial Redundancies.* In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.

Chapter 10.3.2

The Impact of Interacting Transformations

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

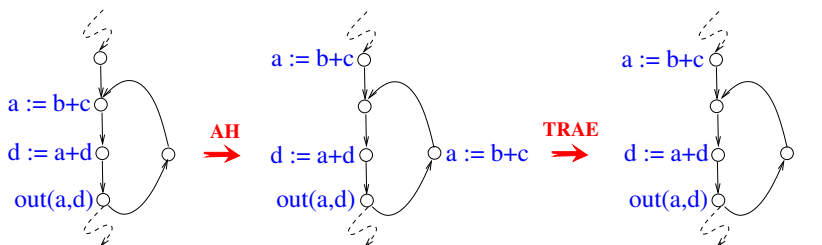
10.4

Part III

973/185

Assignment Hoisting (AH) plus Totally Redundant Assignment Elimination (TRAЕ)

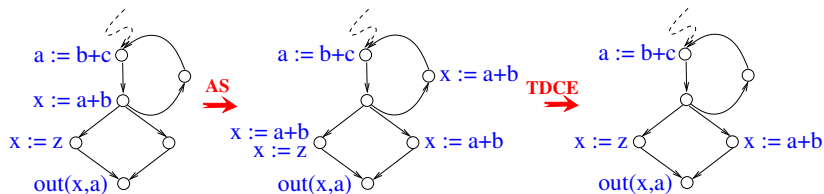
...leads to Partially Redundant Assignment Elimination (PRAE):



...2nd Order Effects!

Assignment Sinking (AS) plus Total Dead-Code Elimination (TDCE)

...leads to Partial Dead-Code Elimination (PDCE):



...2nd Order Effects!

Conceptually

...**PREE**, **PRAE**, and **PDCE** can be understood as follows:

- ▶ **PREE** = EH ; TREE
- ▶ **PRAE** = (AH + TRAE)*
- ▶ **PDCE** = (AS + TDCE)*

Optimality Results for PREE

Theorem 10.3.2.1 (Optimality)

1. *BCM* yields computationally optimal results.
2. *LCM* yields computationally and lifetime optimal results.
3. *SpCM* yields optimal results wrt a given prioritization of the goals of redundancy elimination and register pressure and code size minimization.

Optimality Results for (Pure) PRAE/PDCE

Deriving relation $\vdash \dots$

▶ PRAE... $G \vdash_{AH, TRAE} G'$ ($ET = \{AH, TRAE\}$)

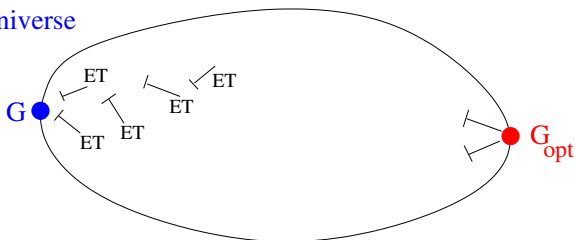
▶ PDCE... $G \vdash_{AS, TDCE} G'$ ($ET = \{AS, TDCE\}$)

We can prove:

Theorem 10.3.2.2 (Optimality)

For PRAE and PDCE the deriving relation \vdash_{ET} is **confluent** and **terminating**.

Universe



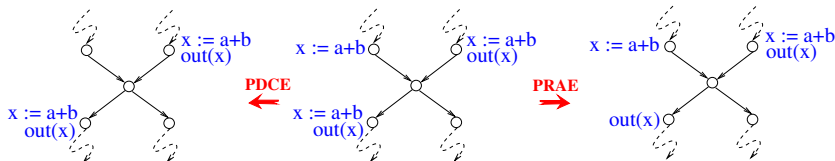
Now

...extend and amalgamate **PRAE** and **PDCE** to **Assignment Placement (AP)**:

$$\blacktriangleright AP = (AH + TRAE + AS + TDCE)^*$$

...**AP** should be more powerful than **PRAE** and **PDCE** alone!

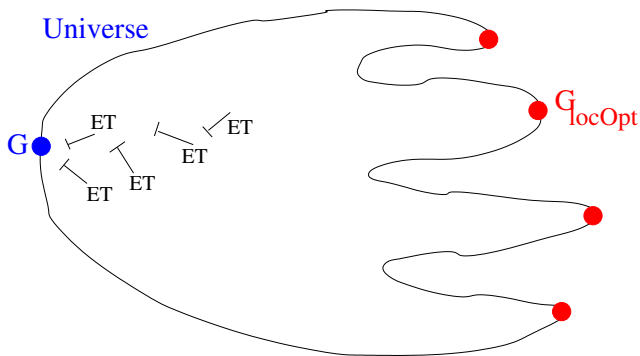
Indeed, it is but:



The resulting two programs are **incomparable**.

Confluence

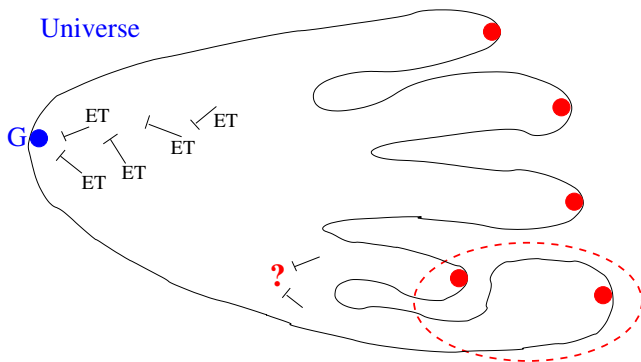
...hence, (global) optimality is lost!



Fortunately, we retain local optimality!




However

...there are [language settings](#) (e.g., [High-Performance Fortran](#)), where we end up w/ [universes](#) like the following:



Here, even [local optimality](#) is lost!

References for Chapter 10.3.2

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Partial Dead Code Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):147-158, 1994.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *The Power of Assignment Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), ACM SIGPLAN Notices 30(6):233-245, 1995.
-  Jens Knoop, Eduard Mehofer. *Optimal Distribution Assignment Placement*. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Springer-V., LNCS 1300, 364 - 373, 1997.

Chapter 10.3.3

The Impact of Paradigm Shifts

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

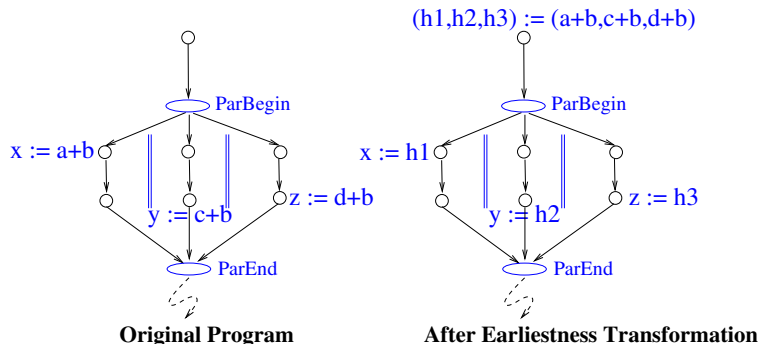
10.3.3

10.4

Part III

Adding Parallelism

...analysing and optimizing parallel programs.



...naively transferring the strategy of 'placing computations as early as possible' leads here to an essentially sequential program!

Adding Procedures

...interprocedural analysis and optimization.



Similar phenomena are encountered when naively transferring successful transformation strategies

- ▶ from the intraprocedural
- ▶ to the interprocedural

setting, e.g., the optimal PRE placement strategies of

- ▶ Busy Code Motion
- ▶ Lazy Code Motion

References for Chapter 10.3.3

-  Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998. (Chapter 10, Interprocedural Code Motion: The Transformations; Chapter 10.1, Essential Differences to the Intraprocedural Setting)
-  Jens Knoop, Bernhard Steffen. *Code Motion for Explicitly Parallel Programs*. In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), ACM SIGPLAN Notices 34(8):13-24, 1999.

Chapter 10.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.4

Part III



Chap. 11

Chap. 12


Further Reading for Chapter 10 (1)

Syntactic PRE



Pioneering, Groundbreaking

-  Ken Kennedy. *Safety of Code Motion*. International Journal of Computer Mathematics 3(2-3):117-130, 1972.
-  Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.

Lazy Code Motion and More

-  Ras Bodik, Rajiv Gupta. *Register Pressure Sensitive Redundancy Elimination*. In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 107-121, 1999.




Further Reading for Chapter 10 (2)

-  Preston Briggs, Keith D. Cooper. *Effective Partial Redundancy Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):159-170, 1994.
-  Keith D. Cooper, Jason Eckhardt, Ken Kennedy. *Redundancy Elimination Revisited*. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008), 12-21, 2008.


Further Reading for Chapter 10 (3)

-  Fred C. Chow, Sun Chan, Robert Kennedy, Shing-Ming Liu, Raymond Lo, Peng Tu. *A New Algorithm for Partial Redundancy Elimination based upon SSA Form*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM SIGPLAN Notices 32(5):273-286, 1997.
-  Robert Kennedy, Sun Chan, Shing-Ming Liu, Raymond Lo, Peng Tu, Fred C. Chow. *Partial Redundancy Elimination in SSA Form*. ACM Transactions of Programming Languages and Systems 32(3):627-676, 1999.


Further Reading for Chapter 10 (4)

-  Dhananjay M. Dhamdhere. *E-path_pre: Partial Redundancy Elimination Made Easy*. ACM SIGPLAN Notices 37(8):53-65, 2002.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.


Further Reading for Chapter 10 (5)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retrospective: Lazy Code Motion*. In “20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection”, ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.


Code-Size Sensitive Code Motion

-  Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Sparse Code Motion*. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), 170-183, 2000.


Further Reading for Chapter 10 (6)

-  Bernhard Scholz, R. Nigel Horspool, Jens Knoop. *Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination*. Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), ACM SIGPLAN Notices 39(7):221-230, 2004.


Complete Removal of Redundancies

-  Ras Bodik, Rajiv Gupta, Mary Lou Soffa. *Complete Removal of Redundant Expressions*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), ACM SIGPLAN Notices 33(5):1-14, 1998.

Further Reading for Chapter 10 (7)




-  Bernhard Steffen. *Property-Oriented Expansion*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 22-41, 1996.

Code Motion for Parallel Programs

-  Jens Knoop, Bernhard Steffen. *Code Motion for Explicitly Parallel Programs*. In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), ACM SIGPLAN Notices 34(8):13-24, 1999.

Further Reading for Chapter 10 (8)

Speculative Code Motion


-  Qiong Cai, Jingling Xue. *Optimal and Efficient Speculation-based Partial Redundancy Elimination*. In Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2003), 91-104, 2003.
-  Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Redundancy Elimination Using Speculation*. In Proceedings of the 1998 International Conference on Computer Languages (ICCL'98), 230-239, 1998.
-  R. Nigel Horspool, H. C. Ho. *Partial Redundancy Elimination Driven by a Cost-benefit Analysis*. In Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering (CSSE'97), 111-118, 1997.

Further Reading for Chapter 10 (9)

-  R. Nigel Horspool, David J. Pereira, Bernhard Scholz. *Fast Profile-based Partial Redundancy Elimination*. In Proceedings of the 7th Joint Modular Languages Conference (JMLC 2006), 362-376, 2006.
-  Jingling Xue, Qiong Cai. *A Lifetime Optimal Algorithm for Speculative PRE*. ACM Transactions on Architecture and Code Optimization 3(2):115-155, 2006.
-  Hucheng Zhou, Wenguang Chen, Fred C. Chow. *An SSA-based Algorithm for Optimal Speculative Code Motion under an Execution Profile*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), ACM SIGPLAN Notices 46(6):98-108, 2011.

Further Reading for Chapter 10 (10)




Unifying Speculative and Non-Speculative Classical PRE

-  Jingling Xue, Jens Knoop. *A Fresh Look at PRE as a Maximum Flow Problem*. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006), Springer-V., LNCS 3923, 139 - 154, 2006.

Composite Syntactic Code Motion and Strength Reduction



-  Dhananjay M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. International Journal of Computer Mathematics 27:1-14,31-32, 1989.
-  Dhananjay M. Dhamdhere, J. R. Isaac. *A Composite Algorithm for Strength Reduction and Code Movement Optimization*. International Journal of Computer and Information Sciences 9(3):243-273, 1980.

Further Reading for Chapter 10 (11)

-  Robert Kennedy, Fred C. Chow, Peter Dahl, Shing-Ming Liu, Raymond Lo, Mark Streich. *Strength Reduction via SSAPRE*. In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 144-158, 1998.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. Journal of Programming Languages 1(1):71-91, 1993.
-  S. M. Joshi, Dhananjay M. Dhamdhere. *A Composite Hoisting- strength Reduction Transformation for Global Program Optimization – Part I and Part II*. International Journal of Computer Mathematics 11:21-41,111-126, 1982.

Further Reading for Chapter 10 (12)

Eliminating Partially Dead/Redundant Code

-  Ras Bodik, Rajiv Gupta. *Partial Dead Code Elimination using Slicing Transformations*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM SIGPLAN Notices 32:159-170, 1997.
-  Alfons Geser, Jens Knoop, Gerald Lüttgen, Oliver Rüthing, Bernhard Steffen. *Non-Monotone Fixpoint Iterations to Resolve Second Order Effects*. In Proceedings of the 6th International Conference on Compiler Construction (CC'96), Springer-V., LNCS 1060, 106-120, 1996.

Further Reading for Chapter 10 (13)

-  Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Dead Code Elimination using Predication*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 102-115, 1997.
-  Jens Knoop, Eduard Mehofer. *Optimal Distribution Assignment Placement*. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Springer-V., LNCS 1300, 364 - 373, 1997.
-  Jens Knoop, Eduard Mehofer. *Interprocedural Distribution Assignment Placement: More than just Enhancing Intraprocedural Placing Techniques*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 26-37, 1997.


Further Reading for Chapter 10 (14)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Partial Dead Code Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):147-158, 1994.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *The Power of Assignment Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), ACM SIGPLAN Notices 30(6):233-245, 1995.
-  Munehiro Takimoto, Kenichi Harada. *Partial Dead Code Elimination Using Extended Value Graph*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Springer-V., LNCS 1694, 179-193, 1999.


Further Reading for Chapter 10 (15)

Semantic PRE




Local Value Numbering (on (extended) basic blocks)

-  John Cocke, Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 2nd Revised Version, 771 pages, 1970. (Chapter 6, Optimization Methods for Algebraic Languages)




Global Value Numbering

-  Bowen Alpern, Mark N. Wegman, F. Kenneth Zadeck. *Detecting Equality of Variables in Programs*. In Conference Record of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 1-11, 1988.




Further Reading for Chapter 10 (16)

-  Preston Briggs, Keith D. Cooper, L. Taylor Simpson. *Value Numbering*. *Software: Practice and Experience* 27(6):701-724, 1997.
-  Cliff Click. *Global Code Motion, Global Value Numbering*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, ACM SIGPLAN Notices 29(6):246-257, 1995.
-  Cliff Click, Keith D. Cooper. *Combing Analyses, Combining Optimizations*. *ACM Transactions on Programming Languages and Systems* 17(2):181-196, 1995.

Further Reading for Chapter 10 (17)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Expansion-based Removal of Semantic Partial Redundancies*. In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.
-  Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. *Global Value Numbers and Redundant Computations*. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 12-27, 1988.
-  Oliver Rüthing. *Interacting Code Motion Transformations: Their Impact and Their Complexity*. Springer-V., LNCS 1539, 1998.


Further Reading for Chapter 10 (18)

-  Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Detecting Equalities of Variables: Combining Efficiency with Precision*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Springer-V., LNCS 1694, 232-247, 1999.
-  Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Proceedings of the 2nd Joint Conference on Theory and Practice of Software Development (TAPSOFT'87), Springer-V., LNCS 249, 52-68, 1987.
-  Bernhard Steffen, Jens Knoop, Oliver Rüthing. *The Value Flow Graph: A Program Representation for Optimal Program Transformations*. In Proceedings of the 3rd European Symposium on Programming (ESOP'90), Springer-V., LNCS 432, 389-405, 1990.

Further Reading for Chapter 10 (19)


-  Munehiro Takimoto, Kenichi Harada. *Effective Partial Redundancy Elimination based on Extended Value Graph*. Information Processing Society of Japan 38(11):2237-2250, 1990.

Composite Semantic Code Motion and Strength Reduction


-  Bernhard Steffen, Jens Knoop, Oliver Rüthing. *Efficient Code Motion and an Adaption to Strength Reduction*. In Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), Springer-V., LNCS 494, 394-415, 1991.

Further Reading for Chapter 10 (20)

Code Motion vs. Code Placement

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Code Motion and Code Placement: Just Synonyms?* In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 154-169, 1998.

BB Graphs vs. SI Graphs

-  Jens Knoop, Dirk Koschützki, Bernhard Steffen. *Basic-block Graphs: Living Dinosaurs?* In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 65-79, 1998.

Part III

Interprocedural Data Flow Analysis

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

G 1008/18

Outline and Orientation

We consider:

- ▶ **Foundations, the Interprocedural DFA Framework** (cf. Chap. 11)
 - ▶ Mutually recursive procedures (no parameters, no local variables)
- ▶ **The Functional Approach** (cf. Chap. 12)
 - ▶ Adding value parameters, local variables: **DFA stacks**
 - ▶ Adding reference parameters, procedural parameters
- ▶ **The Context Information Approach** (cf. Chap. 13)
 - ▶ Call Strings
 - ▶ Assumption Sets
 - ▶ The Cloning-Based Approach

Chapter 11

Foundations, the Interprocedural DFA Framework

Chapter 11.1

Preliminaries, the Setting

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1011/18

The Basic Setting of Interprocedural DFA

Program Setting

- ▶ Programs Π with mutually recursive procedures **without** parameters and local variables.

Program Representations

- ▶ Flow graph systems
- ▶ Interprocedural flow graphs

...two program representations which are **complementary** to each other.

Flow Graph Systems

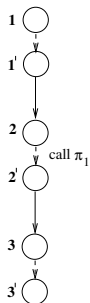
Intuitively, a **flow graph system** is a **system of flow graphs**, where every flow graph is a flow graph in the intraprocedural sense representing a **procedure** of a program Π .

Definition 11.1.1 (Flow Graph System)

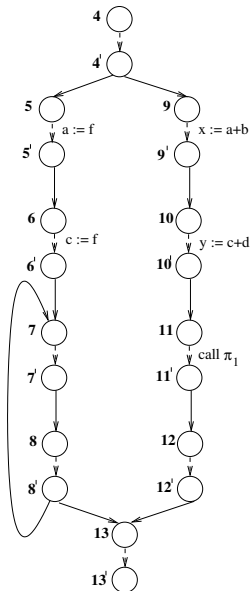
Let $\Pi = \langle \pi_0, \pi_1, \dots, \pi_k \rangle$ be a **program** with **main procedure** (or **main program**) π_0 and procedures π_1, \dots, π_k . A **flow graph system** $S_\Pi = \langle G_0, G_1, \dots, G_k \rangle$ for Π is a system of edge-labelled or node-labelled (intraprocedural) flow graphs in the sense of **Chapter 3**, where flow graph G_i represents procedure π_i , $0 \leq i \leq k$.

An Edge-Labelled Flow Graph System

π_0 ; a, b, x, y, c, d



π_1



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

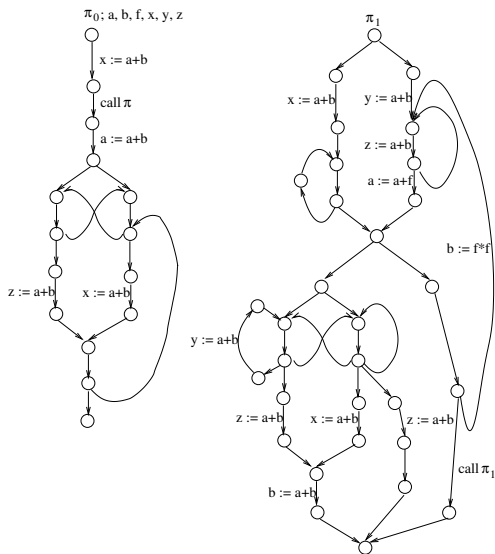
11.5

11.6

1014/18

Flow Graph System after Cleaning Up

...unnecessary/unused nodes and edges can be removed:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1015/18

Notations for Flow Graph Systems

Let $S_{\Pi} = \langle G_0, G_1, \dots, G_k \rangle$ be a flow graph system.

Then:

- ▶ G_0 represents the **main procedure** of Π . Instead of \mathbf{s}_0 and \mathbf{e}_0 , we often simply write \mathbf{s} and \mathbf{e} .
- ▶ The sets of nodes and edges N_i and E_i , $0 \leq i \leq k$, of all flow graphs of S_{Π} are assumed to be pairwise disjoint.
- ▶ $N =_{df} \bigcup_{i=0}^k N_i$ and $E =_{df} \bigcup_{i=0}^k E_i$ denote the set of all nodes and edges of a flow graph system, respectively.
- ▶ $E_{call} \subseteq E$ denotes the set of edges representing a **procedure call**, the set of **call edges**.
- ▶ If Π is obvious from the context and of no further relevance, we often write S instead of S_{Π} .

Interprocedural Flow Graphs

Intuitively, an **interprocedural flow graph** melts the flow graphs of a flow graph system to a **single graph**.

Definition 11.1.2 (Interprocedural Flow Graph)

An **interprocedural flow graph** $G^* = (N^*, E^*, s^*, e^*)$ is induced by a flow graph system S , where G^* evolves from S by replacing every **call edge** e of a flow graph G_i of S by two new edges, the **call edge** e_c and the **return edge** e_r .

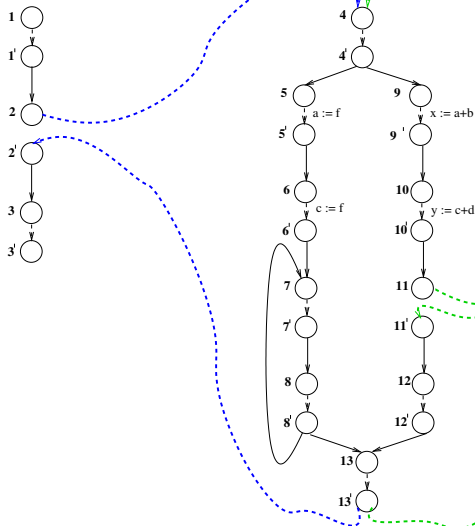
The **call edge** e_c connects the source node of e with the **start node** of the flow graph representing the called procedure.

The **return edge** e_r connects the end node of the flow graph representing the called procedure with the final node of e .

In particular, s^* and e^* are given by s_0 and e_0 , respectively.

An Edge-Labelled Interprocedural Flow Graph

$\pi_0 : a, b, x, y, c, d$



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

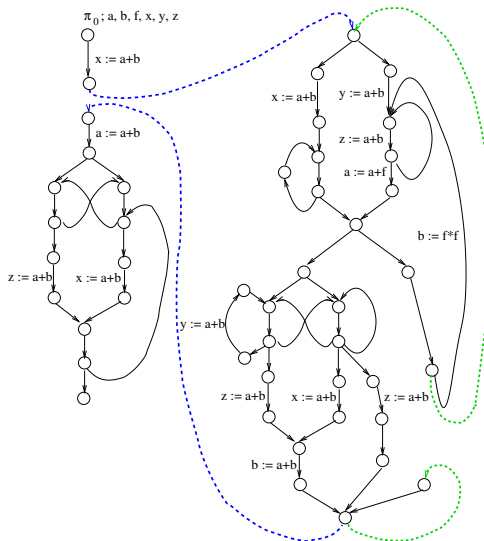
11.5

11.6

1018/18

Interprocedural Flow Graph after Cleaning Up

...unnecessary/unused nodes and edges can be removed:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1019/18

Notations for Interprocedural Flow Graphs

Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be an interprocedural flow graph.

Then:

- ▶ E_c^* and E_r^* denote the set of all call edges and return edges of G^* , respectively.
- ▶ $E_{call}^* =_{df} E_c^* \cup E_r^*$ denotes the union of the sets of call and return edges of G^* .
- ▶ Instead of \mathbf{s}^* and \mathbf{e}^* , we often simply write \mathbf{s} and \mathbf{e} .

Chapter 11.2

IDFA Specifications, IDFA Problems

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Interprocedural DFA Specification

Let S be an edge-labelled flow graph system, and $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be the interprocedural flow graph induced by S .

Definition 11.2.1 (IDFA Specification)

An (interprocedural) DFA specification for G^* is a quadruple $\mathcal{S}_{G^*} = (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, d)$ with

- ▶ $\hat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ a DFA lattice
- ▶ $\llbracket \cdot \rrbracket^* : E^* \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ a DFA functional
- ▶ $c_s \in \mathcal{C}$ an initial information/assertion
- ▶ $d \in \{fw, bw\}$ a direction of information flow

Note: As intraprocedurally, the validity of $c_s \in \mathcal{C}$ at $\mathbf{s} \equiv \mathbf{s}^*$ needs to be ensured by the calling context of G^* .

Notations for IDFA Specifications

Let $\mathcal{S}_{G^*} = (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, d)$ be a DFA specification for G^* .

Then

- ▶ The elements of \mathcal{C} represent the **data flow information** of interest.
- ▶ The functions $\llbracket e \rrbracket^*$, $e \in E^*$, abstract the concrete semantics of instructions to the level of the analysis.
- ▶ In the **parameterless setting** considered in this chapter, the local abstract semantics of **call edges** and **return edges** of E^* are given the **identity function** on \mathcal{C} .

As **intraprocedurally**

- ▶ $\hat{\mathcal{C}}$ is called a **DFA lattice**.
- ▶ $\llbracket \cdot \rrbracket^*$ is called an **(interprocedural) DFA functional**.
- ▶ $\llbracket e \rrbracket^*$, $e \in E^*$, is called a **(local) DFA function**.

Interprocedural DFA Problem

Definition 11.2.2 (IDFA Problem)

An interprocedural DFA specification $\mathcal{S}_{G^*} = (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, d)$ defines an **interprocedural DFA problem** for G^* .

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1024/18

Practically Relevant IDFA Problems

...similarly to the intraprocedural case, **interprocedural DFA problems** are practically relevant, if they are

- ▶ **monotonic**
- ▶ **distributive (additive)**

and satisfy the

- ▶ **descending (ascending) chain condition** (for the function lattice of the **DFA lattice!**).

Properties of IDFA Problems

Definition 11.2.3 (Properties of IDFA Problems)

Let $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, d)$ be an IDFA specification for G^* .

The IDFA problem induced by \mathcal{S}_{G^*}

- ▶ is **monotonic/distributive/additive** iff the DFA functional $\llbracket \cdot \rrbracket^*$ of \mathcal{S}_{G^*} is monotonic/distributive/additive.
- ▶ **satisfies the descending (ascending) chain condition** iff the function lattice $[\mathcal{C} \rightarrow \mathcal{C}]$ of the DFA lattice $\widehat{\mathcal{C}}$ of \mathcal{S}_{G^*} satisfies the descending (ascending) chain condition.

Note: If $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending (ascending) chain condition, then also $\widehat{\mathcal{C}}$ satisfies the descending (ascending) chain condition.

Chapter 11.3

Naive Interprocedural DFA

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Naive Interprocedural DFA

Note:

Considering an **interprocedural flow graph G^*** an **intraprocedural flow graph**, the (intraprocedural) notions of

- ▶ a **path**
- ▶ the ***MOP*** approach
- ▶ the ***MaxFP*** approach
- ▶ the Theorems for **Safety**, **Coincidence**, and **Termination**

carry over from the **intraprocedural setting** and an **intraprocedural flow graph G** to the **interprocedural setting** and an **interprocedural flow graph G^*** .

The *MOP* Approach and *MOP* Solution for G^*

Let $\mathcal{S}_{G^*} =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, fw)$ be a DFA specification for G^* .

Definition 11.3.1 (The *MOP* Solution for G^*)

The *MOP* solution of \mathcal{S}_{G^*} for G^* is defined by:

$$MOP_{\mathcal{S}_{G^*}} : N^* \rightarrow \mathcal{C}$$

$$\forall n \in N^*. MOP_{\mathcal{S}_{G^*}}(n) =_{df} \bigsqcap \{ \llbracket p \rrbracket^*(c_s) \mid p \in \mathbf{P}_{G^*}[\mathbf{s}, n] \}$$

The *MaxFP* Approach for G^*

Let $\mathcal{S}_{G^*} =_{df} (\hat{C}, \llbracket \cdot \rrbracket^*, c_s, fw)$ be a DFA specification for G^* .

Equation System 11.3.2 (The *MaxFP* EQS for G^*)

$$inf(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \prod \{ \llbracket (m, n) \rrbracket^*(inf(m)) \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Let $\nu\text{-}inf_{c_s}(n), n \in N^*$

denote the greatest solution of Equation System 11.3.2.

The *MaxFP* Solution for G^*

Definition 11.3.3 (The *MaxFP* Solution for G^*)

The *MaxFP* solution of \mathcal{S}_{G^*} for G^* is defined by:

$$\text{MaxFP}_{\mathcal{S}_{G^*}} : N^* \rightarrow \mathcal{C}$$

$$\forall n \in N^*. \text{MaxFP}_{\mathcal{S}_{G^*}}(n) =_{df} \nu\text{-inf}_{c_s}(n)$$

Safety and Coincidence

Corollary 11.3.4 (Safety)

The *MaxFP* solution of \mathcal{S}_{G^*} for G^* is a safe (i.e., lower) approximation of the *MOP* solution of \mathcal{S}_{G^*} for G^* , i.e.,

$$\forall n \in N^*. \text{MaxFP}_{\mathcal{S}_{G^*}}(n) \sqsubseteq \text{MOP}_{\mathcal{S}_{G^*}}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket^*$ is monotonic.

Corollary 11.3.5 (Coincidence)

The *MaxFP* solution of \mathcal{S}_{G^*} for G^* and the *MOP* solution of \mathcal{S}_{G^*} for G^* coincide, i.e.,

$$\forall n \in N^*. \text{MaxFP}_{\mathcal{S}_{G^*}}(n) = \text{MOP}_{\mathcal{S}_{G^*}}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket^*$ is distributive.

Termination

Corollary 11.3.6 (Termination)

Applied to G^* and \mathcal{S}_{G^*} , the Generic Fixed Point Algorithm 3.4.3 terminates with the *MaxFP* solution of \mathcal{S}_{G^*} for G^* , if

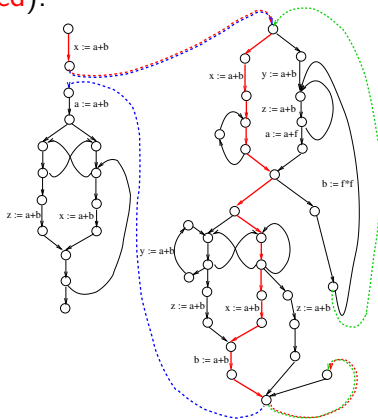
1. $\llbracket \cdot \rrbracket^*$ is monotonic
2. \hat{C} satisfies the descending chain condition.

...all three corollaries follow immediately from their intraprocedural counterparts of Chapter 3.

Everything done? Unfortunately, not!

...the *MOP* approach for G^* considers much too many paths as it does not respect the *call/return behaviour* of interprocedural program paths.

For illustration, consider the *interprocedurally infeasible path* (highlighted in red):



Observations on Naive Interprocedural DFA

- ▶ The notion of a (finite) path of intraprocedural flow graphs extends naturally to interprocedural flow graphs.
- ▶ In contrast to intraprocedural flow graphs, however, where every path connecting two nodes represents (up to non-determinism) a feasible execution of the program, this does not hold for interprocedural flow graphs.
- ▶ This causes the solutions of the naive extensions of the intraprocedural *MOP* and *MaxFP* approach to an interprocedural flow graph to be overly conservative.

In truly interprocedural DFA considered next this is taken care of and avoided by focusing on interprocedurally valid paths.

Chapter 11.4

The Interprocedural Meet over All Paths Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1036/18

Interprocedurally Valid Paths

Intuitively, **interprocedurally valid paths** respect the call/return behaviour of procedure calls

Definition 11.4.1 (Interprocedurally Valid Path)

Identifying call and return edges of G^* with opening and closing brackets '(' and ')', respectively, the set of **interprocedurally valid paths** is given by the set of prefix-closed expressions of the language of balanced bracket expressions.

Notation: In the following we denote the set of interprocedurally valid paths (for short: interprocedural paths) from a node m to a node n by **IP**[m, n].

Remarks on Interprocedurally Valid Paths

- ▶ Considering the sequences of edge labelings (we suppose that each edge is uniquely marked by some label) of a path a word of a formal language, the set of **intraprocedurally valid paths** is given by a **regular language**, the one of **interprocedurally valid paths** by a **context-free language**.
- ▶ The notion of **interprocedurally valid paths** can and has been defined in various ways:
 - ▶ The definition of interprocedurally valid paths as in **Definition 11.4.1** has been proposed by **Reps, Horwitz, and Sagiv (POPL'95)**.
 - ▶ An algorithmic definition of interprocedurally valid paths has been given by **Sharir and Pnueli in 1981**.
 - ▶ Based on the preceding remark, interprocedurally valid paths can also be defined in terms of a context-free language/grammar.

The *IMOP* Approach and the *IMOP* Solution

Let $\mathcal{S}_{G^*} =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, fw)$ be a DFA specification for G^* .

Definition 11.4.2 (The *IMOP* Solution)

The *IMOP* solution of \mathcal{S}_{G^*} is defined by:

$$IMOP_{\mathcal{S}_{G^*}} : N^* \rightarrow \mathcal{C}$$

$$\forall n \in N^*. IMOP_{\mathcal{S}_{G^*}}(n) =_{df} \bigsqcap \{ \llbracket p \rrbracket^*(c_s) \mid p \in \mathbf{IP}[s, n] \}$$

where $\mathbf{IP}[s, n]$ denotes the set of *interprocedurally valid paths* from s to n .

Conservative and Optimal IDFA Algorithms

Definition 11.4.3 (Conservative IDFA Algorithm)

An IDFA algorithm A is *IMOP conservative* for \mathcal{S}_{G^*} , if A terminates with a lower approximation of the *IMOP* solution of \mathcal{S}_{G^*} .

Definition 11.4.5 (Optimal IDFA Algorithm)

An IDFA algorithm A is *IMOP optimal* for \mathcal{S}_{G^*} , if A terminates with the *IMOP* solution of \mathcal{S}_{G^*} .

Chapter 11.5

The Interprocedural Maximal Fixed Point Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.5.1

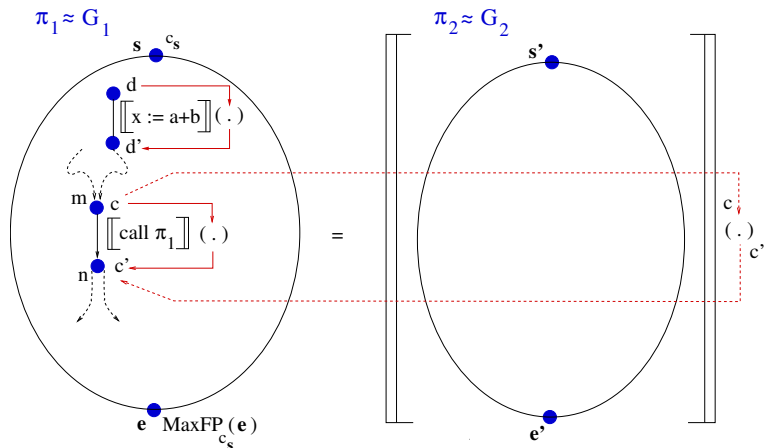
1041/18

Chapter 11.5.1

In Preparation: The Functional Approach of DFA

Towards Computing the *IMaxFP* Solution

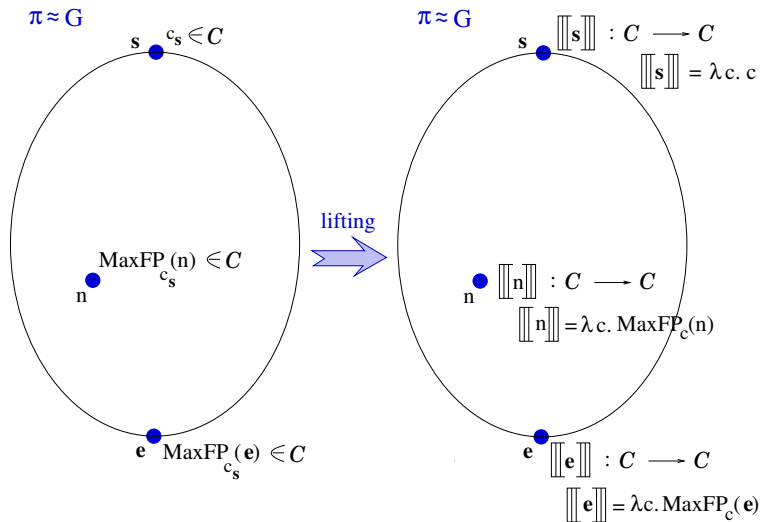
If we were to know the semantics of procedure calls:



...computing the *IMaxFP* solution would be straightforward.

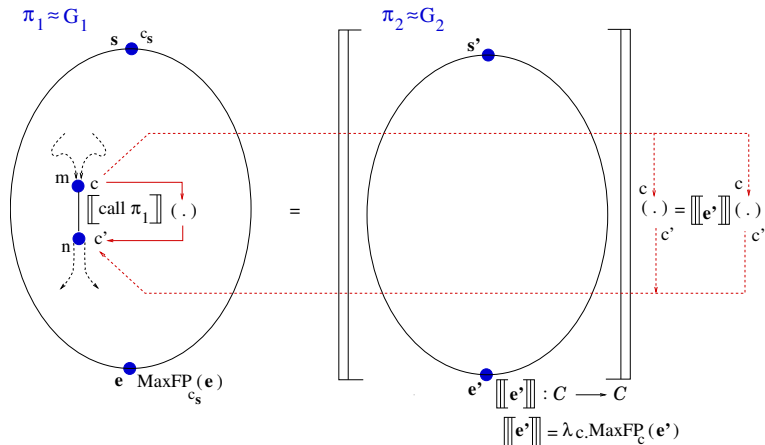
The Key to Computing the *IMaxFP*-Solution (1)

...lifting the analysis level from first order (lattice elements) to higher order (functions on lattices):



The Key to Computing the *IMaxFP*-Solution (2)

...this lifting step allows computing the semantics of procedure calls by a preprocess:



The Key to Computing the *IMaxFP*-Solution (3)

Summing up: If $S = (G_0, G_1, \dots, G_k)$ is a system of intra-procedural flow graphs and G_{j_0} a flow graph of S , then the higher order function

$$\llbracket \cdot \rrbracket : N_{j_0} \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

$$\forall n \in N_{j_0}. \llbracket n \rrbracket =_{df} \lambda c. \text{MaxFP}_{S_{G_{j_0}}^c}(n)$$

with $S_{G_{j_0}}^c =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket, c, fw)$ is the key to computing the *IMaxFP* solution because for all $0 \leq i \leq k$ the function:

$$\llbracket e_i \rrbracket : \mathcal{C} \rightarrow \mathcal{C}$$

$$\llbracket e_i \rrbracket =_{df} \lambda c. \text{MaxFP}_{S_{G_i}^c}(e)$$

is the global semantics of G_i : $\llbracket G_i \rrbracket = \llbracket e_i \rrbracket$.

As illustrated before, it is the functions $\llbracket G_1 \rrbracket, \dots, \llbracket G_k \rrbracket$ which are required for handling procedure calls in the course of computing the *IMaxFP* solution.

Reconsidering Intraprocedural DFA

...the *functional MaxFP Approach* lifts the

- ▶ *MaxFP* approach from elements of \mathcal{C} to functions on \mathcal{C} .

(Intuitively, this can be considered the pointwise extension of the *MaxFP* approach handling all DFA lattice elements simultaneously as start informations.)

Equation System 11.5.1.1 (Functional *MaxFP* EQS)

$$\llbracket n \rrbracket = \begin{cases} Id_{\mathcal{C}} & \text{if } n = \mathbf{s} \\ \bigcap \{ \llbracket (n, m) \rrbracket \circ \llbracket m \rrbracket \mid m \in \text{pred}(n) \} & \text{otherwise} \end{cases}$$

Let

$$\nu\text{-}\llbracket \rrbracket : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

denote the *greatest solution* of Equation System 11.5.1.1.

Main Results: Equivalence, Safety, Coincidence

...the *MaxFP* and the functional *MaxFP* approach are equivalent.

Theorem 11.5.1.2 (Equivalence)

$$\forall n \in N. \forall c_s \in \mathcal{C}. \nu\llbracket n \rrbracket(c_s) = \text{MaxFP}_{S_G^{cs}}(n)$$

Theorem 11.5.1.3 (Safety and Coincidence)

1. *Safety*: $\forall n \in N. \forall c_s \in \mathcal{C}. \nu\llbracket n \rrbracket(c_s) \sqsubseteq \text{MOP}_{S_G^{cs}}(n)$,
if $\llbracket \cdot \rrbracket$ is monotonic.
2. *Coincidence*: $\forall n \in N. \forall c_s \in \mathcal{C}. \nu\llbracket n \rrbracket(c_s) = \text{MOP}_{S_G^{cs}}(n)$,
if $\llbracket \cdot \rrbracket$ is distributive.

Note

The *functional variant* of the *MaxFP* approach is the key not only to *computing* the *IMaxFP* solution and thus to *computable*

- ▶ *interprocedural DFA* (i.e., of programs w/ procedures)

but also to *computable*

- ▶ *object-oriented* (i.e., of programs w/ classes, objects, and methods)
- ▶ *parallel* (i.e., of programs w/ parallelism)
- ▶ ...

data flow analysis.

Chapter 11.5.2

The *IMaxFP* Solution

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.5.1

The *IMaxFP* Approach

...is a **two-stage approach**:

- ▶ **Stage 1: Preprocess** – Computing the semantics of procedures
- ▶ **Stage 2: Main Process** – Computing the *IMaxFP* solution

Notations

The definition of the *two stage IMaxFP* approach requires the following mappings on a *flow graph system S*:

- ▶ *flowGraph* : $N \cup E \rightarrow S$ maps the nodes and edges of *S* to the flow graph containing them.
- ▶ *callee* : $E_{call} \rightarrow S$ maps every call edge to the flow graph of the called procedure.
- ▶ *caller* : $S \rightarrow \mathcal{P}(E_{call})$ maps every flow graph to the set of call edges calling it.
- ▶ *start* : $S \rightarrow \{s_0, \dots, s_k\}$ and *end* : $S \rightarrow \{e_0, \dots, e_k\}$ map every flow graph of *S* to its start node and stop node, respectively.

The *IMaxFP* Approach (1)

Stage 1: Preprocess – Computing the Semantics of Procedures

Equation System 11.5.2.1 (2nd Order *IMaxFP* EQS)

$$\llbracket n \rrbracket = \begin{cases} Id_C & \text{if } n \in \{s_0, \dots, s_k\} \\ \bigcap \{ \llbracket (m, n) \rrbracket \circ \llbracket m \rrbracket \mid m \in \text{pred}_{\text{flowGraph}(n)}(n) \} & \text{otherwise} \end{cases}$$

and

$$\llbracket e \rrbracket = \begin{cases} \llbracket e \rrbracket^* & \text{if } e \in E \setminus E_{\text{call}} \\ \llbracket \text{end}(\text{caller}(e)) \rrbracket & \text{otherwise} \end{cases}$$

Let

$$\nu\text{-}\llbracket n \rrbracket, n \in N, \quad \nu\text{-}\llbracket e \rrbracket, e \in E$$

denote the **greatest solutions** of Equation System 11.5.2.1.

The *IMaxFP* Approach (2)

Stage 2: Main Process – The ‘Actual’ Interprocedural DFA

Equation System 11.5.2.2 (1st Order *IMaxFP* EQS)

$$\mathit{inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} (\equiv \mathbf{s}_0) \\ \bigsqcap \{ \mathit{inf}(\mathit{src}(e)) \mid e \in \mathit{caller}(\mathit{flowGraph}(n)) \} & \text{if } n \in \{\mathbf{s}_1, \dots, \mathbf{s}_k\} \\ \bigsqcap \{ \nu\llbracket (m, n) \rrbracket(\mathit{inf}(m)) \mid m \in \mathit{pred}_{\mathit{flowGraph}(n)}(n) \} & \text{otherwise} \end{cases}$$

Let $\nu\text{-inf}_{c_s}(n), n \in N$

denote the greatest solution of Equation System 11.5.2.2.

The *IMaxFP* Solution

Let $\mathcal{S}_{G^*} =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, fw)$ be a DFA specification for G^* .

Definition 11.5.2.3 (The *IMaxFP* Solution)

The *IMaxFP* solution of \mathcal{S}_{G^*} is defined by:

$$IMaxFP_{\mathcal{S}_{G^*}} : N^* \rightarrow \mathcal{C}$$

$$\forall n \in N^*. IMaxFP_{\mathcal{S}_{G^*}}(n) =_{df} \nu\text{-inf}_{c_s}(n)$$

(Note that $N = N^*$ allows us to identify corresponding nodes of S and G^* .)

Chapter 11.6

The Generic Fixed Point Algorithms

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1056/18

Chapter 11.6.1

Basic Algorithms: Plain Vanilla

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Algorithm 11.6.1.1 – 2nd Order Preprocess

Input: A DFA specification $\mathcal{S}_{G^*} =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, d)$ for G^* resp. S .
If $d = bw$, the reversed versions of all graphs are used.

Output: On termination (cf. [Theorem 11.6.3.1](#)), the variables *gtr* (*global transformation*) and *ltr* (*local transformation*) store the values of the functions $\nu\text{-}\llbracket n \rrbracket : \mathcal{C} \rightarrow \mathcal{C}$, $n \in N$, and $\nu\text{-}\llbracket e \rrbracket : \mathcal{C} \rightarrow \mathcal{C}$, $e \in E$, being the greatest solutions of the [2nd order IMaxFP Equation System 11.5.2.1](#).

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow graph system S . Note that due to the mutual interdependence of the definitions of $\llbracket \cdot \rrbracket$ and $\llbracket \cdot \rrbracket$ the iterative approximation of $\llbracket \cdot \rrbracket$ is superposed by an interprocedural iteration step, which updates the current approximation of the effect function $\llbracket \cdot \rrbracket$ of call edges. The temporary *meet* stores the result of the most recent meet operation.

Algorithm 11.6.1.1 – 2nd Order Preprocess

(Prologue: Initializing the annotation arrays gtr and ltr and the variable $workset$)

FORALL $n \in N$ DO

 IF $n \in \{\mathbf{s}_0, \dots, \mathbf{s}_k\}$ THEN $gtr[n] := Id_C$

 ELSE $gtr[n] := \top_{[C \rightarrow C]}$ FI OD;

FORALL $e \in E$ DO

 IF $e \in E_{call}$ THEN $ltr[e] := \top_{[C \rightarrow C]}$ ELSE $ltr[e] := \llbracket e \rrbracket^*$ FI
OD;

$workset := \{\mathbf{s}_0, \dots, \mathbf{s}_k\};$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Algorithm 11.6.1.1 – 2nd Order Preprocess

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

 CHOOSE $m \in workset$;

$workset := workset \setminus \{m\}$;

 (Update the successor-environment of node m)

 IF $m \in \{e_1, \dots, e_k\}$

 THEN

 FORALL $e \in caller(flowGraph(m))$ DO

$ltr[e] := gtr[m]$;

$meet := ltr[e] \circ gtr[src(e)] \sqcap gtr[dst(e)]$;

 IF $gtr[dst(e)] \sqsupseteq meet$

 THEN

$gtr[dst(e)] := meet$;

$workset := workset \cup \{dst(e)\}$

 FI

 OD

Algorithm 11.6.1.1 – 2nd Order Preprocess

```
ELSE (i.e.,  $m \notin \{e_1, \dots, e_k\}$ )
  FORALL  $n \in \text{succ}_{\text{flowGraph}(m)}(m)$  DO
     $\text{meet} := \text{ltr}[(m, n)] \circ \text{gtr}[m] \sqcap \text{gtr}[n]$ ;
    IF  $\text{gtr}[n] \sqsupseteq \text{meet}$ 
      THEN
         $\text{gtr}[n] := \text{meet}$ ;
         $\text{workset} := \text{workset} \cup \{n\}$ 
      FI
    OD
  FI
ESOOHC
OD.
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1061/18

Algorithm 11.6.1.2 – 1st Order Main Process

Input: A DFA specification $\mathcal{S}_{G^*} =_{df} (\hat{\mathcal{C}}, [\]^*, c_s, d)$ for G^* resp. S . If $d = bw$, the reversed versions of all graphs are used, and the data flow functional $\nu\text{-}[\] : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ computed by Algorithm 11.6.1.1 for \mathcal{S}_{G^*} .

Output: On termination (cf. Theorem 11.6.3.1), variable $inf[n]$, $n \in N$, stores the *IMaxFP*-solution of \mathcal{S}_{G^*} at node n .

Additionally, we have (cf. Interprocedural Safety Theorem 11.7.3 and Interprocedural Coincidence Theorem 11.7.4): If

- ▶ $[\]^*$ is distributive: inf stores
- ▶ $[\]^*$ is monotonic: inf stores a lower approximation of

the *IMOP* solution of \mathcal{S}_{G^*} at node n .

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S . The temporary *meet* stores the result of the most recent meet operation.

Algorithm 11.6.1.2 – 1st Order Main Process

(Prologue: Initialization of the annotation array *inf* and the variable *workset*)

FORALL $n \in N \setminus \{s_0\}$ DO $inf[n] := \top$ OD;

$inf[s_0] := c_s$;

$workset := \{s_0\}$;

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Algorithm 11.6.1.2 – 1st Order Main Process

(Main process: Iterative fixed point computation)

```
WHILE  $workset \neq \emptyset$  DO
  CHOOSE  $m \in workset$ ;
   $workset := workset \setminus \{ m \}$ ;
  ( Update the successor-environment of node  $m$  )
  FORALL  $n \in succ_{flowGraph(m)}(m)$  DO
     $meet := \nu\text{-}\llbracket (m, n) \rrbracket (inf[m]) \sqcap inf[n]$ ;
    IF  $inf[n] \sqsupseteq meet$ 
      THEN
         $inf[n] := meet$ ;
         $workset := workset \cup \{ n \}$ 
  FI;
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Algorithm 11.6.1.2 – 1st Order Main Process

```
IF  $(m, n) \in E_{call}$ 
  THEN
     $meet := inf[m] \sqcap inf[start(callee((m, n)))]$ ;
    IF  $inf[start(callee((m, n)))] \sqsupseteq meet$ 
      THEN
         $inf[start(callee((m, n)))] := meet$ ;
         $workset := workset \cup \{ start(callee((m, n))) \}$ 
      FI
    FI
  OD
ESOOHC
OD.
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Chapter 11.6.2

Enhanced Algorithms: Improving Performance

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Variant 1: Exploiting $\llbracket \rrbracket^*$ More Effectively

...improving performance of the 1st order *IMaxFP*-Algorithm:

- ▶ Algorithm 11.6.1.1 and Algorithm 11.6.2.1 constitute a second pair of algorithms computing the *IMaxFP* solution.
- ▶ Algorithm 11.6.2.1 uses the semantics functions computed by Algorithm 11.6.1.1 more effectively than Algorithm 11.6.1.2.
- ▶ Unlike Algorithm 11.6.1.2, Algorithm 11.6.2.1 does not iterate over all nodes of S but only over procedure start nodes. After stabilization of the solution for the start nodes, a single run over all other nodes (cf. Epilogue) suffices to compute the *IMaxFP* solution at every node.

Algorithm 11.6.2.1 – 1st Order Main Process

Input: A DFA specification $\mathcal{S}_{G^*} =_{df} (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, c_s, d)$ for G^* resp. S . If $d = bw$, the reversed versions of all graphs are used, and the data flow functional $\nu\text{-}\llbracket \cdot \rrbracket : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ computed by Algorithm 11.6.1.1 for \mathcal{S}_{G^*} .

Output: On termination (cf. Theorem 13.6.3.1), variable $inf[n]$, $n \in N$, stores the *IMaxFP*-solution of \mathcal{S}_{G^*} at node n .

Additionally, we have (cf. Interprocedural Safety Theorem 11.7.3 and Interprocedural Coincidence Theorem 11.7.4): If

- ▶ $\llbracket \cdot \rrbracket^*$ is distributive: inf stores
- ▶ $\llbracket \cdot \rrbracket^*$ is monotonic: inf stores a lower approximation of

the *IMOP* solution of \mathcal{S}_{G^*} at node n .

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S . The temporary *meet* stores the result of the most recent meet operation.

Algorithm 11.6.2.1 – 1st Order Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)

```
FORALL  $\mathbf{s} \in \{\mathbf{s}_i \mid i \in \{1, \dots, k\}\}$  DO  $inf[\mathbf{s}] := \top$  OD;  
 $inf[\mathbf{s}_0] := c_{\mathbf{s}}$ ;  
 $workset := \{\mathbf{s}_i \mid i \in \{1, 2, \dots, k\}\}$ ;
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Algorithm 11.6.2.1 – 1st Order Main Process

(Main process: Iterative fixed point computation)

```
WHILE  $workset \neq \emptyset$  DO
  CHOOSE  $s \in workset$ ;
   $workset := workset \setminus \{s\}$ ;
   $meet := inf[s] \sqcap$ 
     $\sqcap \{ \nu \text{-} \llbracket src(e) \rrbracket_{Id_C} (inf[start(flowGraph(e))]) \mid e \in$ 
       $caller(flowGraph(s)) \}$ ;
  IF  $inf[s] \sqsupseteq meet$ 
    THEN
       $inf[s] := meet$ ;
       $workset := workset \cup \{ start(callee(e)) \mid e \in E_{call}.$ 
         $flowGraph(e) = flowGraph(s) \}$ 
    FI
  ESOOHC
OD;
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1070/18

Algorithm 11.6.2.1 – 1st Order Main Process

(Epilogue)

FORALL $n \in N \setminus \{s_i \mid i \in \{0, \dots, k\}\}$ DO

$inf[n] := \nu\text{-}\llbracket n \rrbracket (inf[start(flowGraph(n))])$ OD.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Variant 2: Interleaving 2nd & 1st Order Alg.

...improving performance of the algorithm composition by demand-drivenly invoking the 2nd order algorithm when need arises by the 1st order algorithm:

- ▶ Unlike the two algorithm pairs introduced so far, this algorithm variant interleaves the 1st order main process and the 2nd order preprocess.
- ▶ In effect, the semantics of procedures $\nu\text{-}\llbracket \text{ } \rrbracket : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ is computed demand-drivenly, and hence only partially instead of exhaustively totally, i.e., it is computed only for arguments encountered in the course of the 1st order main process instead of unguidedly for all arguments in a separate preprocess.

Variant 2: Interleaving 2nd & 1st Order Alg.

Algorithm 11.6.2.2 – Sketch of the Interleaved Algorithms

- ▶ The computation starts with the **1st order main process algorithm**.
- ▶ If a procedure call is encountered during the iterative process, the **2nd order preprocess algorithm** is started for this procedure and the current data flow fact.
- ▶ After completion of the computation of the effect of the procedure for this data flow fact, the **1st order main process algorithm** is continued with the computed result.

Note:

- ▶ The semantics of procedures is computed **demand-driven-ly** exclusively for required arguments.
- ▶ Overall, this leads to some **performance gain in practice**, which, however, is difficult to quantify.

Chapter 11.6.3

Termination

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Termination

Theorem 11.6.3.1 (Termination)

- ▶ The sequential compositions of Algorithm 11.6.1.1 (2nd order) and Algorithm 11.6.1.2 (1st order) resp. Algorithm 11.6.2.1 (1st order)
- ▶ Algorithm 11.6.2.2 interleaving Algorithm 11.6.1.2 resp. Algorithm 11.6.2.1, and Algorithm 11.6.1.1

terminate with the *IMaxFP* solution, if the data flow analysis functional $\llbracket \cdot \rrbracket^*$ is monotonic and the function lattice $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Note: Validity of the descending chain condition on the function lattice $[\mathcal{C} \rightarrow \mathcal{C}]$ implies validity of the descending chain condition on the underlying lattice $\hat{\mathcal{C}}$.

Chapter 11.7

Safety and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1076/18

Complete Interprocedural Paths

Definition 11.7.1 (Complete Interprocedural Path)

An **interprocedural path** p from the start node s_i of a procedure G_i , $i \in \{0, \dots, k\}$, to a node n within G_i is **complete**, if every procedure call, i.e., call edge, along p is completed by a corresponding procedure return, i.e., a return edge.

We denote the set of all **complete interprocedural paths** from s_i to n with **CIP** $[s_i, n]$.

Note:

- ▶ Intuitively, **completeness** of a path p , i.e., $p \in \mathbf{CIP}[s_i, n]$, ensures that the occurrences of s_i and n belong to the same incarnation of the procedure.
- ▶ The subpaths of a complete interprocedural path that belong to a procedure call, are either disjoint or properly nested.

Main Results: 2nd Order Analysis

Safety and coincidence results of the 2nd order analysis:

Theorem 11.7.2 (2nd Order Analysis)

For all $e \in E_{call}$ we have:

1. Safety:

$\nu\llbracket e \rrbracket \subseteq \bigcap \{ \llbracket p \rrbracket^* \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, if the data flow analysis functional $\llbracket \rrbracket^*$ is monotonic.

2. Coincidence:

$\nu\llbracket e \rrbracket = \bigcap \{ \llbracket p \rrbracket^* \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, if the data flow analysis functional $\llbracket \rrbracket^*$ is distributive.

where the mappings src and dst yield the start node and the final node of an edge, respectively.

Main Results: 1st Order Analysis

Safety and coincidence results of the 1st order analysis:

Theorem 11.7.3 (Interprocedural Safety)

The *IMaxFP* solution of \mathcal{S}_{G^*} is a safe (i.e., lower) approximation of the *IMOP* solution of \mathcal{S}_{G^*} , i.e.,

$$\forall n \in N. \text{IMaxFP}_{\mathcal{S}_{G^*}}(n) \sqsubseteq \text{IMOP}_{\mathcal{S}_{G^*}}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket^*$ is monotonic.

Theorem 11.7.4 (Interprocedural Coincidence)

The *IMaxFP* solution of \mathcal{S}_{G^*} coincides with the *IMOP* solution of \mathcal{S}_{G^*} , i.e.,

$$\forall n \in N. \text{IMaxFP}_{\mathcal{S}_{G^*}}(n) = \text{IMOP}_{\mathcal{S}_{G^*}}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket^*$ is distributive.

Conservativity, Optimality of IDFA Algorithms

Corollary 11.7.5 (*IMOP* Conservativity)

The IDFA algorithms of Chapter 11.6 are *IMOP* conservative for \mathcal{S}_{G^*} (i.e., terminate with a lower approximation of the *IMOP* solution of \mathcal{S}_{G^*}), if $\llbracket \cdot \rrbracket^*$ is monotonic and $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Corollary 11.7.6 (*IMOP* Optimality)

The IDFA algorithms of Chapter 11.6 are *IMOP* optimal for \mathcal{S}_{G^*} (i.e., terminate with the *IMOP* solution of \mathcal{S}_{G^*}), if $\llbracket \cdot \rrbracket^*$ is distributive and $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Chapter 11.8

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Soundness and Completeness (1)

Analysis Scenario:

- ▶ Let ϕ be a program property of interest (e.g., **availability of an expression**, **liveness of a variable**, etc.).
- ▶ Let $\mathcal{S}_{G^*}^\phi$ be a DFA specification designed for ϕ .

Definition 11.8.1 (Soundness)

$\mathcal{S}_{G^*}^\phi$ is **sound** for ϕ , if, whenever the *IMOP* solution of $\mathcal{S}_{G^*}^\phi$ indicates that ϕ is valid, then ϕ is valid.

Definition 11.8.2 (Completeness)

$\mathcal{S}_{G^*}^\phi$ is **complete** for ϕ , if, whenever ϕ is valid, then the *IMOP* solution of $\mathcal{S}_{G^*}^\phi$ indicates that ϕ is valid.

Soundness and Completeness (2)

Intuitively

- ▶ **Soundness** means: $IMOP_{S_{G^*}^\phi}$ implies ϕ .
- ▶ **Completeness** means: ϕ implies $IMOP_{S_{G^*}^\phi}$.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

1083/18

Soundness and Completeness (3)

If $\mathcal{S}_{G^*}^\phi$ is sound and complete for ϕ , this intuitively means:

We compute

- ▶ the property of interest,
- ▶ the whole property of interest,
- ▶ and only the property of interest.

In other words

- ▶ We compute the program property of interest accurately!

Chapter 11.9

A Uniform Framework and Toolkit View

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

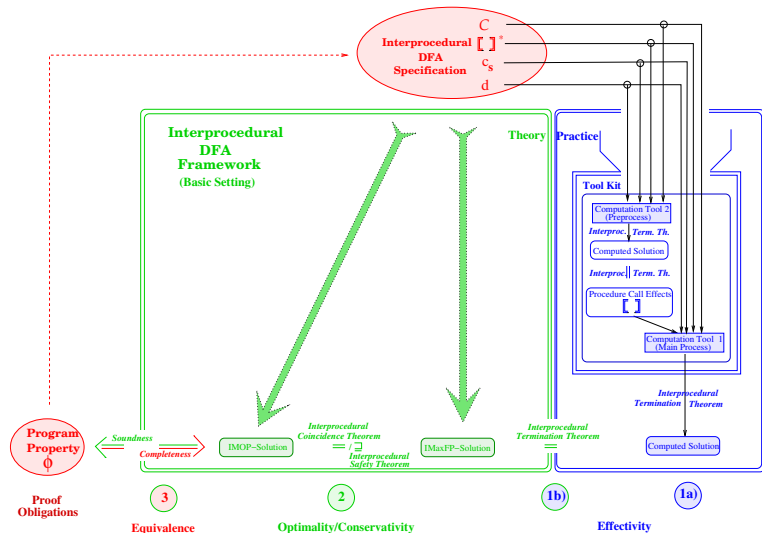
11.5

11.6

Interprocedural DFA: A Holistic Uniform View

...considering interprocedural DFA from a holistic perspective:

► A Uniform Framework and Toolkit View

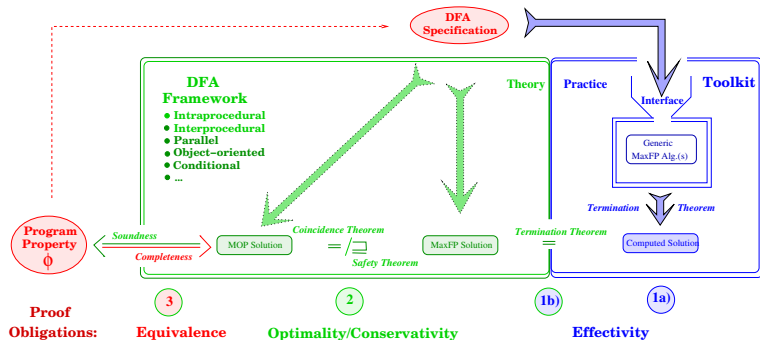


Note: The Preceding Schematic View of IDFA

...provides evidence for the claim of Chapter 3.8 that

► The Uniform Framework and Toolkit View of DFA

...is achievable beyond the base case of intraprocedural DFA.



Chapter 11.10

Applications

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2

11.3

11.4

11.5

11.6

Applications

For the parameterless base setting of interprocedural DFA

- ▶ the specifications of intraprocedural DFA problems can be reused unmodified.

In order to be effective

- ▶ the descending chain condition must hold both for the DFA lattice and its corresponding function lattice.

This requirement is satisfied in particular for all

- ▶ bitvector problems (availability of expressions, liveness of variables, reaching definitions, etc.) but not for simple constants. Therefore, weaker and simpler classes of constants are considered interprocedurally, e.g., linear constants.

Chapter 11.11

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

11.1

11.2



11.3

11.4



11.5

11.6

Further Reading for Chapter 11 (1)

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 12, Interprocedural Analysis)
-  Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 11, Interprocedural Analysis and Optimization)

Further Reading for Chapter 11 (2)

-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 19, Interprocedural Analysis and Optimization)
-  Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.3, The Functional Approach to Interprocedural Analysis, 196-209.

Chapter 12

The Functional Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

12.5

1093/18

Outline and Orientation

In this chapter, we **extend** the **parameterless basic setting** of **interprocedural DFA** (cf. Chapter 11) by successively **adding**:

- ▶ value parameters and local variables (cf. Chapter 12.1)
- ▶ procedural parameters (cf. Chapter 12.2)
- ▶ reference parameters (cf. Chapter 12.3)
- ▶ static procedure nesting (cf. Chapter 12.4)

Subsequently, we sketch **interprocedural applications**:

- ▶ **Availability** (as bitvector analysis) (cf. Chapter 12.5.1)
- ▶ **Copy constants** (as constant propagation analysis) (cf. Chapter 12.5.2)

Chapter 12.1

Value Parameters, Local Variables

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1095/18

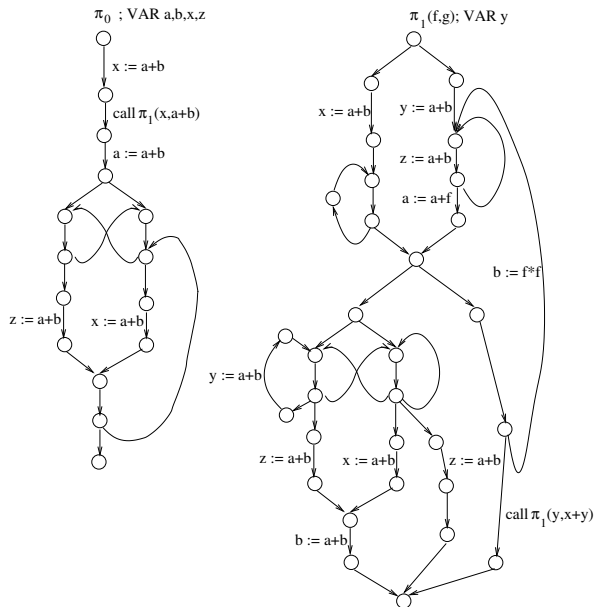
Flow Graph Systems, Interproc. Flow Graphs

Introducing **value parameters** and **local variables** requires to extend the notions of **flow graph systems (FGS)** and **interprocedural flow graphs (IFG)** to

- ▶ **flow graph systems with value parameters and local variables.**
- ▶ **interprocedural flow graphs with value parameters and local variables.**

This extension is straightforward as illustrated next.

FGS w/ Value Parameters and Local Variables



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

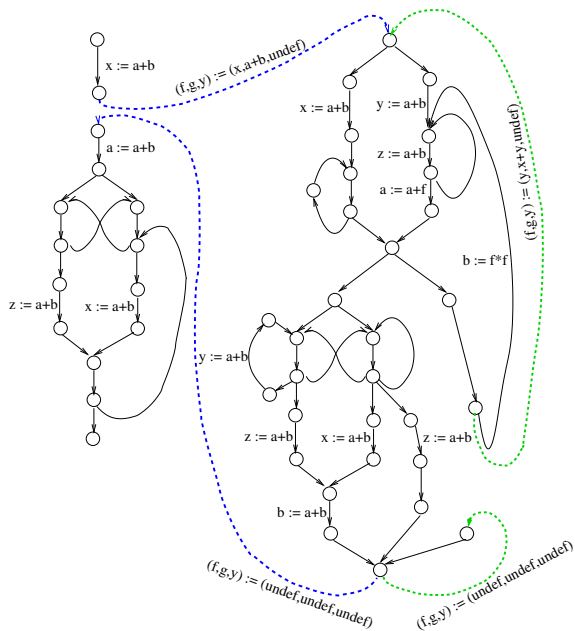
12.1.1

12.1.2

12.1.3

1097/18

IFG w/ Value Parameters and Local Variables



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1098/18

New Phenomena

...by [recursive procedures](#), [value parameters](#), [local variables](#):

- ▶ Existence of a potentially unlimited number of copies (incarnations) of local variables and value parameters due to (mutually) recursive procedure calls at run time.
- ▶ After termination of a recursive procedure call the local variables and value parameters of the preceding not yet finished procedure call become valid again.

The [run-time system](#)

- ▶ handles these phenomena by means of a [run-time stack](#) which stores the [activation records](#) of the various procedure incarnations.

In [data flow analysis](#)

- ▶ we have to take these phenomena into account and to model them properly introducing [DFA stacks](#).

Data Flow Analysis Stacks

Intuitively

- ▶ **DFA stacks** are a compile-time equivalent of run-time stacks.
- ▶ Entries in **DFA stacks** are **elements** (or **data flow facts**) of an underlying **DFA lattice** \hat{C} .
- ▶ **DFA stacks** contain at least one entry abstracting the activation record of the main program; **DFA stacks** are thus non-empty.

We denote

- ▶ the set of all (non-empty) **DFA stacks** by *STACK*.

Generating and Manipulating DFA Stacks

DFA stacks can be **generated** and **manipulated** by:

1. **newstack** : $\mathcal{C} \rightarrow STACK$

newstack(c) generates a new DFA stack with entry c .

2. **push** : $STACK \times \mathcal{C} \rightarrow STACK$

push stores a new entry on top of a DFA stack.

3. **pop** : $STACK \rightarrow STACK$

pop removes the top-most entry of a DFA stack.

4. **top** : $STACK \rightarrow \mathcal{C}$

top yields the contents of the top-most entry of a DFA stack w/out modifying the stack.

Remarks on DFA Stacks (1)

- ▶ DFA stack entries are abstractions of the activation records of procedure calls.
- ▶ The top-most entry of a DFA stack represents the currently valid activation record.
 - ▶ Therefore, DFA stacks are never empty and the commonly considered stack function $\text{emptystack} : \rightarrow \text{STACK}$ yielding an empty stack is replaced by $\text{newstack} : \mathcal{C} \rightarrow \text{STACK}$ yielding a stack with one entry.
- ▶ DFA stack entries other than the top-most entry abstract the activation records of started but not yet finished procedure calls.

Remarks on DFA Stacks (2)

- ▶ DFA stacks are only **conceptually** relevant, i.e., for the specifying *IMOP* approach but not for the **algorithmic** *IMaxFP* approach.
- ▶ In fact, the **algorithmic** *IMaxFP* approach requires only a temporary for storing a **single lattice element** being the **abstraction** of a **single activation record** instead of a full DFA stack.
- ▶ This ensures that
 - ▶ the *IMaxFP* solution gets **effectively computable** (in practically relevant scenarios).
 - ▶ **push** and **pop** allowing and limited to manipulating the top-most entries of a DFA stack are **sufficient** for **inter-procedural DFA** though a run-time stack is manipulated much more flexibly and efficiently by the run-time system for performance reasons.

Chapter 12.1.1

IDFA_{Stk} Specifications, IDFA_{Stk} Problems

DFA Functions and Return Functions

Let S be an edge-labelled flow graph system, and let $G^* = (N^*, E^*, s^*, e^*)$ be the interprocedural flow graph induced by S .

Moreover, let

- ▶ $\hat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ be a DFA lattice
- ▶ $\llbracket \cdot \rrbracket^* : E^* \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ be a DFA functional for G^*
- ▶ $\mathcal{R} : E_{call} \rightarrow (\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C})$ be a return functional for S

Then $\llbracket \cdot \rrbracket^*$ and \mathcal{R} induce a DFA_{Stk} functional on DFA stacks:

- ▶ $\llbracket \cdot \rrbracket_{Stk}^* : E^* \rightarrow (STACK \rightarrow STACK)$

for G^* defined next.

Induced DFA_{Stk} Functional on DFA Stacks

Definition 12.1.1.1 (Induced DFA_{Stk} Functional)

The DFA functional $\llbracket \cdot \rrbracket_{Stk}^* : E^* \rightarrow (STACK \rightarrow STACK)$ on DFA stacks induced by $\llbracket \cdot \rrbracket^*$ and \mathcal{R} is defined by:

$\forall e \in E^* \forall stk \in STACK. \llbracket e \rrbracket_{Stk}^*(stk) =_{df}$

$$\left\{ \begin{array}{ll} \text{push}(\text{pop}(stk), \llbracket e \rrbracket^*(\text{top}(stk))) & \text{if } e \in E^* \setminus E_{call}^* \\ \text{push}(stk, \llbracket e \rrbracket^*(\text{top}(stk))) & \text{if } e \in E_c^* \\ \text{push}(\text{pop}(\text{pop}(stk)), \mathcal{R}_{e_S}(\text{top}(\text{pop}(stk)), \llbracket e \rrbracket^*(\text{top}(stk)))) & \text{if } e \in E_r^* \end{array} \right.$$

where e_S denotes the call edge of S inducing the return edge $e \in E_r^*$ of G^* .

Interprocedural DFA_{Stk} Specification

Definition 12.1.1.2 (IDFA_{Stk} Specification)

An (interprocedural) DFA_{Stk} specification for G^* is a quintuple $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\]^*, \mathcal{R}, c_s, d)$ with

- ▶ $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ a DFA lattice
- ▶ $[\]^* : E^* \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ a DFA functional
- ▶ $\mathcal{R} : E_{call} \rightarrow (\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C})$ a return functional
- ▶ $c_s \in \mathcal{C}$ an initial information/assertion
- ▶ $d \in \{fw, bw\}$ a direction of information flow

Note: Definition 12.1.1.2 and Definition 11.2.1 differ only by the return functional \mathcal{R} . In particular, DFA stacks need not be dealt with on the level of IDFA_{Stk} specifications.

Interprocedural DFA_{Stk} Problem

Definition 12.1.1.3 (IDFA_{Stk} Problem)

An IDFA_{Stk} specification $\mathcal{S}_{G^*} = (\hat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, \mathcal{R}, c_s, d)$ defines an interprocedural DFA_{Stk} problem for G^* .

The Structure of DFA_{stk} Functions

Every DFA_{stk} function occurring in interprocedural DFA is an element of one of the subsets

► $\mathcal{F}_{ord}, \mathcal{F}_{psh}, \mathcal{F}_{pop}$

of the set of all functions $\mathcal{F} =_{df} [STACK \rightarrow STACK]$ on DFA stacks defined by:

$$\mathcal{F}_{ord} =_{df} \{ f \in \mathcal{F} \mid \forall stk \in STACK. \text{pop}(f(stk)) = \text{pop}(stk) \}$$

...at most the top entry of stacks is modified by functions of \mathcal{F}_{ord}

$$\mathcal{F}_{psh} =_{df} \{ f \in \mathcal{F} \mid \forall stk \in STACK. \text{pop}(f(stk)) = stk \}$$

...an element is pushed on top of the stack by functions of \mathcal{F}_{psh}

$$\mathcal{F}_{pop} =_{df} \{ f \in \mathcal{F} \mid \forall stk \in STACK_{\geq 2}. \text{pop}(f(stk)) = \text{pop}(\text{pop}(stk)) \}$$

...the top element of the stack is removed, at most the element below possibly be modified by functions of \mathcal{F}_{pop}

Characterizing DFA_{Stk} Functions

Lemma 12.1.1.4

$\forall f_{pp} \in \mathcal{F}_{pop} \quad \forall f_o, f'_o \in \mathcal{F}_{ord} \quad \forall f_{ph} \in \mathcal{F}_{psh}$.

1. $f_o \circ f'_o \in \mathcal{F}_{ord}$ (composition leaves stack size invariant)
2. $f_{pp} \circ f_o \circ f_{ph} \in \mathcal{F}_{ord}$ (composition leaves stack size invariant)

Lemma 12.1.1.5

1. $\forall e \in E^* \setminus E_{call}^* \cdot \llbracket e \rrbracket_{\text{Stk}}^* \in \mathcal{F}_{ord}$ (leaves stack size invariant)
2. $\forall e \in E_c^* \cdot \llbracket e \rrbracket_{\text{Stk}}^* \in \mathcal{F}_{psh}$ (increases stack size by one)
3. $\forall e \in E_r^* \cdot \llbracket e \rrbracket_{\text{Stk}}^* \in \mathcal{F}_{pop}$ (decreases stack size by one)

The Significant Function of DFA_{Stk} Functions

At most the **top** or the **two top-most entries** of **DFA stacks** are modified by DFA_{Stk} functions (cf. Lemma 12.1.1.5). This gives rise to the following definition:

Definition 12.1.1.6 (Significant Function)

- ▶ Let $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh}$: Then $f_{sig} : \mathcal{C} \rightarrow \mathcal{C}$ is defined by:

$$f_{sig}(c) =_{df} \text{top}(f(\text{newstack}(c)))$$

- ▶ Let $f \in \mathcal{F}_{pop}$: Then $f_{sig} : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is defined by:

$$f_{sig}(c_1, c_2) =_{df} \text{top}(f(\text{push}(\text{newstack}(c_1), c_2)))$$

(Recall that $\mathcal{C} \times \mathcal{C}$ is a lattice, if \mathcal{C} is a lattice.)

The functions f_{sig} are the **significant functions** of the DFA_{Stk} functions f .

Characterizing DFA_{Stk} Functions (Cont'd)

...via **significant** functions:

Lemma 12.1.1.7

1. $\forall e \in E^* \setminus E_r^*. \llbracket e \rrbracket_{\text{Stk sig}}^* = \llbracket e \rrbracket^*$
2. $\forall e \in E_r^* \forall c_1, c_2 \in \mathcal{C} \times \mathcal{C}. \llbracket e \rrbracket_{\text{Stk sig}}^* = \mathcal{R}_{e_S}(c_1, \llbracket e \rrbracket^*(c_2))$

where e_S denotes the call edge of S inducing the return edge $e \in E_r^*$ of G^* .

S-Monotonicity, S-Distributivity

Definition 12.1.1.8 (S-Monotonicity., S-Distrib.)

A DFA_{Stk} function $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh} \cup \mathcal{F}_{pop}$ is

1. **s-monotonic** iff f_{sig} is monotonic
2. **s-distributive** iff f_{sig} is distributive

Characterizing DFA_{Stk} Functions (Cont'd)

Lemma 12.1.1.9

Let $e \in E^*$. The function $\llbracket e \rrbracket_{Stk}^*$ is *s-monotonic* (*s-distributive*), if the following two conditions hold:

- a) $e \in E^* \setminus E_r^* : \llbracket e \rrbracket^*$ is monotonic (distributive)
- b) $e \in E_r^* : \llbracket e \rrbracket^*$ and \mathcal{R}_{e_S} are monotonic (distributive)

where e_S denotes the call edge of S inducing the return edge $e \in E_r^*$ of G^* .

Conventions

In the following, we

- ▶ identify lattice elements with their representation as a DFA stack with just a single entry.
- ▶ extend the meet and join operation \sqcap and \sqcup on DFA lattices as follows to (the top most entries of) sets of DFA stacks $STK \subseteq STACK$:

$$\sqcap STK =_{df} \text{newstack}(\sqcap \{top(stk) \mid stk \in STK\})$$

$$\sqcup STK =_{df} \text{newstack}(\sqcup \{top(stk) \mid stk \in STK\})$$

This allows us to consider

- ▶ s-monotonicity and s-distributivity generalizations of the usual notions of monotonicity and distributivity.

Chapter 12.1.2

The $IMOP_{Stk}$ Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

116/18

The $IMOP_{Stk}$ Approach & $IMOP_{Stk}$ Solution

Let $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, \llbracket \cdot \rrbracket^*, \mathcal{R}, c_s, d)$ be a DFA_{Stk} specification for G^* .

Definition 12.1.2.1 (The $IMOP_{Stk}$ Solution)

The $IMOP_{Stk}$ solution of \mathcal{S}_{G^*} is defined by:

$$IMOP_{Stk}^{\mathcal{S}_{G^*}} : N^* \rightarrow STACK_1$$

$$\forall n \in N^*. IMOP_{Stk}^{\mathcal{S}_{G^*}}(n) =_{df}$$

$$\sqcap \{ \llbracket p \rrbracket_{Stk}^* (\text{newstack}(c_s)) \mid p \in \mathbf{IP}[s, n] \}$$

where $STACK_1$ denotes the set of **DFA stacks** with exactly one entry.

Conservative and Optimal IDFA_{Stk} Algorithms

Definition 12.1.2.2 (Conservative IDFA Algorithm)

An IDFA algorithm A is *IMOP_{Stk} conservative* for \mathcal{S}_{G^*} , if A terminates with a lower approximation of the *IMOP_{Stk}* solution of \mathcal{S}_{G^*} .

Definition 12.1.2.3 (Optimal IDFA Algorithm)

An IDFA algorithm A is *IMOP_{Stk} optimal* for \mathcal{S}_{G^*} , if A terminates with the *IMOP_{Stk}* solution of \mathcal{S}_{G^*} .

Chapter 12.1.3

The $IMaxFP_{Stk}$ Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1119/18

The $IMaxFP_{Stk}$ Approach

...is a two-stage approach:

- ▶ Stage 1: Preprocess – Computing the semantics of procedures
- ▶ Stage 2: Main Process – Computing the $IMaxFP_{Stk}$ solution

Preliminaries

Let

- ▶ Id_{STACK} denote the identity on $STACK$
- ▶ \sqcap be the pointwise meet-operation on \mathcal{F}_{ord}

Note

- ▶ $\forall f, f' \in \mathcal{F}_{ord}. f \sqcap f' =_{df} f'' \in \mathcal{F}_{ord}$ with
 $\forall stk \in STACK. \text{top}(f''(stk)) =$
 $\text{top}(f(stk)) \sqcap \text{top}(f'(stk))$
- ▶ ' \sqcap ' induces an inclusion relation ' \sqsubseteq ' on \mathcal{F}_{ord} by:

$$f \sqsubseteq f' \text{ iff } f \sqcap f' = f$$

The $IMaxFP_{Stk}$ Approach: 2nd Order

Stage 1: Preprocess – Computing the Semantics of Procedures

Equation System 12.1.3.1 (2nd Order $IMaxFP_{Stk}$)

$$\llbracket n \rrbracket_{Stk} = \begin{cases} Id_{STACK} & \text{if } n \in \{s_0, \dots, s_k\} \\ \bigcap \{ \llbracket (m, n) \rrbracket_{Stk} \circ \llbracket m \rrbracket_{Stk} \mid m \in pred_{flowGraph(n)}(n) \} & \\ \text{otherwise} & \end{cases}$$

and

$$\llbracket e \rrbracket_{Stk} = \begin{cases} \llbracket e \rrbracket_{Stk}^* & \text{if } e \in E \setminus E_{call} \\ \llbracket e_r \rrbracket_{Stk}^* \circ \llbracket end(callee(e)) \rrbracket_{Stk} \circ \llbracket e_c \rrbracket_{Stk}^* & \text{otherwise} \end{cases}$$

Let

$$\nu\text{-}\llbracket n \rrbracket_{Stk}, n \in N, \quad \nu\text{-}\llbracket e \rrbracket_{Stk}, e \in E$$

denote the greatest solutions of Equation System 12.1.3.1.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1122/18

The $IMaxFP_{Stk}$ Approach: 1st Order

Stage 2: Main Process – The ‘Actual’ Interprocedural DFA

Equation System 12.1.3.2 (1st Order $IMaxFP_{Stk}$)

$$inf(n) = \begin{cases} newstack(c_s) & \text{if } n = \mathbf{s}_0 \\ \bigcap \{ \llbracket e_c \rrbracket_{Stk}^*(inf(src(e))) \mid e \in caller(flowGraph(n)) \} & \text{if } n \in start(S) \setminus \{ \mathbf{s}_0 \} \\ \bigcap \{ \nu\text{-}\llbracket (m, n) \rrbracket_{Stk}(inf(m)) \mid m \in pred_{flowGraph(n)}(n) \} & \text{otherwise} \end{cases}$$

Let $\nu\text{-}inf_{c_s}(n), n \in N$
denote the greatest solution of Equation System 12.1.3.2.

The $IMaxFP_{Stk}$ Solution

Let $\mathcal{S}_{G^*} = (\hat{C}, [\]^*, \mathcal{R}, c_s, d)$ be a DFA_{Stk} specification for G^* .

Definition 12.1.3.3 (The $IMaxFP_{Stk}$ Solution)

The $IMaxFP_{Stk}$ solution of \mathcal{S}_{G^*} is defined by:

$$IMaxFP_{Stk}^{\mathcal{S}_{G^*}} : N^* \rightarrow STACK_1$$

$$\forall n \in N^*. IMaxFP_{Stk}^{\mathcal{S}_{G^*}}(n) =_{df} \nu\text{-inf}_{c_s}(n)$$

where $STACK_1$ denotes the set of **DFA stacks** with exactly one entry.

Note that $N = N^*$ allows us to identify corresponding nodes of S and G^* .

Chapter 12.1.4

Safety and Coincidence

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1126/18

Towards the Main Results

...on *safety* and *coincidence*.

Lemma 12.1.4.1

For all $n \in N$, we have, if the semantic functions $\llbracket e \rrbracket^*$, $e \in E^*$, are

1. s-monotonic: $\nu\text{-}\llbracket n \rrbracket_{Stk} \sqsubseteq imop_n$
2. s-distributive: $\nu\text{-}\llbracket n \rrbracket_{Stk} = imop_n$

where $imop_n : N \rightarrow (STACK \rightarrow STACK)$ denotes a functional which is defined by:

$$\forall n \in N. imop_n =_{df} \begin{cases} Id_{STACK} & \text{if } n \in start(S) \\ \bigcap \{ \llbracket p \rrbracket_{Stk}^* \mid p \in \mathbf{CIP}[start(flowGraph(n)), n] \} & \text{otherwise} \end{cases}$$

Main Results: 2nd Order Analysis

Safety and coincidence results of the 2nd order analysis:

Theorem 12.1.4.2 (2nd Order Analysis)

For all $e \in E_{call}$ we have:

1. $\nu\llbracket e \rrbracket \subseteq \bigcap \{ \llbracket p \rrbracket_{stk}^* \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, if the data flow analysis functional $\llbracket \cdot \rrbracket_{stk}^*$ is s -monotonic.
2. $\nu\llbracket e \rrbracket = \bigcap \{ \llbracket p \rrbracket_{stk}^* \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$ if the data flow analysis functional $\llbracket \cdot \rrbracket_{stk}^*$ is s -distributive.

where the mappings src and dst yield the start and the final node of an edge, respectively.

Main Results: 1st Order Analysis

Safety and coincidence results of the 1st order analysis:

Theorem 12.1.4.3 (Interprocedural Safety)

The $IMaxFP_{Stk}$ solution of \mathcal{S}_{G^*} is a safe (i.e., lower) approximation of the $IMOP_{Stk}$ solution of \mathcal{S}_{G^*} , i.e.,

$$\forall n \in N. IMaxFP_{Stk}^{S_{G^*}}(n) \sqsubseteq IMOP_{Stk}^{S_{G^*}}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket_{Stk}^*$ is **s-monotonic**.

Theorem 12.1.4.4 (Interprocedural Coincidence)

The $IMaxFP_{Stk}$ solution of \mathcal{S}_{G^*} coincides with the $IMOP_{Stk}$ solution of \mathcal{S}_{G^*} , i.e.,

$$\forall n \in N. IMaxFP_{Stk}^{S_{G^*}}(n) = IMOP_{Stk}^{S_{G^*}}(n)$$

if the DFA functional $\llbracket \cdot \rrbracket_{Stk}^*$ is **s-distributive**.

Chapter 12.1.5

The Generic Fixed Point Algorithms

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1129/18

Algorithms

The [generic fixed point algorithms](#) of [Chapter 11.6](#)

- ▶ can straightforwardly be extended to stack-based ones.
- ▶ This way, we receive
 - ▶ the standard variant of preprocess and main process
 - ▶ the more efficient variant of preprocess and functional main process.
 - ▶ a demand-driven 'by-need' variant interleaving the 1st and 2nd order analyses.

In the following, we [present](#)

- ▶ another variant, which is [stackless](#).
- ▶ The clou of this variant is that stacks have [at most 2 entries during analysis time](#).
- ▶ Therefore, a [single temporary](#) storing the temporarily existing stack entry during procedure calls suffices for the implementation.

Alg. 12.1.5.1 – Stackless 2nd Order Preprocess

Input: (1) A flow-graph system S , and (2) an abstract semantics consisting of a data-flow lattice \mathcal{C} , and a data-flow functional $\llbracket \]^* : E^* \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$.

Output: Under the assumption of termination (cf. [Termination Theorem 12.1.5.4](#)), an annotation of S with functions $\llbracket n \rrbracket : \mathcal{C} \rightarrow \mathcal{C}$ (stored in *gtr*, which stands for *global transformation*), and $\llbracket e \rrbracket : \mathcal{C} \rightarrow \mathcal{C}$ (stored in *ltr*, which stands for *local transformation*) representing the greatest solution of [Equation System 12.1.3.1](#).

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S . Note that due to the mutual interdependence of the definitions of $\llbracket n \rrbracket$ and $\llbracket e \rrbracket$ the iterative approximation of $\llbracket n \rrbracket$ is superposed by an interprocedural iteration step, which updates the current approximation of the effect $\llbracket e \rrbracket$ of call edges. The temporary *meet* stores the result of the most recent meet operation.

Alg. 12.1.5.1 – Stackless 2nd Order Preprocess

(Prologue: Initialization of the annotation arrays gtr and ltr and the variable $workset$)

FORALL $n \in N$ DO

IF $n \in \{s_0, \dots, s_k\}$ THEN $gtr[n] := Id_C$

ELSE $gtr[n] := \top_{[C \rightarrow C]}$ FI OD;

FORALL $e \in E$ DO

IF $e \in E_{call}$ THEN $ltr[e] := \llbracket e_r \rrbracket^* \circ \top_{[C \rightarrow C]} \circ \llbracket e_c \rrbracket^*$

ELSE $ltr[e] := \llbracket e \rrbracket^*$ FI OD;

$\langle \star \rangle$

$workset := \{s_0, \dots, s_k\};$

Alg. 12.1.5.1 – Stackless 2nd Order Preprocess

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

 CHOOSE $m \in workset$;

$workset := workset \setminus \{ m \}$;

 (Update the successor-environment of node m)

 IF $m \in \{ \mathbf{e}_1, \dots, \mathbf{e}_k \}$

 THEN

 FORALL $e \in caller(flowGraph(m))$ DO

$ltr[e] := \mathcal{R}_e \circ (Id_C, \llbracket e_r \rrbracket^* \circ gtr[m] \circ \llbracket e_c \rrbracket^*)$; $\langle \star \rangle$

$meet := ltr[e] \circ gtr[src(e)] \sqcap gtr[dst(e)]$;

 IF $gtr[dst(e)] \sqsupset meet$

 THEN

$gtr[dst(e)] := meet$;

$workset := workset \cup \{ dst(e) \}$

 FI

 OD

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1133/18

Alg. 12.1.5.1 – Stackless 2nd Order Preprocess

```
ELSE (i.e.,  $m \notin \{e_1, \dots, e_k\}$ )
  FORALL  $n \in \text{succ}_{\text{flowGraph}(m)}(m)$  DO
     $\text{meet} := \text{ltr}[(m, n)] \circ \text{gtr}[m] \sqcap \text{gtr}[n]$ ;
    IF  $\text{gtr}[n] \sqsupseteq \text{meet}$ 
      THEN
         $\text{gtr}[n] := \text{meet}$ ;
         $\text{workset} := \text{workset} \cup \{n\}$ 
      FI
    OD
  FI
ESOOHC
OD.
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1134/18

Alg. 12.1.5.2 – Stackl. 1st Ord. Main Process

Input: (1) A flow-graph system S , (2) an abstract semantics consisting of a data-flow lattice \mathcal{C} , and a data-flow functional $\llbracket \cdot \rrbracket$ computed by [Algorithm 12.1.5.1](#), and (3) a context information $c_s \in \mathcal{C}$.

Output: Under the assumption of termination (cf. [Termination Theorem 12.1.5.4](#)), the $IMaxFP_{StkLss}$ -solution. Depending on the properties of the data-flow functional, this has the following interpretation:

- (1) $\llbracket \cdot \rrbracket$ is **distributive**: variable *inf* stores for every node the strongest component information valid there with respect to the context information c_s .
- (2) $\llbracket \cdot \rrbracket$ is **monotonic**: variable *inf* stores for every node a valid component information with respect to the context information c_s , i.e., a lower bound of the strongest component information valid there.

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S . The temporary *meet* stores the result of the most recent meet operation.

Alg. 12.1.5.2 – Stackl. 1st Ord. Main Process

(Prologue: Initialization of the annotation array *inf* and the variable *workset*)

FORALL $n \in N \setminus \{s_0\}$ DO $inf[n] := \top$ OD;

$inf[s_0] := c_s$;

$workset := \{s_0\}$;

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

 CHOOSE $m \in workset$;

$workset := workset \setminus \{m\}$;

 (Update the successor-environment of node m)

 FORALL $n \in succ_{flowGraph(m)}(m)$ DO

$meet := \llbracket (m, n) \rrbracket (inf[m]) \sqcap inf[n]$;

 IF $inf[n] \sqsupseteq meet$

 THEN

$inf[n] := meet$;

$workset := workset \cup \{n\}$

 FI;

Alg. 12.1.5.2 – Stackl. 1st Ord. Main Process

```
IF  $(m, n) \in E_{call}$ 
  THEN
     $meet := \llbracket (m, n)_c \rrbracket^*(inf[m]) \sqcap$ 
       $inf[start(callee((m, n)))]$ ;            $\langle \star \rangle$ 
IF  $(m, n) \in E_{call}$ 
  THEN
     $meet := \llbracket (m, n)_c \rrbracket^*(inf[m]) \sqcap$ 
       $inf[start(callee((m, n)))]$ ;            $\langle \star \rangle$ 
    IF  $inf[start(callee((m, n)))] \sqsupseteq meet$ 
      THEN
         $inf[start(callee((m, n)))] := meet$ ;
         $workset := workset \cup \{ start(callee((m, n))) \}$ 
      FI
    FI
  FI
OD
ESOOHC
OD.
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1137/18

Alg. 12.1.5.3 – Stackless ‘Functional’ 1st Ord. Main Process

Input: (1) A flow-graph system S , (2) an abstract semantics consisting of a data-flow lattice \mathcal{C} , and the data-flow functionals $\llbracket \cdot \rrbracket =_{df} gtr$ and $\llbracket \cdot \rrbracket =_{df} ltr$ with respect to \mathcal{C} (computed by [Algorithm 12.1.5.1](#)), and (4) a context information $c_s \in \mathcal{C}$.

Output: Under the assumption of termination (cf. [Termination Theorem 12.1.5.4](#)), the $IMaxFP_{StkLss}$ -solution. Depending on the properties of the data-flow functional, this has the following interpretation:

- (1) $\llbracket \cdot \rrbracket$ is **distributive**: variable *inf* stores for every node the strongest component information valid there with respect to the context information c_s .
- (2) $\llbracket \cdot \rrbracket$ is **monotonic**: variable *inf* stores for every node a valid component information with respect to the context information c_s , i.e., a lower bound of the strongest component information valid there.

Remark: The variable *workset* controls the iterative process, and the temporary *meet* stores the most recent approximation.

Alg. 12.1.5.3 – Stackless ‘Functional’ 1st Ord. Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)

```
FORALL  $\mathbf{s} \in \{\mathbf{s}_i \mid i \in \{1, \dots, k\}\}$  DO  $inf[\mathbf{s}] := \top$  OD;  
 $inf[\mathbf{s}_0] := c_{\mathbf{s}}$ ;  
 $workset := \{\mathbf{s}_i \mid i \in \{1, 2, \dots, k\}\}$ ;
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1139/18

Alg. 12.1.5.3 – Stackless ‘Functional’ 1st Ord. Main Process

(Main process: Iterative fixed point computation)

```
WHILE  $workset \neq \emptyset$  DO
  CHOOSE  $s \in workset$ ;
   $workset := workset \setminus \{s\}$ ;
   $meet := inf[s] \sqcap$ 
     $\sqcap \{ \llbracket e_c \rrbracket^* \circ \llbracket src(e) \rrbracket (inf[start(flowGraph(e))]) \mid$ 
       $e \in caller(flowGraph(s)) \}$ ;    $\langle \star \rangle$ 
  IF  $inf[s] \sqsupseteq meet$ 
    THEN
       $inf[s] := meet$ ;
       $workset := workset \cup$ 
         $\{ start(callee(e)) \mid e \in E_{call}.$ 
           $flowGraph(e) = flowGraph(s) \}$ 
    FI
  ESOOHC
OD;
```

Alg. 12.1.5.3 – Stackless ‘Functional’ 1st Ord. Main Process

(Epilogue)

```
FORALL  $n \in N \setminus \{s_i \mid i \in \{0, \dots, k\}\}$  DO  
   $inf[n] := \llbracket n \rrbracket (inf[start(flowGraph(n))])$   
OD.
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

Termination

Theorem 12.1.5.4 (Termination)

The sequential composition of [Algorithm 12.1.5.1](#) (2nd order) and [Algorithm 12.1.5.2](#) (1st order) resp. [Algorithm 12.1.5.3](#) (1st order) terminates with the $IMaxFP_{Stk}$ solution, if the DFA functional $\llbracket \cdot \rrbracket^*$ and the return functional \mathcal{R} are monotonic and the function lattice $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Note: Validity of the descending chain condition on the function lattice $[\mathcal{C} \rightarrow \mathcal{C}]$ implies validity of the descending chain condition on the underlying lattice $\hat{\mathcal{C}}$.

Conservativity, Optimality of IDFA Algorithms

Corollary 12.1.5.5 ($IMOP_{Stk}$ Conservativity)

The IDFA algorithms of Chapter 12.1.5 are $IMOP_{Stk}$ conservative for \mathcal{S}_{G^*} (i.e., terminate with a lower approximation of the $IMOP_{Stk}$ solution of \mathcal{S}_{G^*}), if $\llbracket \cdot \rrbracket^*$ and \mathcal{R} are monotonic and $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Corollary 12.1.5.6 ($IMOP_{Stk}$ Optimality)

The IDFA algorithms of Chapter 12.1.5 are $IMOP_{Stk}$ optimal for \mathcal{S}_{G^*} (i.e., terminate with the $IMOP_{Stk}$ solution of \mathcal{S}_{G^*}), if $\llbracket \cdot \rrbracket^*$ and \mathcal{R} are distributive and $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Chapter 12.1.6

Soundness and Completeness

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1144/18

Soundness and Completeness (1)

Analysis Scenario:

- ▶ Let ϕ be a program property of interest (e.g., **availability of an expression**, **liveness of a variable**, etc.).
- ▶ Let $\mathcal{S}_{G^*}^\phi$ be a DFA specification designed for ϕ .

Definition 12.1.6.1 (Soundness)

$\mathcal{S}_{G^*}^\phi$ is **sound** for ϕ , if, whenever the $IMOP_{Stk}$ solution of $\mathcal{S}_{G^*}^\phi$ indicates that ϕ is valid, then ϕ is valid.

Definition 12.1.6.2 (Completeness)

$\mathcal{S}_{G^*}^\phi$ is **complete** for ϕ , if, whenever ϕ is valid, then the $IMOP_{Stk}$ solution of $\mathcal{S}_{G^*}^\phi$ indicates that ϕ is valid.

Soundness and Completeness (2)

Intuitively

- ▶ **Soundness** means: $IMOP_{Stk}^{S_G^\phi}$ implies ϕ .
- ▶ **Completeness** means: ϕ implies $IMOP_{Stk}^{S_G^\phi}$.

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.1.1

12.1.2

12.1.3

1146/18

Soundness and Completeness (3)

If $\mathcal{S}_{G^*}^\phi$ is sound and complete for ϕ , this intuitively means:

We compute

- ▶ the property of interest,
- ▶ the whole property of interest,
- ▶ and only the property of interest.

In other words

- ▶ We compute the program property of interest accurately!

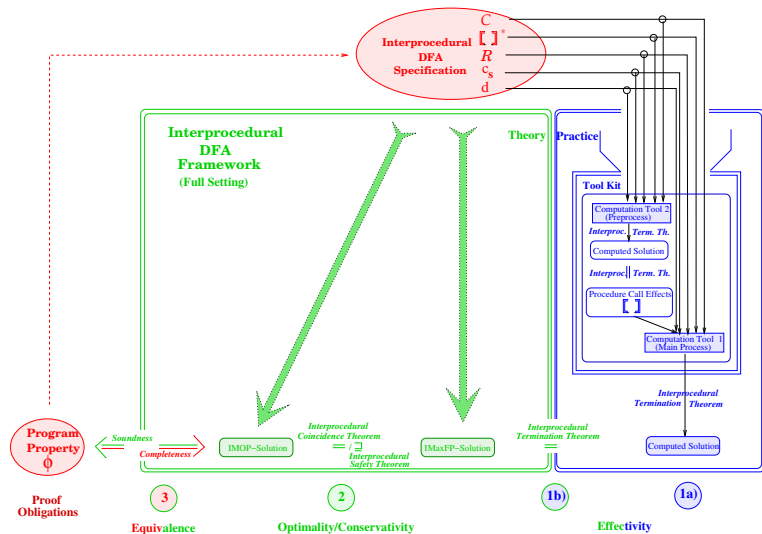
Chapter 12.1.7

A Uniform Framework and Toolkit View

IDFA with Value Parameters, Local Variables

...considered from a **holistic perspective** yields a

► Uniform Framework and Toolkit View



Chapter 12.2

Adding Procedural Parameters

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

1150/18

Procedural Parameters

Let $\Pi = \langle \pi_0, \dots, \pi_k \rangle$ be a program.

So far, we considered

- ▶ Procedure declarations: *proc* $\pi(p_1, \dots, p_r)$
- ▶ Ordinary procedure calls: *call* $\pi(a_1, \dots, a_r)$,
 $\pi \in \{\pi_1, \dots, \pi_k\}$

Now, we introduce **procedural parameters** allowing

- ▶ Procedure declarations: *proc* $\pi(p_1, \dots, p_r, \psi_1, \dots, \psi_q)$
- ▶ Formal procedure calls: *call* $\psi(a_1, \dots, a_r, \bar{\pi}_1, \dots, \bar{\pi}_q)$,
where $\bar{\pi}_1, \dots, \bar{\pi}_q \in \{\pi_1, \dots, \pi_k\}$

Handling Procedural Parameters in IDFA (1)

Key idea

- ▶ The semantics of a formal procedure call ψ is considered the meet of the semantics of the ordinary procedures it might call at runtime:

$$\bigsqcap \{ \llbracket e_r \rrbracket_{Stk}^* \circ \nu - \llbracket end(\pi) \rrbracket_{Stk} \circ \llbracket e_c \rrbracket_{Stk}^* \mid \psi \text{ might call } \pi \}$$

Technically, this means

- ▶ replacing formal procedure calls by the set of ordinary procedure calls they may stand for.

Handling Procedural Parameters in IDFA (2)

Algorithmically

- ▶ this set of procedures can be computed by a **suitable pre-process**.
- ▶ depending on the expressive power of the programming language considered, the specific program under investigation, and the power of the analysis algorithm, the computed set of procedures can be **exact** or a **safe approximation**.

Overall

- ▶ exploiting **precomputed calling information** for formal procedure call **reduces** the **analysis** of programs **with formal procedure calls** to the **analysis** of programs **without formal procedure calls**.

Replacing the Basic 2nd Order Analysis

...of the $IMaxFP_{Stk}$ approach characterized by:

Equation System 12.1.3.1 (2nd Order $IMaxFP_{Stk}$)

$$\llbracket n \rrbracket_{Stk} = \begin{cases} Id_{STACK} & \text{if } n \in \{\mathbf{s}_0, \dots, \mathbf{s}_k\} \\ \bigcap \{ \llbracket (m, n) \rrbracket_{Stk} \circ \llbracket m \rrbracket_{Stk} \mid m \in pred_{flowGraph}(n)(n) \} & \\ \text{otherwise} & \end{cases}$$

and

$$\llbracket e \rrbracket_{Stk} = \begin{cases} \llbracket e \rrbracket_{Stk}^* & \text{if } e \in E \setminus E_{call} \\ \llbracket e_r \rrbracket_{Stk}^* \circ \llbracket end(callee(e)) \rrbracket_{Stk} \circ \llbracket e_c \rrbracket_{Stk}^* & \text{otherwise} \end{cases}$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

1154/18

By the Enhanced 2nd Order Analysis

...characterized by:

Equation System 12.2.1 (Enh'd 2nd Order *IMaxFP*)

$$\llbracket n \rrbracket_{Stk} = \begin{cases} Id_{STACK} & \text{if } n \in \{\mathbf{s}_0, \dots, \mathbf{s}_k\} \\ \bigcap \{ \llbracket (m, n) \rrbracket_{Stk} \circ \llbracket m \rrbracket_{Stk} \mid m \in pred_{flowGraph(n)}(n) \} & \\ \text{otherwise} & \end{cases}$$

and $\llbracket e \rrbracket_{Stk} =$

$$\begin{cases} \llbracket e \rrbracket_{Stk}^* & \text{if } e \in E \setminus E_{call} \\ \llbracket e_r \rrbracket_{Stk}^* \circ \llbracket end(callee(e)) \rrbracket_{Stk} \circ \llbracket e_c \rrbracket_{Stk}^* & \text{if } e \in E_{call}, \text{ ordinary procedure call} \\ \bigcap \{ \llbracket e_r \rrbracket_{Stk}^* \circ \llbracket end(\pi) \rrbracket_{Stk} \circ \llbracket e_c \rrbracket_{Stk}^* \mid \psi \text{ might call } \pi \} & \text{if } e \in E_{call}, \text{ formal procedure call} \end{cases}$$

where the set of procedures that may be called by a **formal call** is computed in a **separate** and **independent preprocess**.

Suitable Preprocesses

...are sketched in [Chapter 15](#) for an [object-oriented setting](#):

- ▶ [Class Hierarchy Analysis](#) (cf. [Chap. 15.2.1](#))
- ▶ [Rapid Type Analysis](#) (cf. [Chap. 15.2.2](#))

Chapter 12.3

Adding Reference Parameters

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

Reference Parameters

Handling the effect of [reference parameters](#)

- ▶ The effect of [reference parameters](#) can be encoded in the [DFA functionals](#) of the application problems.

[Algorithmically](#)

- ▶ This requires [may](#) and [must alias information](#) of [variables](#) and [parameters](#), which can be computed by suitable pre-processes (cf. [Chap. 14](#)).
- ▶ The [computed alias information](#) is then fed into the [generic algorithms](#) of the [toolkit](#) of [Chapter 12.1.7](#) via the definitions of the DFA functions of the application problems (cf. [Chap. 12.5](#)).

Chapter 12.4

Adding Static Procedure Nesting

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

12.5

1159/18

Static Procedure Nesting

Static nesting of procedures

- ▶ introduces **statically nested definitions** (or **declarations**) of **local variables**: Variables are no longer either global (declared in the main program) or local (declared in a procedure) but 'relatively global.'

Algorithmically

- ▶ The effects of relatively global variables can be encoded in the **DFA functionals** of the application problems.

Alternatively

- ▶ De-nesting of procedures by a **suitable preprocess**.
- ▶ This way, the analysis of programs with static procedure nesting is reduced to analysing programs without static procedure nesting.

Chapter 12.5

Applications

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

Chapter 12.5.1

Interprocedural Availability

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

1162/18

Preliminaries

In the following we **assume**:

- ▶ No static procedure nesting, no procedural parameters.
- ▶ $MstAliases_G(v)$ und $MayAliases_G(v)$ denote the sets of **must-aliases** and **may-aliases** different from v .

These notions can straightforward be **extended** to terms t :

- ▶ A term t' is a **must-alias** (**may-alias**) of t , if t' results from t by replacing of variables by variables that are must-aliases (may-aliases) of each other.

This allows us to feed alias information in a **parameterized fashion** into the definitions of **DFA functionals** and **return functionals** and to take their effects during the analysis into account.

Useful Notations

We define:

- ▶ $GlobVar(S)$: the set of **global variables** of S , i.e., the set of variables which are declared in the main program of S . They are accessible in each procedure of S .
- ▶ $Var(t)$: the set of variables occurring in t .
- ▶ $LhsVar(e)$: the **left hand side variable** of the assignment of edge e .
- ▶ $GlobId(t)$ and $LocId(t)$: abbreviations of $GlobVar(S) \cap Var(t)$ and $Var(t) \setminus GlobVar(S)$.

Useful Notations (Cont'd)

- ▶ *NoGlobalChanges* : $E^* \rightarrow \text{IB}$: indicates that a modification of variable $v \in \text{Var}(t)$ by e will not be visible after finishing the call as the relevant memory location of v is local for the currently active call.
- ▶ *PotAccessible* : $S \rightarrow \text{IB}$: indicates that the memory locations of all variables $v \in \text{Var}(t)$, which are accessible immediately before entering G remain accessible after entering it, either by referring to v itself or by referring to one of its *must-aliases*.

Local Predicates

The definition of the preceding functions utilizes the predicates $Transp_{Locld}$ and $Transp_{Globld}$ defined as follows:

$$Transp_{Locld}(e) =_{df} Locld(t) \cap MayAliases_{flowGraph(e)}(LhsVar(e)) = \emptyset$$

$$Transp_{Globld}(e) =_{df} Globld(t) \cap (LhsVar(e) \cup MayAliases_{flowGraph(e)}(LhsVar(e))) = \emptyset$$

This allows us to define:

$\forall e \in E^*. NoGlobalChanges(e)$

$$=_{df} \begin{cases} true & \text{if } e \in E_c^* \cup E_r^* \\ Transp_{Locld}(n) \wedge Transp_{Globld}(n) & \text{otherwise} \end{cases}$$

Parameterized Local Predicates

...parameterized wrt alias information:

$$\forall e \in E^*. A\text{-Comp}_e =_{df} \text{Comp}_e \vee \text{Comp}_e^{MstAI}$$

$$\forall e \in E^*. A\text{-Transp}_e =_{df} \text{Transp}_e \wedge \begin{cases} \text{true} & \text{if } e \in E_{call}^* \\ \text{Transp}_e^{MayAI} & \text{otherwise} \end{cases}$$

Parameterized Local Predicates (Cont'd)

Intuitively

- ▶ $A\text{-Comp}_e$ is **true** for t , if t itself (i.e., Comp_e) or one of its must-aliases is computed at edge e (i.e., $\text{Comp}_e^{\text{MstAl}}$).
- ▶ $A\text{-Transp}_e$, $e \in E^* \setminus E_{\text{call}}^*$, is **true** for t , if neither an operand of t (i.e., Transp_e) nor one of its may-aliases is modified by the statement at edge e (i.e., $\text{Transp}_e^{\text{MayAl}}$).
- ▶ For call and return edges $e \in E_{\text{call}}^*$, $A\text{-Transp}_e$ is **true** for t , if no operand of t is modified (i.e., Transp_e). This makes the difference between ordinary assignments and reference parameters and parameter transfers to reference parameters; the latter are updates of pointers that leave the memory invariant except of that update.

Interprocedural Availability

Key Ingredients of the DFA Specification:

1. Data flow lattice:

$$(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathbb{B}^2, \wedge, \vee, \leq, (false, false), (true, true))$$

2. Data flow functional:

$$\llbracket \cdot \rrbracket_{av}^* : E^* \rightarrow (\mathbb{B}^2 \rightarrow \mathbb{B}^2) \text{ defined by}$$

$$\forall e \in E^* \forall (b_1, b_2) \in \mathbb{B}^2. \llbracket e \rrbracket_{av}^*(b_1, b_2) =_{df} (b'_1, b'_2)$$

where

$$b'_1 =_{df} A\text{-Transp}_e \wedge (A\text{-Comp}_e \vee b_1)$$

$$b'_2 =_{df} \begin{cases} b_2 \wedge \text{NoGlobalChanges}_e & \text{if } e \in E^* \setminus E_c^* \\ true & \text{otherwise} \end{cases}$$

Interprocedural Availability (Cont'd)

3. Return functional:

$\mathcal{R}_{av} : E_{call} \rightarrow (\mathbb{IB}^2 \times \mathbb{IB}^2 \rightarrow \mathbb{IB}^2)$ defined by

$\forall e \in E_{call} \forall ((b_1, b_2), (b_3, b_4)) \in \mathbb{IB}^2 \times \mathbb{IB}^2.$

$\mathcal{R}_{av}(e)((b_1, b_2), (b_3, b_4)) =_{df} (b_5, b_6)$ where

$$b_5 =_{df} \begin{cases} b_3 & \text{if } PotAccessible(callee(e)) \\ (b_1 \vee A-Comp_e) \wedge b_4 & \text{otherwise} \end{cases}$$

$$b_6 =_{df} b_2 \wedge b_4$$

Findings on Interprocedural Availability

Lemma 12.5.1.1

1. The lattice \mathbb{B}^2 and the induced lattice of functions satisfy the descending chain condition.
2. The functionals $\llbracket \cdot \rrbracket_{av}^*$ and \mathcal{R}_{av} are distributive.

This means, the preconditions of the [Interprocedural Coincidence Theorem 12.1.4.4](#) and the [Termination Theorem 12.1.5.4](#) are satisfied.

Chapter 12.5.2

Interprocedural Copy Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

12.3

12.4

1172/18

Interprocedural Simple Constants – Naively

Key Ingredients of the DFA Specification:

1. Data flow lattice:

$$(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\Sigma, \sqcap, \sqcup, \sqsubseteq, \sigma_{\perp}, \sigma_{\top})$$

2. Data flow functional: $\llbracket \cdot \rrbracket_{sc}^* : E \rightarrow (\Sigma \rightarrow \Sigma)$ defined by

$$\forall e \in E. \llbracket e \rrbracket_{sc}^* =_{df} \theta_e$$

3. Return functional: $\mathcal{R}_{sc} : E_{call} \rightarrow (\Sigma \times \Sigma \rightarrow \Sigma)$ defined by

$$\forall e \in E_{call} \forall (\sigma_1, \sigma_2) \in \Sigma \times \Sigma. \mathcal{R}_{sc}(e)(\sigma_1, \sigma_2) =_{df} \sigma_3$$

where

$$\forall x \in Var. \sigma_3(x) =_{df} \begin{cases} \sigma_2(x) & \text{if } x \in GlobVar(S) \\ \sigma_1(x) & \text{otherwise} \end{cases}$$

Problems, Consequences

Unfortunately

- ▶ The preceding **DFA specification** for **interprocedural simple constants** does not induce a terminating analysis since the lattice of functions on Σ does not satisfy the descending chain condition.

In practice, thus

- ▶ simpler variants of the interprocedural constant propagation problem are considered, e.g., **interprocedural copy** and **linear constants**.

Copy Constants, Linear Constants Recalled

Intuitively, a term is a

- ▶ **copy constant** at a program point, if it is a source-code constant or an operator-less term which is itself a copy constant (cf. Chap. 5.5)
- ▶ **linear constant** at a program point, if it is a source-code constant or of the form $a * x + b$ with a , b source-code constants and x a linear constant (cf. Chap. 5.4).

Interprocedural Copy Constants: Findings (1)

We have:

- ▶ The number of source-code constants and program variables are finite.
- ▶ Hence, the lattice of functions induced by the relevant sublattice of Σ for copy constants is finite satisfying thus the descending chain condition.
- ▶ Hence, the generic 2nd order and 1st order DFA algorithms for copy constants terminate with the $IMaxFP_{Stk}$ solution for copy constants.
- ▶ Last but not least, the computable $IMaxFP_{Stk}$ solution for copy constants coincides with the specifying $IMOP_{Stk}$ solution, since the DFA functions $\llbracket \cdot \rrbracket_{cc}^*$ and the return functions $\mathcal{R}_{cc}(\cdot)$ for copy constants are distributive.

Interprocedural Copy Constants: Findings (2)

Lemma 12.5.2.1

1. The lattice $\Sigma_{cc} \subseteq \Sigma$ and its induced lattice of functions satisfy the descending chain condition.
2. The functionals $\llbracket \cdot \rrbracket_{cc}^*$ and \mathcal{R}_{cc} are distributive.

This means, the preconditions of the [Interprocedural Coincidence Theorem 12.1.4.4](#) and the [Termination Theorem 12.1.5.4](#) are satisfied.

Chapter 12.6

Summary, Looking Ahead

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2

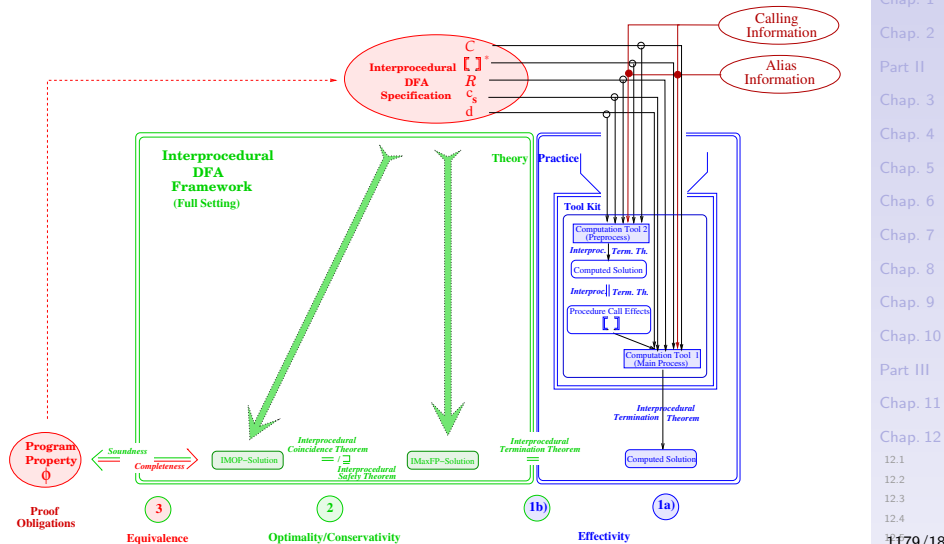
12.3

12.4

Interprocedural DFA: A Holistic Uniform View

...considered from a **holistic perspective** yields a

► Uniform Framework and Toolkit View

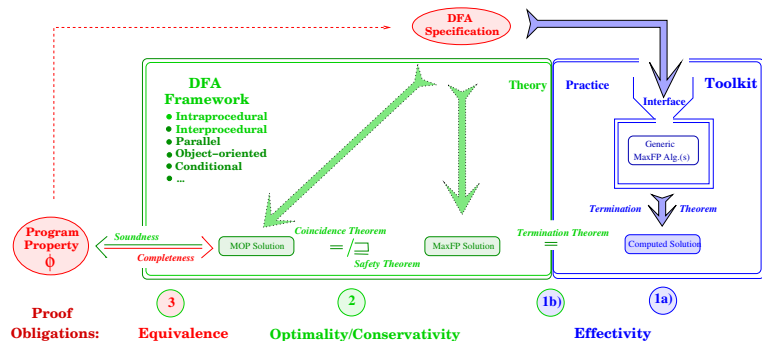


Overall, this provides

...further evidence for the claim of Chapter 3.8 that

- The Uniform Framework and Toolkit View of DFA

...is achievable beyond the base case of intraprocedural DFA.



Chapter 12.7

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

12.1

12.2




12.3

12.4



12.5

1181/18




Further Reading for Chapter 12 (1)

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 12, Interprocedural Analysis)
-  Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 11, Interprocedural Analysis and Optimization)
-  Thomas Ball, Sriram K. Rajamani. *Bebop: A Path-Sensitive Interprocedural Dataflow Engine*. In Proceedings of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), 97-103, 2001.




Further Reading for Chapter 12 (2)

-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 7, Introduction to Interprocedural Data Flow Analysis, Chapter 8, Functional Approach to Interprocedural Data Flow Analysis; Chapter 9, Value-Based Approach to Interprocedural Data Flow Analysis)
-  Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998. (Chapter 10, Interprocedural Code Motion: The Transformations, Chapter 11, Interprocedural Code Motion: The IDFA-Algorithms)



Further Reading for Chapter 12 (3)

-  Jens Knoop. *Formal Callability and its Relevance and Application to Interprocedural Data Flow Analysis*. In Proceedings of the 6th IEEE International Conference on Computer Languages (ICCL'98), 252-261, 1998.
-  Jens Knoop. *From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Springer-V., LNCS 1579, 360-374, 1999.
-  Jens Knoop, Bernhard Steffen. *The Interprocedural Coincidence Theorem*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-V., LNCS 641, 125-140, 1992.

Further Reading for Chapter 12 (4)

-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 19, Interprocedural Analysis and Optimization)
-  Tom Reps, Susan Horwitz, Mooly Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 49-61, 1995.
-  Mooly Sagiv, Tom Reps, Susan Horwitz. *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation*. In Proceedings of the 6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95), Springer-V., LNCS 915, 651-665, 1995.

Further Reading for Chapter 12 (5)

-  Helmut Seidl, Christian Fecht. *Interprocedural Analyses: A Comparison*. The Journal of Logic Programming 43:123-156, 2000.
-  Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.3, The Functional Approach to Interprocedural Analysis, 196-209.

Chapter 13

The Context Information Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

1187/18

Motivation

In this chapter, we complement the **functional approach** for **IDFA** by sketching a selection of so-called

- ▶ **context information approaches.**

Context information approaches

- ▶ allow the user to control the trade-off between power and performance.
- ▶ promise to be more efficient in practice.
- ▶ are heuristic in nature.

Outline

The presentation follows the one of [Nielson, Nielson, and Hankin \(2005\)](#) using their (extended) setting and notation of [Chapter 2](#).

We start by extending the programming language **WHILE** by introducing programs with

- ▶ top-level declarations of global mutually recursive procedures and
- ▶ two parameters: a call-by-value and a call-by-result parameter.

Note: Extensions to multiple call-by-value, call-by-result, and call-by-value-result parameters are straightforward.

Chapter 13.1

Preliminaries, the Setting

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.1.1

13.1.2

1190/18

Syntax: Introducing Procedures

Extended WHILE-Language $WHILE_{\pi}$:

$$P_{\star} ::= \text{begin } D_{\star} \ S_{\star} \ \text{end}$$
$$D ::= D; D \mid \text{proc } p(\text{val } x; \text{ res } y) \text{ is}^{\ell_n} \ S \ \text{end}^{\ell_x}$$
$$S ::= \dots \mid [\text{call } p(a, z)]_{\ell_r}^{\ell_c}$$

Labeling scheme

► Procedure declarations

ℓ_n : for entering the body

ℓ_x : for exiting the body

► Procedure calls

ℓ_c : for the call

ℓ_r : for the return

Assumptions

We *assume* that

- ▶ $WHILE_{\pi}$ is statically scoped.
- ▶ The *parameter mechanism* is
 - ▶ *call-by-value* for the first parameter.
 - ▶ *call-by-result* for the second parameter.
- ▶ Procedures may be *mutually recursive*.
- ▶ Procedure names are *pairwise disjoint*.
- ▶ There are only calls to *internally declared procedures*.
- ▶ Programs are *uniquely labelled*.

Illustrating Example

The procedure `proc fib` computing the Fibonacci numbers:

```
begin
  proc fib(val z,u; res v) is
    if z<3 then
      (v:=u+1; r:=r+1)
    else (
      call fib (z-1,u,v);
      call fib (z-2,v,v)
    )
  end;
  r:=0;
  call fib(x,0,y)
end
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

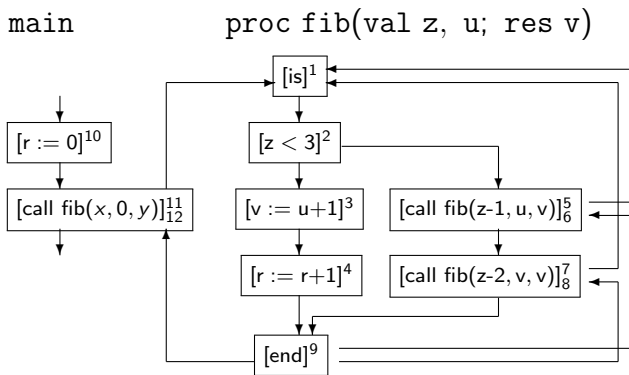
13.1.1

13.1.2

1193/18

The Interprocedural Flow Graph

...of procedure `proc fib` embedded in a calling context with some main program `main`:



Notions and Notations for Flow Graphs (1)

...for procedure calls and procedure declarations:

	$[\text{call } p(a, z)]_{\ell_r}^{\ell_c}$	$\text{proc } p(\text{val } x; \text{ res } y) \text{ is}^{\ell_n} S \text{ end}^{\ell_x}$
init	ℓ_c	ℓ_n
final	$\{\ell_r\}$	$\{\ell_x\}$
blocks	$\{[\text{call } p(a, z)]_{\ell_r}^{\ell_c}\}$	$\{\text{is}^{\ell_n}\} \cup \text{blocks}(S) \cup \{\text{end}^{\ell_x}\}$
labels	$\{\ell_c, \ell_r\}$	$\{\ell_n, \ell_x\} \cup \text{labels}(S)$
flow	$\{(\ell_c; \ell_n), (\ell_x; \ell_r)\}$	$\{(\ell_n, \text{init}(S))\} \cup \text{flow}(S) \cup \{\ell, \ell_x \mid \ell \in \text{final}(S)\}$

Note: $(\ell_c; \ell_n)$ and $(\ell_x; \ell_r)$ denote a new kind of flow, **interprocedural flow**:

- ▶ $(\ell_c; \ell_n)$ is the flow corresponding to **calling** a procedure at ℓ_c and entering the procedure body at ℓ_n and
- ▶ $(\ell_x; \ell_r)$ is the flow corresponding to exiting a procedure body at ℓ_x and **returning** to the call at ℓ_r .

Remark: Intraprocedural flow uses ‘,’ while interprocedural flow uses ‘;’.

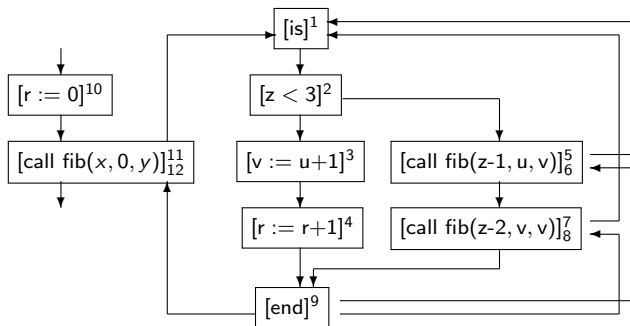
Notions and Notations for Flow Graphs (2)

...for (whole) programs:

	P_*
init_*	$\text{init}(S_*)$
final_*	$\text{final}(S_*)$
blocks_*	$\bigcup \{ \text{blocks}(p) \mid \text{proc } p(\text{val } x; \text{ res } y) \text{ is}^{\ell_n} S \text{ end}^{\ell_x} \text{ is in } D_* \} \cup \text{blocks}(S_*)$
labels_*	$\bigcup \{ \text{labels}(p) \mid \text{proc } p(\text{val } x; \text{ res } y) \text{ is}^{\ell_n} S \text{ end}^{\ell_x} \text{ is in } D_* \} \cup \text{labels}(S_*)$
flow_*	$\bigcup \{ \text{flow}(p) \mid \text{proc } p(\text{val } x; \text{ res } y) \text{ is}^{\ell_n} S \text{ end}^{\ell_x} \text{ is in } D_* \} \cup \text{flow}(S_*)$
Lab $_*$	labels_*

$\text{inter-flow}_* = \{ (\ell_c, \ell_n, \ell_x, \ell_r) \mid P_* \text{ contains } [\text{call } p(a, z)]_{\ell_c}^{\ell_r} \text{ as well as } \text{proc } p(\text{val } x; \text{ res } y) \text{ is}^{\ell_n} S \text{ end}^{\ell_x} \}$

Illustrating Example



$$\begin{aligned} flow_{\star} = & \{(1, 2), (2, 3), (3, 4), (4, 9), \\ & (2, 5), (5, 1), (9, 6), (6, 7), (7, 1), (9, 8), (8, 9), \\ & (11, 1), (9, 12), (10, 11)\} \end{aligned}$$

$$inter-flow_{\star} = \{(11, 1, 9, 12), (5, 1, 9, 6), (7, 1, 9, 8)\}$$

Metavariables for Forward/Backward Analyses

Forward Analyses:

- ▶ $F = flow_{\star}$
- ▶ $E = init_{\star}$
- ▶ $IF = inter-flow_{\star}$

Backward Analyses:

- ▶ $F = flow_{\star}^R$
- ▶ $E = final_{\star}$
- ▶ $IF = inter-flow_{\star}^R$

Towards Interprocedural DFA

New **transfer functions** dealing with **interprocedural flow** are required:

- ▶ For each **procedure call** $[\text{call } p(a, z)]_{\ell_r}^{\ell_c}$ we require two transfer functions

- ▶ f_{l_c} and f_{l_r}

corresponding to **calling** the procedure and **returning** from the call.

- ▶ For each **procedure definition** $\text{proc } p(\text{val } x; \text{ res } y) \text{ is}^{\ell_n} S \text{ end}^{\ell_x}$ we require two transfer functions

- ▶ f_{l_n} and f_{l_x}

corresponding to **entering** and **exiting** the procedure body.

Chapter 13.1.1

Naive Interprocedural DFA

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.1.1

13.1.2

1200/18

Interprocedural DFA: Naive Formulation (1)

- ▶ Treat the three kinds of flow, (l_1, l_2) , $(l_c; l_n)$, $(l_x; l_r)$ in the same way.
- ▶ Assume that the 4 transfer functions associated with procedure calls and procedure definitions are given by the identity functions, i.e., the parameter-passing is effectively ignored.

Then:

Naive Interprocedural *MaxFP*-Equation System:

$$\begin{aligned} A_o(l) &= \bigsqcap \{A_\bullet(l') \mid (l', l) \in F \vee (l'; l) \in F\} \sqcap \iota_E^l \\ A_\bullet(l) &= f_\ell^A(A_o(l)) \end{aligned}$$

where

$$\iota_E^l =_{df} \begin{cases} \perp & \text{if } l \in E \\ \top & \text{if } l \notin E \end{cases}$$

Interprocedural DFA: Naive Formulation (2)

Given the [previous assumptions](#) we have:

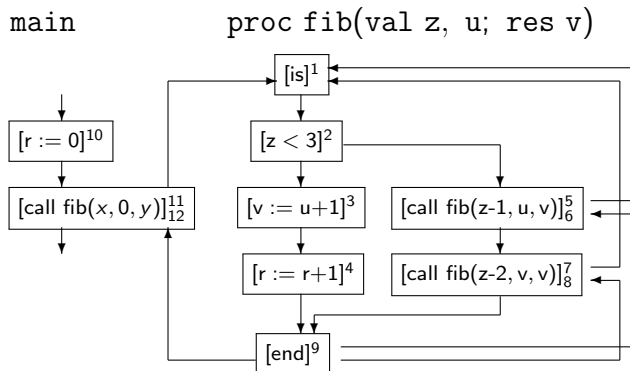
- ▶ Both procedure calls $(l_c; l_n)$ and procedure returns $(l_x; l_r)$ are treated like 'goto's'.
- ▶ There is no mechanism for ensuring that information flowing along $(l_c; l_n)$ flows back along $(l_x; l_r)$ to the same call.
- ▶ Intuitively, the equation system considers a much **too large set of 'paths'** through the program and hence will be **grossly imprecise** (although formally on the safe side).

Chapter 13.1.2

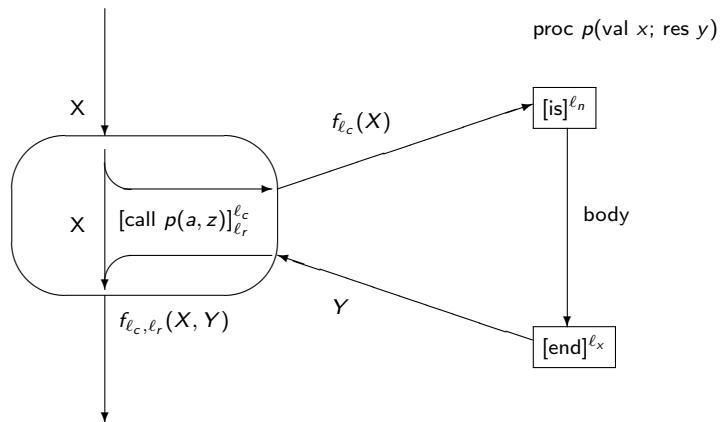
Interprocedurally Valid and Complete Paths

Interprocedurally Valid Program Paths

We want to overcome the shortcoming of the naive formulation by restricting attention to paths that have the proper nesting of procedure calls and exits. Important are the notions of **matching procedure entries** and **exits** and of **complete** and **valid paths**.



Matching Procedure Entries and Exits



Complete Paths

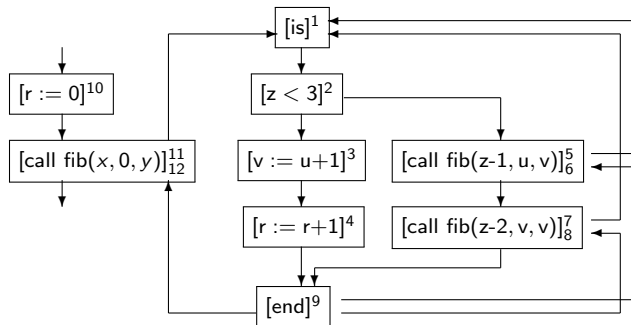
A **complete path** from l_1 to l_2 in P_\star has proper nesting of procedure entries and exits; and a procedure returns to the point where it was called:

$$\begin{array}{ll} CP_{l_1, l_2} \longrightarrow l_1 & \text{whenever } l_1 = l_2 \\ CP_{l_1, l_3} \longrightarrow l_1, CP_{l_2, l_3} & \text{whenever } (l_1, l_2) \in \text{flow}_\star \\ CP_{l_c, l} \longrightarrow l_c, CP_{l_n, l_x}, CP_{l_r, l} & \text{whenever } (l_c, l_n, l_x, l_r) \in \text{inter-flow}_\star \end{array}$$

Recall:

$(l_c, l_n, l_r, l_x) \in \text{inter-flow}_\star$, if P_\star contains $[\text{call } p(a, z)]_{l_r}^{l_c}$ as well as $\text{proc } p(\text{val } x; \text{res } y) \text{ is}^{l_n} S \text{ end}^{l_x}$.

Illustrating Example: Complete Paths



$CP_{10,12} \rightarrow 10,$	$CP_{11,12}$	$CP_{3,9} \rightarrow 3,$	$CP_{4,9}$
$CP_{11,12} \rightarrow 11,$	$CP_{1,9}, CP_{12,12}$	$CP_{4,9} \rightarrow 4,$	$CP_{9,9}$
$CP_{1,9} \rightarrow 1,$	$CP_{2,9}$	$CP_{5,9} \rightarrow 5,$	$CP_{1,9}, CP_{6,9}, CP_{12,12} \rightarrow 12$
$CP_{2,9} \rightarrow 2,$	$CP_{3,9}$	$CP_{6,9} \rightarrow 6,$	$CP_{7,9}, CP_{9,9} \rightarrow 9$
$CP_{2,9} \rightarrow 2,$	$CP_{5,9}$	$CP_{7,9} \rightarrow 7,$	$CP_{1,9}, CP_{8,9}$
		$CP_{8,9} \rightarrow 8,$	$CP_{9,9}$

Chapter 13.2

MVP Approach and *MVP* Solution

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

1208/18

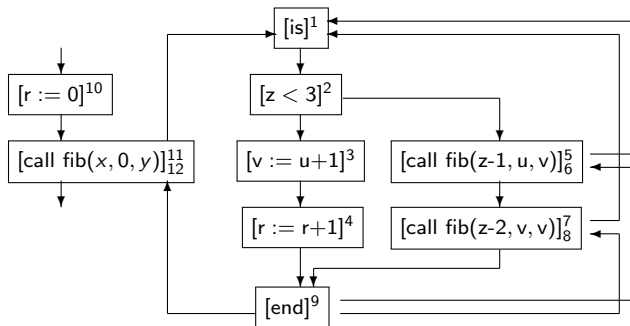
Valid Paths

A **valid path** starts at the entry node init_* of P_* , all the procedure exits match the procedure entries but some procedures might be entered but not yet exited:

$VP_* \longrightarrow VP_{\text{init}_*,l}$	whenever $l \in \text{Lab}_*$
$VP_{l_1,l_2} \longrightarrow l_1$	whenever $l_1 = l_2$
$VP_{l_1,l_3} \longrightarrow l_1, VP_{l_2,l_3}$	whenever $(l_1, l_2) \in \text{flow}_*$
$VP_{l_c,l} \longrightarrow l_c, CP_{l_n,l_x}, VP_{l_r,l}$	whenever $(l_c, l_n, l_x, l_r) \in \text{inter-flow}_*$
$VP_{l_c,l} \longrightarrow l_c, VP_{l_n,l}$	whenever $(l_c, l_n, l_x, l_r) \in \text{inter-flow}_*$

Note: The **valid paths** are generated by the non-terminal VP_* .

Illustrating Example: Valid Paths



$CP_{10,12} \rightarrow 10, CP_{11,12}$
 $CP_{11,12} \rightarrow 11, CP_{1,9}, CP_{12,12}$
 $CP_{1,9} \rightarrow 1, CP_{2,9}$
 $CP_{2,9} \rightarrow 2, CP_{3,9}$
 $CP_{2,9} \rightarrow 2, CP_{5,9}$
 $CP_{3,9} \rightarrow 3, CP_{4,9}$
 $CP_{4,9} \rightarrow 4, CP_{9,9}$
 $CP_{5,9} \rightarrow 5, CP_{1,9}, CP_{6,9}, CP_{12,12} \rightarrow 12$
 $CP_{6,9} \rightarrow 6, CP_{7,9}, CP_{9,9} \rightarrow 9$
 $CP_{7,9} \rightarrow 7, CP_{1,9}, CP_{8,9}$
 $CP_{8,9} \rightarrow 8, CP_{9,9}$

Some valid paths: [10,11,1,2,3,4,9,12] and [10,11,1,2,5,1,2,3,4,9,6,7,1,2,3,4,9,8,9,12]

A non-valid path: [10,11,1,2,5,1,2,3,4,9,12]

Meet over Valid Paths: The MVP Solution

$$MVP_{\circ}(\ell) = \bigcap \{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in vpath_{\circ}(\ell)\}$$

$$MVP_{\bullet}(\ell) = \bigcap \{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in vpath_{\bullet}(\ell)\}$$

where

$$vpath_{\circ}(\ell) = \{[\ell_1, \dots, \ell_{n-1}] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1, \dots, \ell_n] \text{ is valid path}\}$$

$$vpath_{\bullet}(\ell) = \{[\ell_1, \dots, \ell_n] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1, \dots, \ell_n] \text{ is valid path}\}$$

Discussing the MVP Solution (1)

The **MVP solution** may be undecidable (even) for lattices satisfying the descending chain condition, just as was the case for the **MOP solution**.

Therefore, we need to reconsider the **maximal fixed point approach (MFP)** and adapt it to

- ▶ avoid considering too many paths.
- ▶ taking call context information into account.

Discussing the MVP Solution (2)

In more detail:

We have to

- ▶ reconsider the **MFP solution** and avoid taking too many invalid paths into account.

An **obvious approach** is to

- ▶ encode information about the paths taken into the data flow properties themselves.

This can be achieved by

- ▶ introducing **context information** $\delta \in \Delta$.

Chapter 13.3

Call Strings, Assumption Sets

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

1214/18

Towards the MFP Counterpart of MVP

- ▶ **Context insensitive analysis:** No context information is used.
- ▶ **Context sensitive analysis:** Context information $\delta \in \Delta$ is used.
 - ▶ **Call strings:**
 - ▶ An abstraction of the sequences of procedure calls that have been performed so far.
 - ▶ **Example:** The program point where the call was initiated.
 - ▶ **Assumption sets:**
 - ▶ An abstraction of the states in which previous calls have been performed.
 - ▶ **Example:** An abstraction of the actual parameters of the call.

Chapter 13.3.1

Call Strings

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

1216/18

Call Strings δ as Context Information Δ

- ▶ Encode the path taken.
- ▶ Only record flows of the form $(\ell_c; \ell_n)$ corresponding to a procedure call.
- ▶ we take as context $\Delta = \text{Lab}_*^*$ where the most recent label ℓ_c of a procedure call is at the right end.
- ▶ Elements of Δ are called **call strings**.
- ▶ The sequence of labels $\ell_c^1, \ell_c^2, \dots, \ell_c^m$ is the **call string** leading to the current call which happened at ℓ_c^m ; the previous calls where at $\ell_c^2 \dots \ell_c^1$. If $m = 0$ then no calls have been performed so far.

For the example program the following **call strings** are of interest:

$\Lambda, [11], [11, 5], [11, 7], [11, 5, 5], [11, 5, 7], [11, 7, 5], [11, 7, 7], \dots$

The Adapted MFP Equation System

The Adapted MFP-Equation System:

$$\begin{aligned}A_o(\ell) &= \sqcap \{A_\bullet(\ell') \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\} \sqcap \hat{\iota}_E^\ell \\A_\bullet(\ell) &= \widehat{f}_\ell^A(A_o(\ell))\end{aligned}$$

where

- ▶ $\hat{L} = \Delta \rightarrow L$ maps a context to a data flow property (i.e., a data flow lattice element)
- ▶ each transfer function \widehat{f}_i is given by $\widehat{f}_i(\hat{l})(\delta) = f_i(\hat{l}(\delta))$ (i.e., \widehat{f}_i adapts resp. specializes f_i to the call context δ)

▶

$$\hat{\iota}_E^\ell =_{df} \begin{cases} \iota_E^\ell & \text{if } \delta = \Lambda \\ \top & \text{otherwise} \end{cases}$$

Making it Practical: Bounding Call Strings

Problem: Call strings can be arbitrarily long (recursive calls)

Solution: Truncate the call strings to have length of at most k for some fixed number k

In practice:

- ▶ $\Delta = \text{Lab}_*^{\leq k}$, i.e. call strings of bounded length k .
- ▶ $k = 0$ — remember no procedure call (**context insensitive analysis**):
 - ▶ Λ (the (only) call string is the **empty string**)
- ▶ $k = 1$ — remember the **last** procedure call:
 - ▶ $\Lambda, [11], [5], [7]$
- ▶ $k = 2$ — remember the **last two** procedure calls:
 - ▶ $\Lambda, [11], [11, 5], [11, 7], [5, 5], [5, 7], [7, 5], [7, 7]$
- ▶ $k = 3$ — remember **last three**...

Chapter 13.3.2

Assumption Sets

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

1220/18

Assumption Sets δ as Context Information Δ

Instead of describing a path directly in terms of the calls being performed (as a [call string](#) does), information about the state in which a call was made can be stored (as an [assumption set](#) does).

For a detailed account of the [assumption set](#) approach refer to

- ▶ Flemming Nielson, Hanne Riis Nielson, Chris Hankin.
Principles of Program Analysis. 2nd edition, Springer-V., 2005. (Chapter 2.5.5, Assumption Sets as Context)

Chapter 13.3.3

Advanced Topics

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

Advanced Topics

...of **interprocedural program analysis** and a glimpse on how they can be addressed by **static program analysis**.

- ▶ Function pointers
- ▶ Virtual function calls
- ▶ Overloaded functions

Function Pointers

Values of function pointer variables

The value of a function pointer variable is the address of a function. At run-time different values can be assigned to pointer variables.

Interprocedural Control Flow

Any function with the same signature (=parameter types) can be potentially called by using a function pointer.

Program analysis can reduce the number of functions that may be called at run-time by computing the set of possible pointer values assigned to function pointer variables in a given program.

Virtual Function Calls & Overloaded Functions

...in [object-oriented programming](#).

Virtual function calls

By taking the class hierarchy into account, we can limit the methods that can be called to the set of overriding methods of subclasses. Program analysis can further reduce the number of methods that may be called at run-time.

Overloaded functions

Calls to overloaded functions are resolved at compile time.

Chapter 13.4

The Cloning-based Approach

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

1226/18

Cloning-based Approaches

Especially popular

...for object-oriented and points-to analyses.

Key idea

...distinguishing contexts via [cloning](#).

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

13.1

13.2

13.3

1227/18

Applications

- ▶ *k*-object sensitive analysis for object-oriented programs (e.g., [MRR'02,SBL'11]).
- ▶ **Pointer analyses** (e.g., [BLQ'03,WL'04,ZC'04,Wha'07,BS'09])
 - ▶ **Cloning-based pointer analyses** are often expressed in Datalog solved using specialized Datalog solvers exploiting redundancy arising from large numbers of similar contexts for high *k* values ([Wha'07,BS'09]).
 - ▶ **Contexts** are typically represented by binary decision diagrams (BDDs) ([BLQHU'03,WL'04,ZC'04]) or explicit representations from the database literature ([BS'09]).
 - ▶ **Recursion** is typically approximated in an ad hoc manner. Exceptions are the approaches of [KK'08,KMR'12].

Chapter 13.5

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13



13.1

13.2




13.3

1229/18




Further Reading for Chapter 13 (1)

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 12.1.3, Call Strings)
-  Uday P. Khedker, Bageshri Karkare. *Efficiency, Precision, Simplicity, and Generality in Interprocedural Dataflow Analysis: Resurrecting the Classical Call Strings Method*. In Proceedings of the 17th International Conference on Compiler Construction (CC 2008), Springer-V., LNCS 4959, 213-228, 2008.


Further Reading for Chapter 13 (2)

-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 7, Introduction to Interprocedural Data Flow Analysis, Chapter 8, Functional Approach to Interprocedural Data Flow Analysis; Chapter 9, Value-Based Approach to Interprocedural Data Flow Analysis)
-  Se-Won Kim, Xavier Rival, Sukyoung Ryu. *A Theoretical Framework of Sensitivity in an Abstract Interpretation Framework*. ACM Transactions on Programming Languages and Systems 40(3), 13:1-44, 2018.
-  Ravi Mangal, Mayur Naik, Hongseok Yang. *A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join*. In Proceedings of the 23rd European Symposium on Programming (ESOP 2014), Springer-V., LNCS 8410, 513-533, 2014.


Further Reading for Chapter 13 (3)

-  Matthew Might, Yannis Smaragdakis, David Van Horn. *Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. OO Program Analysis*. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), ACM SIGPLAN Notices 45(6):305-315, 2010.
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 2.5, Interprocedural Analysis)
-  Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.3, The Call-String Approach to Interprocedural Analysis, 210-217.



References for Chapter 13 (4)

-  Yannis Smaragdakis, Martin Bravenboer, Ondřej Lhoták. *Pick Your Contexts Well: Understanding Object-Sensitivity*. In Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), 17-30, 2011.




Selected References for Chapter 13.4

-  Marc Berndl, Ondřej Lhoták, Feng Qian. Laurie Hendren, Navindra Umanee. *Points-to Analysis using BDDs*. In Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003), ACM SIGPLAN Notices 38:103-114, 2003.




References for Chapter 13 (5)

-  Martin Bravenboer, Yannis Smaragdakis. *Strictly Declarative Specification of Sophisticated Points-to Analyses*. In Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009), 243-262, 2009.
-  Uday P. Khedker, Bageshri Karkare. *Efficiency, Precision, Simplicity, and Generality in Interprocedural Dataflow Analysis: Resurrecting the Classical Call Strings Method*. In Proceedings of the 17th International Conference on Compiler Construction (CC 2008), Springer-V., LNCS 4959, 213-228, 2008.




References for Chapter 13 (6)

-  Uday P. Khedker, Alan Mycroft, Prashant Singh Rawat. *Liveness-based Pointer Analysis*. In Proceedings of the 19th Static Analysis Symposium (SAS 2012), Springer-V., LNCS 7460, 265-282, 2012.
-  Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to and Side-effect Analyses for JAVA*. In Proceedings of the 6th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), 1-11, 2002.
-  Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to Analysis for JAVA*. ACM Transactions on Software Engineering and Methodology 14(4):431-477, 2005.

References for Chapter 13 (7)

-  Robert Muth, Saumya K. Debray. *On the Complexity of Flow-Sensitive Dataflow Analyses*. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), 67-80, 2000.
-  Olin Shivers. *Control-Flow Analysis in Scheme*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88), ACM SIGPLAN Notices 23:164-174, 1988.
-  Yannis Smaragdakis, Martin Bravenboer, Ondřej Lhoták. *Pick Your Contexts Well: Understanding Object-Sensitivity*. In Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), 17-30, 2011.

References for Chapter 13 (8)

-  John Whaley. *Context-sensitive Pointer Analysis using Binary Decision Diagrams*. PhD Thesis, Stanford University, CA, USA, 2007.
-  John Whaley, Monica S. Lam. *Cloning-based Context-sensitive Pointer Alias Analysis using Binary Decision Diagrams*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:131-144, 2004.
-  Jianwen Zhu, Silvan Calman. *Symbolic Pointer Analysis Revisited*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:145-157, 2004.

Part IV

Extensions, Other Settings

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1238/18

Chapter 14

Aliasing

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

Chap. 14

Pointer/Alias/Shape Analysis (1)

Problem

- ▶ Ambiguous memory references interfere with an optimizer's ability to improve code.
- ▶ One major source of ambiguity is the use of pointer-based values.

Goal of Pointer/Alias/Shape Analysis

- ▶ determine for each pointer the set of memory locations to which it may refer.

Pointer/Alias/Shape Analysis (2)

Without such analysis the **compiler must assume that each pointer can refer to any addressable value**, including

- ▶ any space allocated in the run-time heap.
- ▶ any variable whose address is explicitly taken.
- ▶ any variable passed as a call-by-reference parameter.

Forms of Pointer Analysis

- ▶ points-to sets
- ▶ alias pairs
- ▶ shape analysis

Chapter 14.1

Sources of Aliasing

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1242/18

Aliasing Everywhere: Answers

...to the question 'What is an alias?' in different areas:

- ▶ A **short, easy to remember name** created for use in place of a longer, more complicated name; commonly used in e-mail applications. Also referred to as a **'nickname.'**
- ▶ A **hostname that replaces another hostname**, such as an alias which is another name for the same Internet address. For example, **www.company.com** could be an alias for **server03.company.com**.
- ▶ A **feature of UNIX shells** that enables users to define program names (and parameters) and commands with abbreviations. (e.g., **alias ls 'ls -l'**)
- ▶ In **MGI (Mouse Genome Informatics)**, an **alternative symbol or name** for part of the sequence of a known gene that resembles names for other anonymous DNA segments. For example, **D6Mit236** is an alias for **Cftr**.

Aliasing in Programs

In programs **aliasing occurs** when there exists **more than one access path** to a storage location.

An **access path** is the **l-value of an expression** that is constructed from variables, pointer dereference operators, and structure field operation operators.

Java (References)

```
A a,b;  
a = new A();  
b = a;  
b.val = 0;
```

C++ (References)

```
A& a = *new A();  
A& b = a;  
b.val = 0;
```

C++ (Pointers)

```
A* a; A* b;  
a = new A();  
b = a;  
b->val = 0;
```

C (Pointers)

```
A *a, *b;  
a = (A*)malloc(sizeof(A));  
b = a;  
b->val = 0;
```

Examples of Different Forms of Aliasing (1)

Fortran 77

EQUIVALENCE statement can be used to specify that two or more scalar variables, array variables, and/or contiguous portions of array variables begin at the same storage location.

Pascal, Modula 2/3, Java

- ▶ Variable of a reference type is restricted to have either the value nil/null or to refer to objects of a particular specified type.
- ▶ An object may be accessible through several references at once, but it cannot both have its own variable name **and** be accessible through a pointer.

Examples of Different Forms of Aliasing (2)

C/C++

- ▶ The union type specifier allows to create static aliases. A union type may have several fields declared, all of which overlap in (= share) storage.
- ▶ It is legal to compute the address of an object with the & operator (statically, automatically, or dynamically allocated).
- ▶ Allows arithmetic on pointers and considers it equivalent to array indexing

Summary

Challenges to Alias Analysis

- ▶ Aggregate data structures
- ▶ Recursive data structures
- ▶ Heap objects
- ▶ Function calls
- ▶ Function pointers
- ▶ Virtual function calls
- ▶ Type casting (for type based alias analysis)
- ▶ Pointer arithmetic

Chapter 14.2

Relevance of Aliasing for Program Optimization

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1248 / 18

Relevance of Alias Analysis to Optimization

Alias analysis refers to the determination of storage locations that may be **accessed in two or more ways**.

- ▶ **Ambiguous memory references** interfere with an optimizer's ability to improve code.
- ▶ One **major source of ambiguity** is the use of **pointer-based values**.

Goal: determine for each pointer the set of memory locations to which it may refer.

Recall: Without **alias analysis** the compiler must assume that each pointer can refer to any **addressable value**, including

- ▶ any space allocated in the run-time heap.
- ▶ any variable whose address is explicitly taken.
- ▶ any variable passed as a call-by-reference parameter.

Characterization of Aliasing

Flow-insensitive information

Binary relation on the variables in a procedure, $alias \in Var \times Var$ such that $x \text{ alias } y$ if and only if x and y

- ▶ **may** possibly at different times refer to the same memory location.
- ▶ **must** throughout the execution of the procedure refer to the same memory location.

Flow-sensitive information

A function from program points and variables to sets of abstract storage locations. $alias(p, v) = Loc$ means that at program point p variable v

- ▶ **may** refer to any of the locations in Loc .
- ▶ **must** refer to the location $l \in Loc$ with $|Loc| \leq 1$.

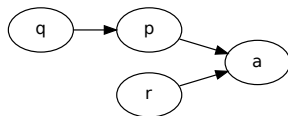
Representation of Alias Information

Representation of aliasing with pairs:

	<code>q=&p; p=&a; r=&a;</code>
complete alias pairs	<code><*q,p>, <*p,a>, <*r,a>, <**q,*p>, <**q,a>, <*p,*r>, <**q,*r></code>
compact alias pairs	<code><*q,p>, <*p,a>, <*r,a></code>
points-to relations	<code>(q,p), (p,a), (r,a)</code>

Representation of aliases and shapes of data structures:

- ▶ graphs
- ▶ regular expressions
- ▶ 3-valued logic



Chapter 14.3

Shape Analysis

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1252 / 18

Questions about Heap Contents (1)

Execution State

Let **execution state** mean the set of cells in the heap, the connections between them (via pointer components of heap cells) and the values of pointer variables in the store.

- ▶ **NULL pointers (Question 1)**: Does a pointer variable or a pointer component of a heap cell contain NULL at the entry to a statement that dereferences the pointer or component?
 - ▶ **Yes (for every state)**: Issue an error message.
 - ▶ **No (for every state)**: Eliminate a check for **NULL**.
 - ▶ **Maybe**: Warn about the potential **NULL** dereference.
- ▶ **Memory leak (Question 2)**: Does a procedure or a program leave behind unreachable heap cells when it returns?
 - ▶ **Yes (in some state)**: Issue a warning.

Questions about Heap Contents (2)

- ▶ **Aliasing (Question 3):** Do two pointer expressions reference the same heap cell?
 - ▶ **Yes (for every state):**
 - ▶ trigger a prefetch to improve cache performance
 - ▶ predict a cache hit to improve cache behavior prediction
 - ▶ increase the sets of uses and definitions for an improved liveness analysis
 - ▶ **No (for every state):** Disambiguate memory references and improve program dependence information.
- ▶ **Sharing (Question 4):** Is a heap cell shared? (within the heap)
 - ▶ **Yes (for some state):** Warn about explicit deallocation, because the memory manager may run into an inconsistent state.
 - ▶ **No (for every state):** Explicitly deallocate the heap cell when the last pointer to ceases to exist.

Questions about Heap Contents (3)

- ▶ **Reachability (Question 5):** Is a heap cell reachable from a specific variable or from any pointer variable?
 - ▶ **Yes (for every state):** Use this information for program verification.
 - ▶ **No (for every state):** Insert code at compile time that collects unreachable cells at run-time.
- ▶ **Disjointness (Question 6):** Do two data structures pointed to by two distinct pointer variables ever have common elements?
 - ▶ **No (for every state):** Distribute disjoint data structures and their computations to different processors.
- ▶ **Cyclicity (Question 7):** Is a heap cell part of a cycle?
 - ▶ **No (for every state):** Perform garbage collection of data structures by reference counting. Process all elements in an acyclic linked list in a doall-parallel fashion.

Shape Analysis

Aim of Shape Analysis

The **aim of shape analysis (SA)** is to determine a **finite representation** of heap allocated data structures which can grow arbitrarily large.

SA can determine the **possible shapes data structures** may take such as:

- ▶ lists, trees
- ▶ directed acyclic graphs, arbitrary graphs
- ▶ properties such as whether a data structure is or may be cyclic.

As **example** we shall discuss a **precise shape analysis** (from Nielson/Nielson/Hankin, PoPA, Chap. 2.6) that performs strong update and uses shape graphs to represent heap allocated data structures. It emphasises the analysis of list like data structures.

Strong Update

Here 'strong' means that an update or nullification of a pointer expression allows one to remove (kill) the existing binding before adding a new one (gen).

We shall study a powerful analysis that achieves

- ▶ Strong nullification
- ▶ Strong update

for destructive updates that destroy (overwrite) existing values in pointer variables and in heap allocated data structures in general.

Examples:

- ▶ $[x := nil]^\ell$
- ▶ $[x.sel_1 := y.sel_2]^\ell$

Extending the WHILE Language

We extend the **WHILE-language syntax** with constructs that allow to create cells in the heap.

- ▶ the cells are structured and may contain values as well as pointers to other cells.
- ▶ the data stored in cells is accessed via selectors; we assume that a finite and non-empty set **Sel** of selector names is given:

$$sel \in \text{Sel} \quad \text{selector names}$$

- ▶ we add a **new syntactic category**

$$p \in \text{PExp} \quad \text{pointer expressions}$$

- ▶ op_r is extended to allow for testing of equality of pointers.
- ▶ unary operations op_p on pointers (e.g., is-null) are added.

Abstract Syntax of Pointer Language

The **syntax** of the **WHILE-language** is **extended** to have:

$$\begin{aligned} p &::= x \mid x.sel \mid \text{null} \\ a &::= x \mid n \mid a_1 \text{ op}_a a_2 \\ b &::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\ S &::= [p:=a]^\ell \mid [\text{skip}]^\ell \\ &\quad \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \\ &\quad \mid \text{while}[b]^\ell \text{ do } S \text{ od} \\ &\quad \mid [\text{new } (p)]^\ell \\ &\quad \mid S_1; S_2 \end{aligned}$$

In the case where p contains a selector we have a **destructive update** of the heap. Statement **new creates** a new cell pointed to by p .

Shape Graphs

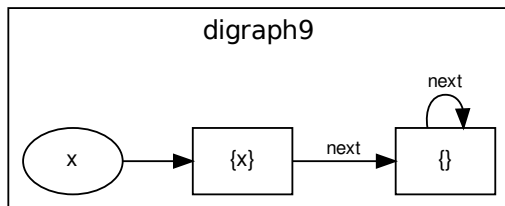
We shall introduce a **method for combining the locations** of the semantics into a **finite number of abstract locations**.

The **analysis** operates on **shape graphs** (S, H, is) consisting of:

- ▶ an **abstract state**, S (mapping variables to abstract locations).
- ▶ an **abstract heap**, H (specifying links between abstract locations).
- ▶ **sharing information**, is , for the abstract locations.

The **last component** allows us to **recover some of the imprecision** introduced by **combining many locations into one abstract location**.

Example



$g_9 = (S, H, is)$ where

$$S = \{(x, n_{\{x\}})\}$$

$$H = \{(n_{\{x\}}, next, n_{\emptyset}), (n_{\emptyset}, next, n_{\emptyset})\}$$

$$is = \emptyset$$

Abstract Locations

The **abstract locations** have the form n_X where X is a subset of the variables of Var_* :

$$\text{ALoc} = \{n_X \mid X \subseteq \text{Var}_*\}$$

A **shape graph** contains a subset of the locations of ALoc .

The **abstract location** n_\emptyset is called the **abstract summary location** and represents all the locations that cannot be reached directly from the state without consulting the heap.

Clearly, n_X and n_\emptyset represent **disjoint sets of locations** when $X \neq \emptyset$.

Invariant 1: If two abstract locations n_X and n_Y occur in the same **shape graph** then either $X = Y$ or $X \cap Y = \emptyset$ (i.e., two distinct abstract locations n_X and n_Y always represent disjoint sets of locations).

Abstract State

The **abstract state**, S , maps variables to **abstract locations**.

To maintain the naming convention for abstract locations we shall ensure that:

Invariant 2: If x is mapped to n_x by the abstract state then $x \in X$.

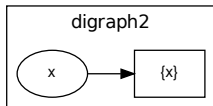
From **Invariant 1** it follows that there will be at most one abstract location in the (same) shape graph containing a given variable.

We shall only be interested in the shape of heap so we shall not distinguish between integer values, nil-pointers, and uninitialized fields; hence we can view the abstract state as an element of

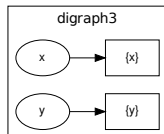
$$S \in AState = \mathcal{P}(\text{Var}_* \times ALoc)$$

Example: Creating Linked Data Structures

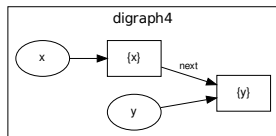
$[new(x)]^2$



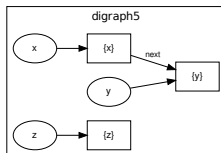
$[new(y)]^3$



$[x.next := y]^4$



$[new(z)]^5$



Abstract Heap

The **abstract heap**, H , specifies the links between the abstract locations.

The links will be specified by triples (n_V, sel, n_W) and formally we take the **abstract heap** as an element of

$$H \in AHeap = \mathcal{P}(ALoc \times Sel \times ALoc)$$

where we again not distinguish between integers, nil-pointers and uninitialized fields.

Invariant 3: Whenever (n_V, sel, n_W) and (n_V, sel, n'_W) are in the **abstract heap** then either $V = \emptyset$ or $W = W'$.

Thus the **target of a selector field** will be **uniquely determined** by the source unless the source is the abstract summary location n_\emptyset .

Sharing Information

The idea is to **specify** a **subset, is**, of the abstract locations that represents locations that are shared due to pointers **in the heap**:

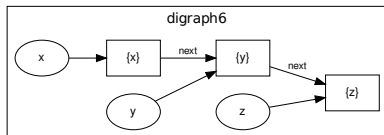
- ▶ an abstract location n_x will be included in **is** if it represents a location that is the target of more than one pointer in the heap.

In the case of the **abstract summary location**, n_\emptyset , the **explicit sharing information** clearly gives extra information:

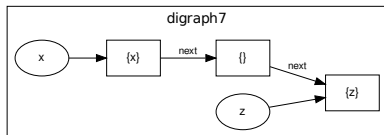
- ▶ if $n_\emptyset \in \text{is}$ then there might be a location represented by n_\emptyset that is the **target of two or more heap pointers**.
- ▶ if $n_\emptyset \notin \text{is}$ then **all the locations** represented by n_\emptyset will be the **target of at most one heap pointer**.

Maintaining Sharing Information (1)

$[y.next := z]^6$

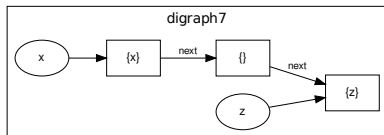


$[y := null]^7$

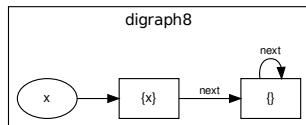


Maintaining Sharing Information (2)

$[y := \text{null}]^7$



$[z := \text{null}]^8$



Sharing Information Invariants (1)

We shall impose **two invariants** to ensure that information in the sharing component is also reflected in the abstract heap, and vice versa.

The first invariant, **Invariant 4**, ensures that information in the sharing component is also reflected in the abstract heap:

Invariant 4: If $n_X \in$ is then either

- a) (n_\emptyset, sel, n_X) is in the abstract heap for some sel , or
 - b) there exist two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap (that is either $sel_1 \neq sel_2$ or $V \neq W$).
- ▶ **Case 4a)** means that there might be several locations represented by n_\emptyset that point to n_X .
 - ▶ **Case 4b)** means that two distinct pointers (with different source or different selectors) point to n_X .

Sharing Information Invariants (2)

The second invariant, **Invariant 5**, ensures that sharing information present in the abstract heap is also reflected in the sharing component:

Invariant 5: Whenever there are two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap and $n_X \neq n_\emptyset$ then $n_X \in is$.

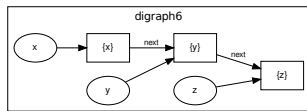
This invariant takes care of the situation where n_X represents a single location being the target of two or more heap pointers.

Note that **Invariant 5** is the ‘inverse’ of **Invariant 4(b)**.

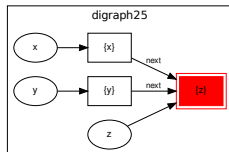
We have no ‘inverse’ of **Invariant 4(a)** - the presence of a pointer from n_\emptyset to n_X gives no information about sharing properties of n_X that are represented in is .

Sharing Component: Example 1

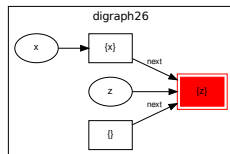
$[y.next := z]^6$



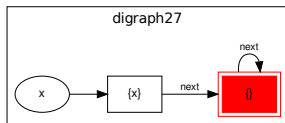
$[x.next := z]^7$



$[y := null]^8$

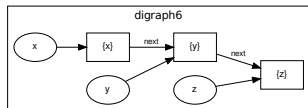


$[z := null]^9$

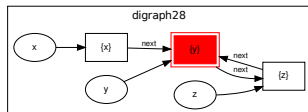


Sharing Component: Example 2

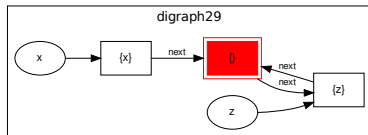
$[y.next := z]^6$



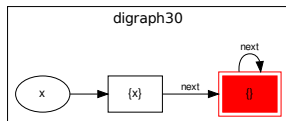
$[z.next := y]^7$



$[y := null]^8$



$[z := null]^9$



Compatible Shape Graphs

A **shape graph** is a triple (S, H, is) :

$$\begin{aligned} S \in AState &= \mathcal{P}(\text{Var}_* \times ALoc) \\ H \in AHeap &= \mathcal{P}(ALoc \times \text{Sel} \times ALoc) \\ is \in IsShared &= \mathcal{P}(ALoc) \end{aligned}$$

where $ALoc = \{n_X \mid X \subseteq \text{Var}_*\}$.

A **shape graph** is a **compatible shape graph** if it fulfills the five invariants, 1-5, presented above.

The **set of compatible shape graphs** is denoted by

$$SG = \{(S, H, is) \mid (S, H, is) \text{ is compatible}\}$$

Complete Lattice of Shape Graphs

The analysis, to be called *Shape*, will operate over *sets of compatible shape graphs*, i.e., elements of $\mathcal{P}(SG)$.

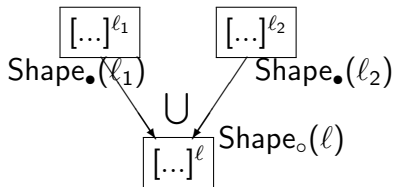
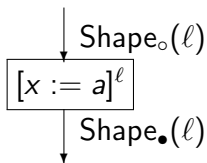
Since $\mathcal{P}(SG)$ is a power set, it is trivially a *complete lattice* with

- ▶ ordering relation \sqsubseteq being \subseteq
- ▶ combination operator \sqcap being \cup (may analysis)

$\mathcal{P}(SG)$ is finite because $SG \subseteq AState \times AHeap \times IsShared$ and all of *AState*, *AHeap*, *IsShared* are finite.

The *analysis* will be *specified* as an *instance* of a *Monotone Framework* with the complete lattice of properties being $\mathcal{P}(SG)$, and as a *forward analysis*.

Analysis



$$\begin{aligned} \text{Shape}_o(l) &= \begin{cases} \iota & : \text{ if } l = \text{init}(S_\star) \\ \bigcup \{ \text{Shape}_\bullet(l') \mid (l', l) \in \text{flow}(S_\star) \} & : \text{ otherwise} \end{cases} \\ \text{Shape}_\bullet(l) &= f_l^{\text{SA}}(\text{Shape}_o(l)) \end{aligned}$$

where $\iota \in \mathcal{P}(SG)$ is the extremal value holding at entry to S_\star .

Transfer Functions

The transfer function $f_\ell^{\text{SA}} : \mathcal{P}(\text{SG}) \rightarrow \mathcal{P}(\text{SG})$ has the form

$$f_\ell^{\text{SA}}(\text{SG}) = \bigcup \{ \phi_\ell^{\text{SA}}((S, H, \text{is})) \mid (S, H, \text{is}) \in \text{SG} \}$$

where ϕ_ℓ^{SA} specifies how a *single shape graph* (in $\text{Shape}_o(\ell)$) may be transformed into a *set of shape graphs* (in $\text{Shape}_\bullet(\ell)$).

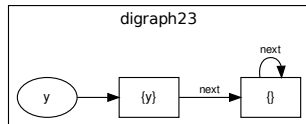
The functions ϕ_ℓ^{SA} for the statements (illustrated by example)

$x := a$	$x := y$	$x := y.\text{sel}$
$x.\text{sel} := a$	$x.\text{sel} := y$	$x.\text{sel} := y.\text{sel}$

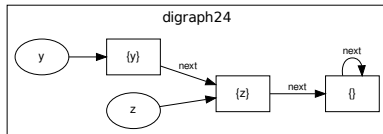
transform a shape graph into a set of different shape graphs.

The transfer functions for other statements and expressions are specified by the identity function.

Example: Materialization



$[z := y.next]^7$

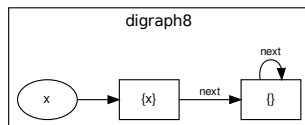


Example: Reverse List

```
[y := null]1;  
while [not isnull(x)]2 do  
  [t := y]3;  
  [y := x]4;  
  [x := x.next]5;  
  [y.next := t]6;  
od  
[t := null]7
```

The program **reverses the list** pointed to by `x` and leaves the result in `y`.

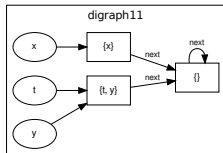
Reverse List: Extremal Value



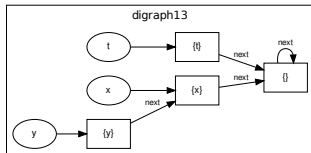
The extremal value ι is a set of graphs. The above graph is an element of this set for our example analysis of the list reversal program.

Shape Graphs in Shape.(ℓ) (1)

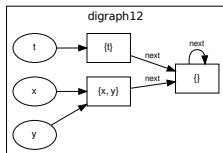
$[t := y]^3$



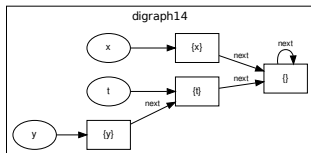
$[x := x.next]^5$



$[y := x]^4$

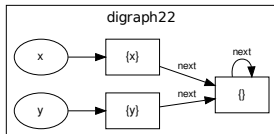


$[y.next := t]^6$

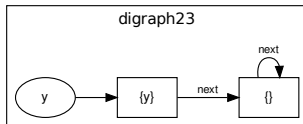


Shape Graphs in Shape.(ℓ) (2)

$[t := \text{null}]^7$



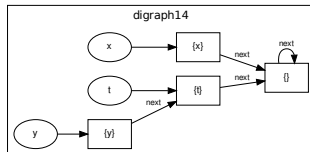
$[x := \text{null}]^7$



Reverse List: Established Properties

For the list reversal program [shape analysis](#) can detect that at the beginning of each iteration of the loop the following properties hold:

- Invariant 1:** Variable x points to an unshared, acyclic, singly linked list.
- Invariant 2:** Variable y points to an unshared, acyclic, singly linked list, and variable t may point to the second element of the y -list (if such an element exists).
- Invariant 3:** The lists pointed to by x and y are disjoint.



Drawbacks and Improvements

An [improved version](#), on which the discussed analysis is based on, can be found in [\[SRW'98\]](#):

- ▶ Operates on a single shape graph instead of sets of shape graphs.
- ▶ Merges sets of compatible shape graphs in one summary shape graph.
- ▶ Uses various mechanisms for extracting parts of individual compatible shape graphs.
- ▶ Avoids the exponential factor in the cost of the discussed analysis.

The [sharing component of the shape graphs](#) is designed to [detect list-like properties](#):

- ▶ It can be replaced by other components detecting other shape properties [\[SRW'02; Compiler Design Handbook, Chap. 5\]](#).

Chapter 14.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11



Chap. 12

Chap. 13



Part IV

1284/18




Further Reading for Chapter 14 (1)

-  Michael G. Burke, Paul R. Carini, Jong-Deok Choi, Michael Hind. *Flow-insensitive Interprocedural Alias Analysis in the Presence of Pointers*. In Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC'94), Springer-V., LNCS 892, 234-250, 1994.
-  Venkatesan T. Chakaravarthy. *New Results on the Computability and Complexity of Points-to-Analysis*. In Conference Record of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 115-125, 2003.




Further Reading for Chapter 14 (2)

-  David Chase, Mark N. Wegmann, F. Kenneth Zadeck. *Analysis of Pointers and Structures*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), ACM SIGPLAN Notices 25:296-310, 1990.
-  Jong-Deok Choi, Michael G. Burke, Paul R. Carini. *Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects*. In Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), 49-61, 1993.

Further Reading for Chapter 14 (3)

-  Maryam Emami, Rakesh Ghiya, Laurie J. Hendren. *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):242-256, 1994.
-  Ben Hardekopf, Calvin Lin. *Semi-sparse Flow-sensitive Pointer Analysis*. In Conference Record of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009), 226-238, 2009.
-  Rakesh Ghiya, Laurie J. Hendren. *Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 1-15, 1996.




Further Reading for Chapter 14 (4)

-  Susan Horwitz. *Precise Flow-insensitive Alias Analysis is NP-hard*. ACM Transactions on Programming Languages and Systems 19(1):1-6, 1997.
-  Bertrand Jeannot, Alexey Logivon, Thomas W. Reps, Mooly Sagiv. *A Relational Approach to Interprocedural Shape Analysis*. In Proceedings of the 11th Static Analysis Symposium (SAS 2004), Springer-V., LNCS 3248, 246-264, 2004.
-  Vini Kanvar, Uday P. Khedker. *Heap Abstractions for Static Analysis*. ACM Computing Surveys 49(2):29, 47 pages, 2017.




Further Reading for Chapter 14 (5)

-  Roman Manevich, Boris Dogadov, Noam Rinetzky. *From Shape Analysis to Termination Analysis in Linear Time*. In Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), Springer-V., LNCS 9779, 426-446, 2016.
-  Roman Manevich, Mooly Sagiv, Ganesan Ramalingam, John Field. *Partially Disjunctive Heap Abstraction*. In Proceedings of the 11th Static Analysis Symposium (SAS 2004), Springer-V., LNCS 3248, 265-279, 2004.
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 10, Alias Analysis)




Further Reading for Chapter 14 (6)

-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 2.6, Shape Analysis)
-  Viktor Pavlu, Markus Schordan, Andreas Krall. *Computation of Alias Sets from Shape Graphs for Comparison of Shape Analysis Precision*. In Proceedings of the 11th International IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2011), 25-34, 2011. [Best Paper Award SCAM 2011]
-  G. Ramalingam. *The Undecidability of Aliasing*. ACM Transactions on Programming Languages and Systems 16(5):1467-1471, 1994.




Further Reading for Chapter 14 (7)

-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-analysis Problems in Languages with Destructive Updating*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 16-31, 1996.
-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-analysis Problems in Languages with Destructive Updating*. ACM Transactions on Programming Languages and Systems 20(1):1-50, 1998.
-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. In Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), 105-118, 1999.

Further Reading for Chapter 14 (8)

-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-valued Logic*. ACM Transactions on Programming Languages and Systems 24(3):217-298, 2002.
-  Marc Shapiro, Susan Horwitz. *Fast and Accurate Flow-insensitive Points-to Analysis*. In Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), 1-14, 1997.
-  Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations & Machine Code Generation*. CRC Press, 2002. (Chapter 5, Shape Analysis and Application)

Further Reading for Chapter 14 (9)

-  Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2nd edition, 2008. (Chapter 5, Shape Analysis and Application)
-  Bjarne Steensgaard. *Points-to Analysis in Almost Linear Time*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 32-41, 1996.
-  Hongseok Yang, Oukse Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn. *Scalable Shape Analysis for Systems Code*. In Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), Springer-V., LNCS 5123, 385-398, 2008.

Further Reading for Chapter 14 (10)



Sean Zhang, Barbara G. Ryder, William Landi. *Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses*. In Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'96), 81-92, 1996.

Chapter 15

Optimizations for Object-Oriented Languages

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1295/18

Optimizations f. Object-Oriented Languages (1)

...related to **method invocation**.

Invoking a **method** in an **object-oriented language** requires **looking up** the **address** of the block of code which implements that method and passing control to it.

Opportunities for optimization

- ▶ **Look-up** may be performed **at compile time**.
- ▶ There is **only one implementation** of the method in the class and in its subclasses.
- ▶ **Language** provides a declaration which forces the **call** to be **non-virtual**.
- ▶ **Compiler** performs **static analysis** which can determine that a **unique implementation** is always called at a particular call-site.

Optimizations f. Object-Oriented Languages (2)

Related optimizations for exploiting these opportunities:

- ▶ Dispatch table compression
- ▶ Devirtualization
- ▶ Inlining
- ▶ Escape analysis for allocating objects on the run-time stack (instead of the heap)

Overview and Orientation

- ▶ **Object Layout and Method Invocation** (cf. Chapter 15.1)
 - ▶ Single inheritance
 - ▶ Multiple inheritance
- ▶ **Devirtualization of Method Calls** (cf. Chapter 15.2)
 - ▶ Class hierarchy analysis
 - ▶ Rapid type analysis
 - ▶ Inlining
- ▶ **Escape Analysis** (cf. Chapter 15.3)
 - ▶ Connection graphs
 - ▶ Intraprocedural
 - ▶ Interprocedural

Chapter 15.1

Object Layout and Method Invocation

Object Layout and Method Invocation

The **memory layout** of an object and how the layout supports **dynamic dispatch** are crucial factors for **performance**.

- ▶ **Single Inheritance**

- ▶ with and without **virtual dispatch table** (i.e., direct calling guarded by a type test)

- ▶ **Multiple Inheritance**

...**various techniques** with different compromises

- ▶ embedding superclasses
- ▶ trampolines
- ▶ table compression

Chapter 15.1.1

Single Inheritance

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

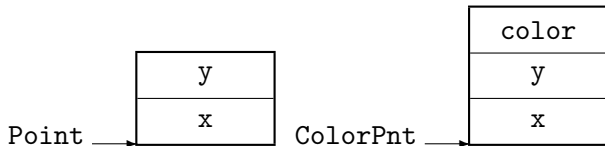
Part IV

1301/18

Single Inheritance Layout

```
class Point {  
    int x, y;  
}
```

```
class ColorPnt extends Point {  
    int color;  
}
```

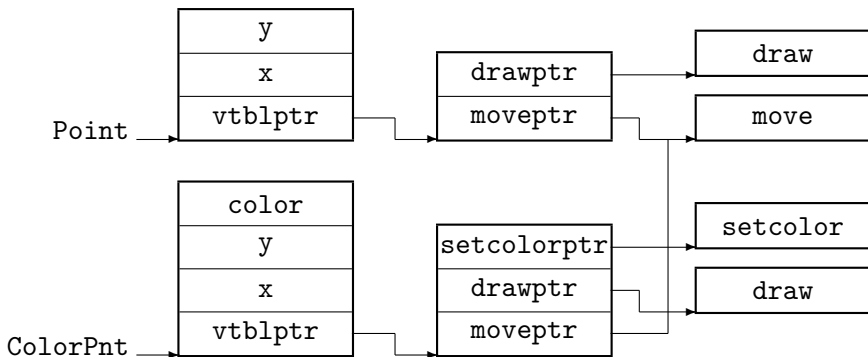


- ▶ **Memory layout** of an **object of a superclass** is a prefix of the memory layout of an object of the subclass.
- ▶ **Instance variables access** requires just **one load** or **store instruction**.

Single Inheritance Layout with vtbl

```
class Point {  
    int x, y;  
    void move(int x, int y) {...}  
    void draw() {...}  
}
```

```
class ColorPnt extends Point {  
    int color;  
    void draw() {...}  
    void setcolor(int c) {...}  
}
```



Invocation of Virtual Methods with vtbl

- ▶ **Dynamic dispatching** using a **virtual method table (vtbl)** has the advantage of being fast and executing in constant time.
- ▶ It is possible to **add new methods** and to **override methods**.
- ▶ Each method is assigned a fixed offset in the **virtual method table (vtbl)**.
- ▶ **Method invocation** is just **three machine code instructions**:

```
LDQ vtblptr, (obj)           ; load vtbl pointer
LDQ mptr, method(vtblptr)   ; load method pointer
JSR (mptr)                   ; call method
```
- ▶ **One extra word of memory** is needed in each object for the pointer to the **virtual method table (vtbl)**.

Dispatch Without Virtual Method Tables

Despite the use of **branch target caches**, **indirect branches** are **expensive** on modern architectures.

The **pointer to the class information** and **virtual method table** is replaced by a type identifier:

- ▶ A **type identifier** is an integer **representing** the **type** of the **object**.
- ▶ It is used in a **dispatch function** which searches for the **type** of the **receiver**.
- ▶ Example: **SmallEiffel** (binary search).
- ▶ **Dispatch functions** are shared between calls with the same statically determined set of concrete types.
- ▶ In the **dispatch function** a **direct branch** to the **dispatched method** is used (or it is **inlined**).

Example

Let type identifiers $T_A, T_B, T_C,$ and T_D be sorted by increasing number. The dispatch code for calling $x.f$ is:

```
if  $id_x \leq T_B$  then  
    if  $id_x \leq T_A$  then  $f_A(x)$   
    else  $f_B(x)$   
else if  $id_x \leq T_C$  then  $f_C(x)$   
else  $f_D(x)$ 
```

Comparison with dispatching using a virtual method table:

- ▶ Empirical study showed that for a method invocation with three concrete types, dispatching with binary search is between 10% and 48% faster.
- ▶ For a megamorphic call with 50 concrete types, the performance is about the same.

Chapter 15.1.2

Multiple Inheritance

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1307/18

Multiple Inheritance

...extending the superclasses as in single inheritance does not work anymore.

Instead

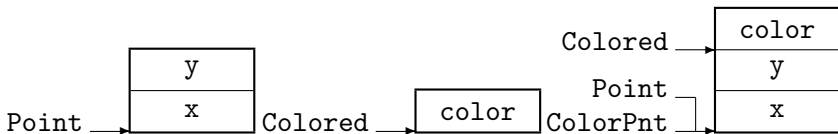
- ▶ Fields of superclass are embedded as contiguous block.
- ▶ Embedding allows fast access to instance variables exactly as in single inheritance.
- ▶ Garbage collection becomes more complex because pointers also point into the middle of objects.

Object Memory Layout (without vtbl)

```
class Point {  
    int x, y;  
}
```

```
class Colored {  
    int color;  
}
```

```
class ColorPnt extends Point, Colored {  
}
```



Dynamic Dispatching for Embedding

- ▶ Allows fast access to instance variables exactly as with single inheritance.
- ▶ For every superclass
 - ▶ virtual method tables (*vbt*) have to be created.
 - ▶ multiple *vbt* pointers are included in the object.
- ▶ The object pointer is adjusted to the embedded object whenever explicit or implicit pointer casting occurs (assignments, type casts, parameter and result passing).

Multiple Inheritance with vtbl (1)

```
class Point {  
    int x, y;  
    void move(int x, int y) {...}  
    void draw() {...}  
}
```

```
class Colored {  
    int color;  
    void setcolor(int c) {...}  
}
```

```
class ColorPnt extends Point, Colored {  
    void draw() {...}  
}
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

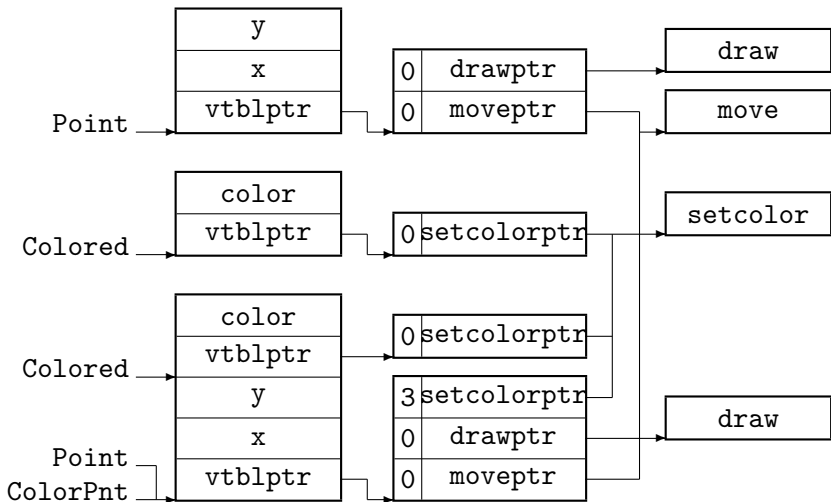
Chap. 12

Chap. 13

Part IV

1311/18

Multiple Inheritance with vtbl (2)



Pointer Adjustment and Adjustment Offset

Pointer adjustment has to be suppressed for casts of null pointers:

```
Colored col; ColorPnt cp; ...;  
col = cp; // if (cp!=null)col=(Colored)((int*)cp+3)
```

Problem w/ implicit casts from actual receiver to formal receiver:

- ▶ Caller has no type info of formal receiver in the callee.
- ▶ Callee has no type info of actual receiver of the caller.
- ▶ Therefore this type info has to be stored as an adjustment offset in the vtbl.

Method Invocation with vtbl

Method invocation now takes 4 to 5 machine instructions (depending on the architecture).

```
LD  vtblptr,(obj)          ; load vtbl pointer
LD  mptr,method_ptr(vtblptr) ; load method pointer
LD  off,method_off(vtblptr) ; load adjustment offset
ADD obj,off,obj           ; adjust receiver
JSR (mptr)                ; call method
```

This overhead in table space and program code is even necessary when multiple inheritance is not used (in the code).

Furthermore, adjustments to the remaining parameters and the result are not possible.

Trampoline

To **eliminate** much of the **overhead** a small piece of code, called **trampolin** is inserted that **performs** the **pointer adjustments** and the **jumps to the original code**.

The **advantages** are

- ▶ **smaller table size** (no storing of an offset)
- ▶ **fast method invocation** when multiple inheritance is not used
 - ▶ the **same dispatch code** as in single inheritance

The **method pointer** `setcolorptr` in the **virtual method table** of `Colorpoint` would (instead) point to code which adds 3 to the receiver before jumping to the code of **method** `setcolor`:

```
ADD obj,3,obj      ; adjust receiver
BR setcolor        ; call method
```

Lookup at Compile-Time

Invoking a method requires looking up the address of the method and passing control to it.

In some cases, the lookup may be performed at compile-time:

- ▶ There is only one implementation of the method in the class and its subclasses.
- ▶ The language provides a declaration that forces the call to be non-virtual.
- ▶ The compiler has performed static analysis that can determine that a unique implementation is *always* called at a particular call site.

In other cases, a runtime lookup is required.

Dispatch Table

In principle the **lookup** can be implemented as indexing a **two-dimensional table**. A number is given to

- ▶ each method in the program
- ▶ each class in the program

The **method call**

```
result = obj.m(a1,a2);
```

can be **implemented by** the following three actions:

1. Fetch a **pointer** to the appropriate row of the **dispatch table** from the object `obj`.
2. Index the **dispatch table row** with the method number.
3. **Transfer control** to the address obtained.

Dispatch Table Compression (1)

▶ Virtual Tables

- ▶ Effective method for statically typed languages.
- ▶ Methods can be numbered compactly for each class hierarchy to leave no unused entries in each vtbl.

▶ Row Displacement Compression

- ▶ **Idea:** combine all rows into a single very large vector.
- ▶ It is possible to have rows overlapping as long as an entry in one row corresponds to empty entries in the other rows.
- ▶ **Greedy algorithm:** place first row; for all subsequent rows: place on top and shift right if conflicts exist.
- ▶ **Unchanged:** implementation of method invocation.
- ▶ **Penalty:** verify class of current object at the beginning of any method that can be accessed via more than one row.

Dispatch Table Compression (2)

▶ Selector Coloring Compression

- ▶ **Graph coloring:** two rows can be merged if no column contains different method addresses for the two classes.
- ▶ **Graph:** one node per class; an edge connects two nodes if the corresponding classes provide different implementations for the same method name.
- ▶ **Coloring:** each color corresponds to the index for a row in the compressed table.
- ▶ Each object contains a reference to a possibly shared row.
- ▶ **Unchanged:** implementation of method invocation code.
- ▶ **Penalty:** if classes C1 and C2 share the same row and C1 implements method m whereas C2 does not, then the code for m should begin with a check that control was reached via dispatching on an object of type C1.

Chapter 15.2

Devirtualization of Method Invocations

Devirtualization

Devirtualization is a technique to reduce the overhead of virtual method invocation.

The aim of this technique is to statically determine which methods can be invoked by virtual method calls.

- ▶ If exactly one method is resolved for a method call, the method can be inlined or the virtual method call can be replaced by a static method call.

The analyses necessary for devirtualization also improve the accuracy of the call graph and the accuracy of subsequent interprocedural analyses.

Chapter 15.2.1

Class Hierarchy Analysis

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1322/18

Class Hierarchy Analysis

The **simplest devirtualization technique** is **class hierarchy analysis (CHA)**, which determines the **class hierarchy used** in a program.

The information about all referenced classes is used to create a **conservative approximation of the class hierarchy**.

- ▶ The transitive closure of all classes referenced by the class containing the main method is computed.
- ▶ The declared types of the receiver of a virtual method call are used for determining all possible receivers.

Example: Class Hierarchy Analysis

```
class A extends Object {
    void m1() {...}
    void m2() {...}
}
class B extends A {
    void m1() {...}
}
class C extends A {
    void m1() {...}
    public static void main(...) {
        A a = new A();
        B b = new B();
        ...
        a.m1(); b.m1(); b.m2();
    }
}
```

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

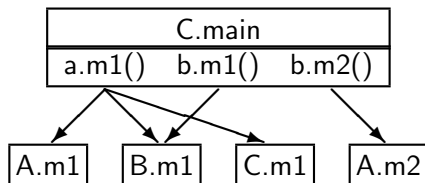
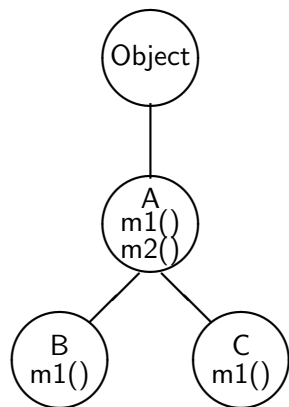
Chap. 12

Chap. 13

Part IV

1324/18

Example: Class Hierarchy and Call Graph



The CHA Algorithm

main // the main method in a program
x() // call of static method *x*
type(x) // the declared type of the expression *x*
x.y() // call of virtual method *y* in expression *x*
subtype(x) // *x* and all classes which are a subtype of class *x*
method(x, y) // the method *y* which is defined for class *x*

callgraph := *main*

hierarchy := {}

for each *m* ∈ *callgraph* **do**

for each *m_{stat}*(*e*) occurring in *m* **do**

if *m_{stat}* ∉ *callgraph* **then**

add *m_{stat}* to *callgraph*

for each *e.m_{vir}*(*e*) occurring in *m* **do**

for each *c* ∈ *subtype(type(e))* **do**

m_{def} := *method(c, m_{vir})*

if *m_{def}* ∉ *callgraph* **then**

add *m_{def}* to *callgraph*

add *c* to *hierarchy*

Chapter 15.2.2

Rapid Type Analysis

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1327/18

Rapid Type Analysis (1)

Rapid type analysis (RTA) uses the fact that a method m of a class c can be invoked only if an object of type c is created during the execution of the program.

- ▶ **RTA** refines the class hierarchy (compared to **CHA**) by only including classes for which objects can be created at **runtime**.

Based on this idea

- ▶ pessimistic
- ▶ optimistic

algorithms are possible.

Rapid Type Analysis (2)

1. The pessimistic algorithm

...includes all classes in the class hierarchy for which instantiations occur in methods of the call graph from CHA.

2. The optimistic algorithm

- ▶ Initially assumes that no methods besides *main* are called and that no objects are instantiated.
- ▶ It traverses the call graph initially ignoring virtual calls (marking them in a mapping as potential calls only) following static calls only.
- ▶ When an instantiation of an object is found during analysis, all virtual methods of the corresponding objects that were left out previously are then traversed as well.
- ▶ The live part of the call graph and the set of instantiated classes grow interleaved as the algorithm proceeds.

Chapter 15.2.3

Inlining

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1330/18

Using Devirtualization Information

Inlining is an important usage of devirtualization information.

If a virtual method call can be devirtualized

- ▶ it might completely be replaced by inlining the call (supposed it is not recursive).

Chapter 15.3

Escape Analysis

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1332/18

Escape Analysis

The goal of **escape analysis** is to determine which objects have **lifetimes** which **do not stretch outside the lifetime of their immediately enclosing scopes**.

- ▶ The **storage** for such objects can be safely allocated as part of the **current stack frame** – that is, their storage can be allocated on the **run-time stack**.
- ▶ **At method return**, **deallocation** of the **memory space** used by non-escaping objects is automatic. **No garbage collection** is required.
- ▶ The transformation also **improves** the **data locality** of the program and, depending on the computer's cache, can significantly **reduce execution time**. Objects not escaping a thread can be allocated in the processor where that thread is scheduled.

Using Escape Information

Objects whose lifetimes are confined to within a single scope cannot be shared between two threads.

- ▶ Synchronization actions for these objects can be eliminated.

Escape Analysis by Abstract Interpretation

A [prototype implementation of escape analysis](#) was included in the [IBM High Performance Compiler for Java](#).

The [approach of Choi et al. \(OOPSLA'99\)](#) attempts to determine whether the object

- ▶ [escapes from a method](#) (i.e., from the scope where it is allocated).
- ▶ [escapes from the thread](#) that created it
 - ▶ the object can escape a method but does not escape from the thread.

Note: The converse is not possible (if it does not escape the method then it cannot escape the thread).

Essence of Choi et al.'s Approach

- ▶ Introducing of a **simple program abstraction** called **connection graph**:

Intuitively, a **connection graph** captures the **connectivity** relationship between heap allocated objects and object references.

- ▶ Demonstrating that **escape analysis boils down** to a **reachability problem within connections graphs**:

If an object is reachable from an object that might escape, it might escape as well.

Experimental Results Reported by Choi et al.

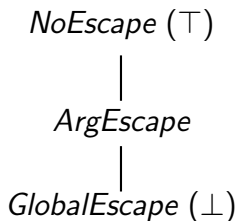
...based on 10 benchmark programs:

- ▶ Percentage of objects that may be allocated on the stack:
Up to 70 + %, with a median of 19%.
- ▶ Percentage of all lock operations eliminated:
From 11% to 92%, with a median of 51%.
- ▶ Overall execution time reduction:
From 2% to 23%, with a median of 7%.

These results make [escape analysis](#) and the [optimizations](#) based thereon [worthwhile](#).

Escape States

The analysis uses a [simple lattice](#) to represent different [escape states](#):



State	Escapes the method	Escapes the thread
NoEscape	no	no
ArgEscape	may (via args)	no
GlobalEscape	may	may

Using Escape Information

All objects which are marked

- ▶ **NoEscape**: are **stack-allocatable** in the method where they are created.
- ▶ **NoEscape** or **ArgEscape**: are local to the thread in which they are created; hence **synchronization statements** in accessing these objects **can be eliminated**.

Chapter 15.3.1

Connection Graphs

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1340/18

Connection Graphs

We are interested only in

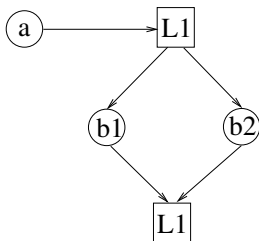
- ▶ following the object O from its point of allocation.
- ▶ knowing which variables reference O .
- ▶ and which other objects are referenced by O fields.

We 'abstract out' the referencing information, using a graph structure where

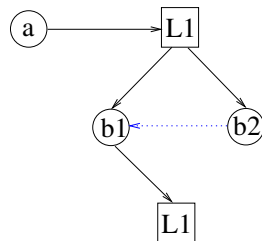
- ▶ a circle node represents a variable.
- ▶ a square node represents objects in the heap.
- ▶ an edge from circle to square represents a reference.
- ▶ an edge from square to circle represents ownership of fields.

Example: Connection Graphs

```
A a = new A(); // line L1
a.b1 = new B(); // line L2
a.b2 = a.b1; // line L3
```



Simple Version



Using Deferred Edges

An edge drawn as a dotted arrow is called a **deferred edge** and shows the effect of an assignment from one variable to another (example: created by the assignment in line 3) \rightsquigarrow **improves efficiency of the approach.**

Chapter 15.3.2

Intraprocedural Setting

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

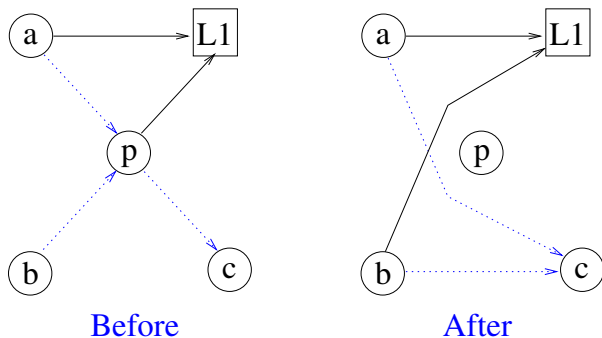
Part IV

1343/18

Intraprocedural Abstract Interpretation

Actions for assignments involve an update of the connection graph.

- ▶ An assignment to a variable p kills any value the variable previously had. The kill function is called *byPass*(p):



Analyzing Statements (1)

$p = \text{new } C();$ // line L The operation $\text{byPass}(p)$ is applied.

An object node labeled L is added to the graph – and nodes for the fields of C that have nonintrinsic types are also created and connected by edges pointing from the object node.

$p = q;$ The operation $\text{byPass}(p)$ is applied. A new deferred edge from p to q is created.

$p.f = q;$ The operation byPass is *not* applied for f (no strong update!). If p does not point to any node in the graph a new (phantom) node is created. Then, for each object node connected to p by an edge, an assignment to the field f of that object is performed.

Analyzing Statements (2)

$p = q.f$; If q does not point at any object node then a phantom node is created and an edge from q to the new node is added. Then *byPass*(p) is applied and deferred edges are added from p to all the f nodes that q is connected to by field edges.

For each statement one graph represents the state of the program at the statement.

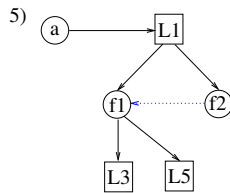
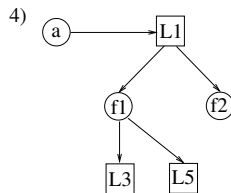
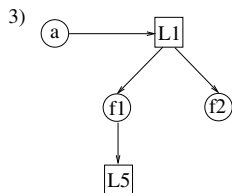
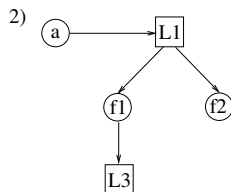
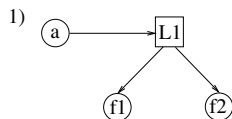
At a point where two or more control paths converge, the connection graphs from each predecessor statements are merged.

Example: Connection Graphs (1)

Suppose that the code inside some method is as follows. The declarations of classes `A`, `B1` and `B2` are omitted.

```
A a = new A();      // line L1
if (i > 0)
    a.f1 = new B1(); // line L3
else
    a.f1 = new B2(); // line L5
a.f2 = a.f1;       // line L6
```

Example: Connection Graphs (2)



G_1 : out: A a = new A(); // line L1

G_2 : out: a.f1 = new B1(); // line L3

G_3 : out: a.f1 = new B2(); // line L5

G_4 : out: $G_2 \cup G_3$

G_5 : out: a.f2 = a.f1; // line L6

Chapter 15.3.3

Interprocedural Setting

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1349/18

Interprocedural Abstract Interpretation (1)

Analyzing methods:

- ▶ It is necessary to analyze each method in the **reverse order** implied by the **call graph**.
- ▶ If method **A** may call methods **B** and **C**, then **B** and **C** should be analyzed before **A**.
- ▶ **Recursive edges** in the call graph are ignored when determining the order.
- ▶ **Java** has **virtual method calls** – at a method call site where it is not known which method implementation is being invoked, the analysis must assume that all of the possible implementations are called, combining the effects from all the possibilities.
- ▶ The interprocedural analysis iterates over all the methods in the call graph until the results converge (fixed point).

Interprocedural Abstract Interpretation (2)

- ▶ A call to a method M is equivalent to copying the actual parameters (i.e., the arguments being passed in the method call) to the formal parameters, then executing the body of M , and finally copying any value returned by M as its result back to the caller.
- ▶ If M has already been analyzed intraprocedurally following the approach described above, the effect of M can be summarized with a connection graph. That summary information eliminates the need to re-analyze M for each call site in the program.

Analysis Results (1)

After the operation *byPass* has been used to eliminate all deferred edges, the connection graph can be partitioned into three subgraphs:

Global escape nodes: All nodes reachable from a node whose associated state is *GlobalEscape* are themselves considered to be global escape nodes (Subgraph 1)

- ▶ the nodes initially marked as *GlobalEscape* are the static fields of any classes and instances of any class that implements the *Runnable* interface.

Argument escape nodes: All nodes reachable from a node whose associated state is *ArgEscape*, but are not reachable from a *Global Escape* node. (Subgraph 2)

- ▶ the nodes initially marked as *ArgEscape* are the argument nodes a_1, \dots, a_n .

Analysis Results (2)

No escape nodes: All other nodes have *NoEscape* status.
(Subgraph 3).

The third subgraph represents the summary information for the method because it shows which objects can be reached via the arguments passed to the method.

All objects created within a method *M* and that have the *NoEscape* status after the three subgraphs have been determined can be **safely allocated on the stack**.

Chapter 15.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11




Chap. 12

Chap. 13



Part IV

1354/18

Further Reading for Chapter 15 (1)

-  John-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. *Escape Analysis for Java*. In Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99), 1-19, 1999.
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)
-  Hemant D. Pande, Barbara Ryder. *Data-flow-based Virtual Function Resolution*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 238-254, 1996.

Further Reading for Chapter 15 (2)

-  Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 1st edition, CRC Press, 2002. (Chapter 6, Optimizations for Object-Oriented Languages)
-  Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 2nd edition, CRC Press, 2008. (Chapter 13, Optimizations for Object-Oriented Languages)

Part V

Conclusions and Perspectives

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1357/18

Chapter 16

Conclusions, Emerging and Future Trends

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1358/18

Chapter 16.1

Reconsidering Optimization

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

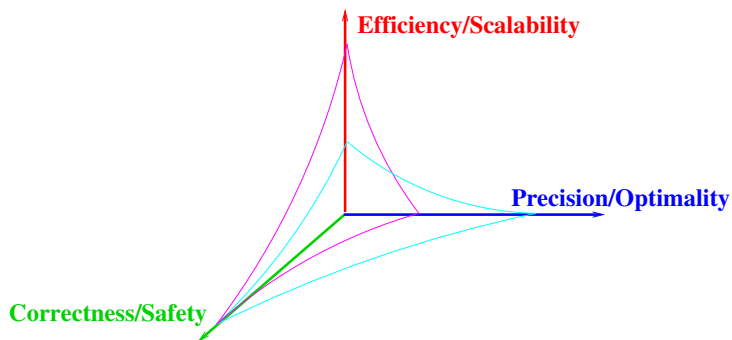
Part IV

1359/18

Program Analysis and Optimization (1)

...takes place in the area of conflict between:

- ▶ Correctness/Safety
- ▶ Precision/Optimality
- ▶ Efficiency/Scalability



Program Analysis and Optimization (2)

In principle

- ▶ **Correctness/safety**, **precision/optimality**, and **efficiency/scalability** can be traded for each other.

For example

- ▶ **Iterative Compilation**: Analytically, experimentally
...trades **efficiency/scalability** for **precision/optimality**.
- ▶ **Adaptive Compilation**: Experimentally
...trades **efficiency/scalability** for **precision/optimality**.
- ▶ **Aggressive Optimization**
...trades **safety/correctness** and/or **efficiency/scalability**
for **impact** (rather than **precision/optimality**).

Program Analysis and Optimization (3)

Different fields also impose different **performance demands**:

- ▶ **Compilation** – trading precision/optimality for efficiency/scalability
 - ▶ Interactive, user: **high**
 - ▶ Batch, embedded systems compilation: **moderate**
 - ▶ Dynamic: **extremely high**
- ▶ **Verification** – trading efficiency/scalability for precision/optimality
 - ▶ moderate to low notwithstanding as fast as possible
- ▶ **On-line monitoring/verification** – trading precision/optimality for efficiency/scalability
 - ▶ real-time empowered (autonomous systems,...)

The characteristics and demands of an application scenario have a tremendous impact on the kind of analyses and transformations/optimizations which are considered reasonable.

Optimization worth the Effort?

...which options do we have if our program is **too slow**?

A radical view:

- ▶ **Option 1: Buying new hardware!**

Moore's Law. Hardware performance gains double the computing power every 18 months.

- ▶ **Option 2: Buying a new compiler!**

Proebsting's Law. Compiler optimizations gains double the computing power every 18 years.

Note: Proebsting's Law above is a corollary of his 'finding/observation:'

- ▶ *"Compiler optimizations have yielded annual performance gains an order of magnitude worse than hardware performance gains."*

Optimization of Little to No Relevance?

No, by contrast.

- ▶ Program analysis and optimization are more important than ever these days, and will so continue in the years to come.

What evidence do we have?

Most importantly

- ▶ Moore's Law is vanishing: 'The end' of Moore's Law due to physical limitations is foreseeable.
 - ▶ Waiting for the next processor generation with higher clock rate is no longer an option:
There is no free lunch any longer!
- ▶ The improvement by compiler optimizations is always on top of any improvement by hardware advancements.

Optimization of the Highest Relevance

In fact, in response to the foreseeable 'end' of Moore's Law

- ▶ All major chip vendors switched their focus from processors with higher and higher clock rates to processors with many and multi-cores.

Again

- ▶ **There is no free lunch any longer!**

In the words of a speaker at the CGO 2007 conference:

We asked for more computing power.

We got more processor cores.

A Speaker at CGO 2007

Hence

- ▶ New parallelization and optimization techniques are required!

Drivers of the Relevance of Optimization (1)

New advances in hardware and software demand strong compiler and optimizer support:

- ▶ **Parallelism**
 - ▶ **Hardware/processors:** Many/multi-core processors, CPUs, GPUs, GPGPUs, FPGAs, and other accelerators, heterogeneous hardware,...
 - ▶ **Software:** Parallel languages, parallelization of sequential programs (legacy software),...
 - ▶ **New computing paradigms:** Cloud computing, software-as-a-service,...
- ▶ **Embedded and cyber-physical systems**
 - ▶ **Mobile systems:** Laptops, tablets, smartphones,...
 - ▶ **Autonomous mobile systems, (safety-critical) real-time systems:** Robots in outer space, in co-working spaces with humans, fully autonomous cars, trains, subways, airplanes, ships,...) impose rigorous demands for safety and security, performance, power consumption, etc.

Drivers of the Relevance of Optimization (2)

Grand Challenges of Informatics pose new and strong demands on compilers and optimization, e.g.:

- ▶ The Verifying Compiler, Sir Tony Hoare.
 - ▶ Related Endeavours
 - ▶ Compiler verification: ProCoS, Verifix, CompCert,...
 - ▶ Translation/optimization validation: C3PRO, TVOC, CVT, VOC CovaC,...
 - ▶ ...
- ▶ Verlässlichkeit von Software, Gesellschaft für Informatik e.V. (GI), Fachbereich Softwaretechnik.

...contributions and advances in compiler construction, program analysis and optimization are crucial for successfully mastering these and other (grand) challenges.

Drivers of the Relevance of Optimization (3)

...research on **optimization** impacts other research fields and vice versa:

- ▶ **Program analysis** and **optimization**
- ▶ **Computer aided verification**
- ▶ **Software engineering**
 - ▶ Program understanding, program debugging, program re-engineering, program re-factoring, program (re-) specification,...
 - ▶ Model-driven code generation, model-driven code transformation,...
- ▶ **Safety and security/privacy analysis**
 - ▶ E.g., individual code generation for each compiler run to enhance security (Michael Franz, Stefan Brunthaler et al.)
- ▶ **Language and compiler design**
- ▶ **Hardware/processor design**
- ▶ ...

...mutually benefit of and challenge each other.

Chapter 16.2

Summary, Looking Ahead

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1369/18

Summing up, Looking ahead (1)

All this shows:

New topics in research on optimization pop up:

- ▶ Portable performance
 - ▶ Write once, run everywhere with the highest performance (CPUs, GPUs, GPGPUs, FPGAs, Multi-/many-core architectures, heterogeneous architectures,...)
- ▶ Power consumption
 - ▶ Mobile devices: laptops, tablets, smartphones (Pokémon Go), robots,...
 - ▶ Outer-space objects: spacecrafts, satellites, robots (ESA Rosetta, ESA Philae, MER-A Spirit, MER-B Opportunity,...),...
- ▶ Green IT
 - ▶ Power consumption (in the large)
- ▶ ...

Summing up, Looking ahead (2)

Established topics in research on optimization gain new momentum and experience a renaissance:

- ▶ Parallelization

- ▶ Parallelism for the masses (parallelism is no longer a niche for the expert).

- ▶ Performance

- ▶ Moore's law is nearing its end due to physical limits.

- ▶ Resource Analysis

- ▶ Embedded and cyber-physical systems are ubiquitous.
 - ▶ Size, Power, Performance (WCET, ACET)

- ▶ Selection and Phase-ordering Problem of Optimizations

- ▶ Machine learning approaches seem promising (cf. recent survey article by Ashouri et al. 2019)

- ▶ ...

Summing up, Looking ahead (3)

For all these reasons, it is fair to say:

Compiler construction, program analysis, and optimization is

- ▶ a vibrant, theoretically and practically relevant field of research in informatics and will remain so in the years to come.
- ▶ among the most influential fields for the further progress and advancement of informatics.

Summing up, Looking ahead (4)

Key issues

- ▶ Keeping pace with **advances** in **software** and **hardware design**.
- ▶ Impacting **language** and **hardware design**.
- ▶ Complementing the well-established and powerful **theory of program analysis** with an equally powerful **theory of program transformations**.

Summing up, Looking ahead (5)

Overall

- ▶ The future of the field of compiler construction, program analysis, and optimization is bright!

In particular

- ▶ Compiler construction, program analysis, and optimization are an unexhaustable source of challenging theoretically and practically relevant topics for PhD, master, and bachelor theses.

Chapter 16.3

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1375/18

Further Reading for Chapter 16 (1)

On Moore's and Proebsting's Laws





 IEEE Spectrum. *Special Report on 50 Years of Moore's Law*. IEEE Spectrum, April 2015.

<http://spectrum.ieee.org/static/special-report-50-years-of-moores-law>





 Robert Colwell. *The Chip Design Game at the End of Moore's Law*. In Proceedings of the IEEE/ACM Symposium on High-Performance Chips (Hot Chips 2013), 2013.

http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.15-keynote1-Chipdesign-epub/HC25.26.190-Keynote1-ChipDesignGame-Colwell-DARPA.pdf




Further Reading for Chapter 16 (2)

-  Peter J. Denning, Ted G. Lewis. *Exponential Laws of Computing Growth*. Communications of the ACM 60(1):54-65, 2017.
-  Don Monroe. *Electronics are Leaving the Plane*. Communications of the ACM 61(8):17-18, 2018.
-  Don Monroe. *And Then, There Were Three — How long can the silicon foundry sector continue to adapt, as physical limits make further shrinkage virtually impossible?* Communications of the ACM 62(6):16-18, 2019.
-  Gordon E. Moore. *Cramming More Components onto Integrated Circuits*. Electronics 38(8), 114-117, 1965. http://web.eng.fiu.edu/npala/eee6397ex/gordon_moore_1965_article.pdf

Further Reading for Chapter 16 (3)

-  Gordon E. Moore. *Progress in Digital Integrated Electronics*. International Electron Devices Meeting, IEEE, IEDM Tech. Digest, 11–13, 1975.
http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf
-  John M. Shalf, Robert Leland. *Computing beyond Moore's Law*. IEEE Computer 48(12):14-23, 2015.
-  Tom Simonite. *Moore's Law Is Dead. Now What?* MIT Technology Review, 2016.
<https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/>
-  Arnold Thackray, David Brock, Rachel Jones. *Moore's Law: The Life of Gordon Moore. Silicon Valley's Quiet Revolutionary*. Basic Books, New York, 2015.

Further Reading for Chapter 16 (4)


-  M. Mitchell Waldrop. *More than Moore*.
Nature 530(7589):144-147, 2016.
http://www.nature.com/polopoly_fs/1.19338!/menu/main/topColumns/topLeftColumn/pdf/530144a.pdf
-  Todd A. Proebsting. *Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years*.
<http://proebsting.cs.arizona.edu/law.html>
-  International Roadmap for Devices and Systems 2017 Edition, IEEE. <https://irds.ieee.org/roadmap-2017>

Further Reading for Chapter 16 (5)


Iterative Compilation

-  Grigori Fursin, Michael F.P. O'Boyle, Peter M.W. Knijnenburg. *Evaluating Iterative Compilation*. In Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing (LCPC 2002), Revised Papers, Springer-V., LNCS 2481, 362-376, 2005.
-  Toru Kisuki, Peter M.W. Knijnenburg, Michael F.P. O'Boyle, François Bodin, Harry A.G. Wijshoff. *A Feasibility Study in Iterative Compilation*. In Proceedings of the 2nd International Symposium on High Performance Computing (ISHPC'99), Springer-V., LNCS 1615, 121-132, 1999.



Further Reading for Chapter 16 (6)

-  Peter M.W. Knijnenburg, Toru Kisuki, Michael F.P. O'Boyle. *Iterative Compilation*. In Proceedings of the 1st Workshop on Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS 2001), Springer-V., LNCS 2268, 171-187, 2002.

Adaptive Compilation




-  L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *Finding Effective Compilation Sequences*. In Proceedings of the 2004 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), ACM SIGPLAN Notices 39(7):231-239, 2004.

Further Reading for Chapter 16 (7)

-  Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *ACME: Adaptive Compilation Made Efficient*. In Proceedings of the 2005 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005), 69-77, 2005.
-  Keith D. Cooper, Devika Subramanian, Linda Torczon. *Adaptive Optimizing Compilers for the 21st Century*. The Journal of Supercomputing 23(1):7-22, 2002.



Further Reading for Chapter 16 (8)

Machine-learning enhanced Optimization


-  Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, Cristina Silvano. *A Survey on Compiler Auto-tuning using Machine Learning*. *Computing Surveys* 51(5):96.1-96.42, 2019.
-  Chris Cummins, Pavlos Petoumenos, Zheng Wang, Hugh Leather. *End-to-end Deep Learning of Optimization Heuristics*. In *Proceedings of the 26th ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT 2017)*, 219-232, 2017.
-  Sameer Kulkarni, John Cavazos. *Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning*. *Proceedings of the ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2012)*, 147-162, 2012.

Further Reading for Chapter 16 (9)



Aggressive Optimization

-  Li-Ling Chen, Youfeng Wu. *Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation*. In Proceedings of the 2003 IEEE International Conference on Parallel Processing (ICPP 2003), 607-614, 2003.
-  Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Leaf Petersen. *A Verifiable SSA Program Representation for Aggressive Compiler Optimization*. In Conference Record of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), 397-408, 2006.

Further Reading for Chapter 16 (10)

-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 1.5, Placement of Optimizations in Aggressive Optimizing Compilers)

Grand Challenges

-  Charles A.R. Hoare. *The Verifying Compiler: A Grand Challenge for Computing Research*. Journal of the ACM 50(1):63-69, 2003.
-  Gesellschaft für Informatik e.V. (GI), Fachbereich Softwaretechnik. *Verlässlichkeit von Software*, 2014.
<https://www.gi.de/themen/grand-challenges/verlaesslichkeit-von-software.html>

Further Reading for Chapter 16 (11)




Compiler Verification

-  Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, Jean Souyris. *Towards Formally Verified Optimizing Compilation in Flight Control Software*. In Proceedings of the Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011), 59-68, 2011.
-  Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard von Karger, Yassine Lakhnech, Hans Langmaack, Markus Müller-Olm. *Provably Correct Compiler Development and Implementation*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-V., LNCS 641, 141-155, 1992.




Further Reading for Chapter 16 (12)

-  Sabine Glesner, Gerhard Goos, Wolf Zimmermann. *Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers)*. *it - Information Technology* 46(5):265-276, 2004.
-  Gerhard Goos, Wolf Zimmermann. *Verification of Compilers*. *Correct System Design: Recent Insight and Advances*. Springer-V., LNCS 1710, 201-230, 1999.
-  Andreas Krall. *Correct Compilers for Correct Processors*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: <http://old.hipeac.net/system/files/hipeac14.pdf>

Further Reading for Chapter 16 (13)

-  Hans Langmaack. *Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit*. *it+ti - Informationstechnik und Technische Informatik* 39(3):41-47, 1997.
-  Hans Langmaack: *The ProCoS Approach to Correct Systems*. *Real-Time Systems* 13(3):253-275, 1997.
-  Xavier Leroy. *Formal Verification of a Realistic Compiler*. *Communications of the ACM* 52(7):107-115, 2009.

Further Reading for Chapter 16 (14)

-  Xavier Leroy. *Formally Verifying a Compiler: Why? How? How far?* Invited Talk, In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011), xxxi, 2011.
-  Xavier Leroy. *Compiler Verification for Fun and Profit.* Invited Talk, In Proceedings of the 2014 IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD 2014), 9, 2014.
-  John McCarthy, James Painter. *Correctness of a Compiler for Arithmetical Expressions.* Mathematical Aspects of Computer Science, ser. Proceedings of Symposia in Applied Mathematics, Vol. 19, American Mathematical Society, 33-41, 1967.

Further Reading for Chapter 16 (15)





Robin Milner, R. Weyrauch. *Proving Compiler Correctness in a Mechanized Logic*. In Proceedings of the 7th Annual Machine Intelligence Workshop, ser. Machine Intelligence, B. Meltzer and D. Michie, Eds., Vol. 7., Edinburgh University Press, 51-72, 1972.





Wolf Zimmermann. *On the Correctness of Transformations in Compiler Back-Ends*. In Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004), Springer-V, LNCS 4313, 74-95, 2004.

Further Reading for Chapter 16 (16)




Translation Validation

-  Clark W. Barret, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, Lenore Zuck. *TVOC: A Translation Validator for Optimizing Compilers*. In Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005), Springer-V., LNCS 3576, 291-295, 2005.
-  Aditya Kanade, Amitabha Sanyal, Uday Khedker. *A PVS based Framework for Validating Compiler Optimizations*. In Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 108-117, 2006.



Further Reading for Chapter 16 (17)

-  George C. Necula. *Translation Validation for an Optimizing Compiler*. In Proceedings of the 20th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000), ACM SIGPLAN Notices 35:83-95, 2000.
-  Amir Pnueli, Michael Siegel, Eli Singerman. *Translation Validation*. In Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Springer-V., LNCS 1384, 151-166, 1998.

Further Reading for Chapter 16 (18)

-  Amir Pnueli, Ofer Strichman, Michael Siegel. *Translation Validation for Synchronous Languages*. In Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98), Springer-V., LNCS 1443, 235-246, 1998.
-  Amir Pnueli, Ofer Strichman, Michael Siegel. *The Code Validation Tool (CVT) – Automatic Verification of a Compilation Process*. International Journal on Software Tools for Technology Transfer 2(2):192-201, 1998.
-  Jean-Baptiste Tristan, Xavier Leroy. *Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations*. In Conference Record of the 35th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2008), 17-27, 2008.

Further Reading for Chapter 16 (19)

-  Jean-Baptiste Tristan, Xavier Leroy. *Verified Validation of Lazy Code Motion*. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), ACM SIGPLAN Notices 44:316-326, 2009.
-  Anna Zaks, Amir Pnueli. *CovaC: Compiler Validation by Program Analysis of the Cross-product*. In Proceedings of the 15th International Symposium on Formal Methods (FM 2008), Springer-V., LNCS 5014, 35-51, 2008.
-  Lenore Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg. *VOC: A Methodology for Translation Validation of Optimizing Compilers*. Journal of Universal Computer Science 9(3):223-247, 2003.

Further Reading for Chapter 16 (20)

New Programming Models and Architectures

-  David Kaeli. *The Road to new Programming Models and Architectures for Future Heterogeneous Systems*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides:
http://old.hipeac.net/system/files/David%20Kaeli_reduced.pdf
-  Margaret Martonosi. *Power-Aware Computing: Then, Now, and into the Future*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides:
<http://old.hipeac.net/system/files/MartonosiHIPEACfinal.pdf>

Further Reading for Chapter 16 (21)

 András Vajda. *Programming Many-Core Chips*.

Springer-V., 2011. (Chapter 1.1, The End of Endless Scalability; Chapter 2, Multi-core and Many-core Processor Architecture; Chapter 3, State of the Art Multi-core Operating Systems; Chapter 10, Looking Ahead)

References

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1397/18

Recommended Reading






...for deepened and independent studies.

- ▶ I Textbooks
- ▶ II On-line Compilers Resources, Compiler Writing Tools
- ▶ III Monographs, Handbooks, Volumes
- ▶ IV Articles






I Textbooks (1)

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2. aktualisierte Auflage, 2008.
-  Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002.
-  Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1997.
-  Andrew W. Appel with Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.







I Textbooks (2)

-  Andrew W. Appel with Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.
-  André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
-  Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Vieweg+Teubner, 2008.
-  Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012.
-  Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013.

I Textbooks (3)

-  Rudolf Berghammer. *Mathematik für Informatiker: Grundlegende Begriffe und Strukturen*. Springer-V., 2014.
-  Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004.
-  Peter Crawley, Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.
-  Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002.






I Textbooks (4)

-  Marcel Erné. *Einführung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
-  Shimon Even. *Graph Algorithms*. Pitman, 1979.
-  C. Fischer, R. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Co., Inc. Menlo Park, CA, 1988.
-  Helmuth Gericke. *Theorie der Verbände*. Bibliographisches Institut, 2. Auflage, 1967.
-  George Grätzer. *General Lattice Theory*. Birkhäuser, 2nd edition, 1998.
-  George Grätzer. *Lattice Theory: Foundation*. Birkhäuser, 2011.







I Textbooks (5)

-  Dick Grune, Criel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-V., 2nd edition, 2008.
-  Dick Grune, Kees van Reeuwijk, Henri E. Bal, Criel J.H. Jacobs, Koen G. Langendoen. *Modern Compiler Design*. Springer-V., 2nd edition, 2012.
-  Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001.
-  Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
-  Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.






I Textbooks (6)

-  Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009.
-  Janusz Laski, William Stanley. *Software Verification and Analysis*. Springer-V., 2009.
-  Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.
-  Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998.
-  David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008.

I Textbooks (7)

-  Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
-  Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997.
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005.
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007.
-  Steven Roman. *Lattices and Ordered Sets*. Springer-V., 2008.




I Textbooks (8)

-  Helmut Seidl, Reinhard Wilhelm, Sebastian Hack. *Compiler Design: Analysis and Transformation*. Springer-V., 2012.
-  Patrick D. Terry. *Compilers and Compiler Generators: An Introduction with C++*. International Thomson Computer Press, 1997.
-  Patrick D. Terry. *Compiling with C# and Java*. Addison-Wesley, 2005.
-  András Vajda. *Programming Many-Core Chips*. Springer-V., 2011.
-  William M. Waite, Lynn R. Carter. *An Introduction to Compiler Construction*. HarperCollins College Publishers, 1993.




I Textbooks (9)

-  William M. Waite, Gerhard Goos. *Compiler Construction*. Springer-V., 1984.
-  Reinhard Wilhelm, Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.
-  Reinhard Wilhelm, Dieter Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-V., 2. Auflage, 1997.
-  Reinhard Wilhelm, Helmut Seidl. *Compiler Design: Virtual Machines*. Springer-V., 2010.
-  Reinhard Wilhelm, Helmut Seidl, Sebastian Hack. *Compiler Design: Syntactic and Semantic Analysis*. Springer-V., 2013.

II On-line Compiler Resources, Compiler Writing Tools (1)

-  Jack Crenshaw. *Let's build a Compiler*. A set of tutorial articles, on-line published, 1988-1995.
<http://www.iecc.com/compilers/crenshaw>
-  German National Research Center for Information Technology, Fraunhofer Institute for Computer Architecture and Software Technology. *The Catalog of Compiler Construction Tools*, 1996-2006.
<http://catalog.compilertools.net/>
-  Compilers.net Team. *Search Machine on Compilers and Programming Languages, Directory of Compiler and Language Resources*, 1997-2007.
<http://www.compilers.net>

II On-line Compiler Resources, Compiler Writing Tools (2)

-  Nullstone Corporation. *The Compiler Connection: A Resource for Compiler Developers and Those who use Their Products and Services (Books, Tools, Techniques, Conferences, Jobs and more, 2011-2012.*
<http://www.compilerconnection.com>
-  Olaf Langmack. *Catalog of Compiler Construction Products 01-98.* 13th Issue, 1998.
<http://compilers.iecc.com/tools.html>
-  William M. Waite, Uwe Kastens et al. *Eli: Translator Construction made Easy, 1989-today.*
<http://eli-project.sourceforge.net/>






II On-line Compiler Resources, Compiler Writing Tools (3)

-  Free Software Foundation (FSF). *GCC, the GNU Compiler Collection*. <https://gcc.gnu.org/>
-  LLVM Foundation. *The LLVM Compiler Infrastructure*. <http://llvm.org/>
-  The SUIF Group (Monica S. Lam et al.), Stanford University. *The SUIF (Stanford University Intermediate Format) Compiler System*. <http://suif.stanford.edu/>
-  Sable Research Group (Laurie Hendren et al.), McGill University, Secure Software Engineering Group (Eric Bodden et al.), TU Darmstadt/U. Paderborn. *Soot: A Framework for Analyzing and Transforming Java and Android Applications*. <https://sable.github.io/soot/>




III Monographs, Handbooks, Volumes (1)

-  Roland Backhouse, Roy Crole, Jeremy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002.
-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. I*. Birkhäuser, 2014.
-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. II*. Birkhäuser, 2016.
-  Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998.




III Monographs, Handbooks, Volumes (2)

-  Yuri V. Matijasevič. *Hilbert's Tenth Problem*. MIT Press, 1993.
-  Markus Müller-Olm. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*. Springer-V., LNCS 3800, 2006.
-  Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.
-  Oliver Rüthing. *Interacting Code Motion Transformations: Their Impact and Their Complexity*. Springer-V., LNCS 1539, 1998.
-  Y. N. Srikant, Priti Shankar (Eds.). *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.




III Monographs, Handbooks, Volumes (3)

-  Y. N. Srikant, Priti Shankar (Eds.) *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2nd edition, 2008.
-  Arnold Thackray, David Brock, Rachel Jones. *Moore's Law: The Life of Gordon Moore. Silicon Valley's Quiet Revolutionary*. Basic Books, New York, 2015.
-  John Whaley. *Context-sensitive Pointer Analysis using Binary Decision Diagrams*. PhD Thesis, Stanford University, CA, USA, 2007.



IV Articles (1)

-  Frances E. Allen, John A. Cocke. *A Program Data Flow Analysis Procedure*. *Communications of the ACM* 19(3):137-147, 1976.
-  Frances E. Allen, John Cocke, Ken Kennedy. *Reduction of Operator Strength*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 3, 79-101.
-  L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *Finding Effective Compilation Sequences*. In Proceedings of the 2004 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), ACM SIGPLAN Notices 39(7):231-239, 2004.




IV Articles (2)

-  Bowen Alpern, Mark N. Wegman, F. Kenneth Zadeck. *Detecting Equality of Variables in Programs*. In Conference Record of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 1-11, 1988.
-  Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, Cristina Silvano. *A Survey on Compiler Auto-tuning using Machine Learning*. *Computing Surveys* 51(5):96.1-96.42, 2019.
-  Clement A. Baker-Finch, Kevin Glynn, Simon L. Peyton Jones. *Constructed Product Result Analysis for Haskell*. *Journal of Functional Programming* 14(2):211-245, 2004.



IV Articles (3)

-  Thomas Ball, Sriram K. Rajamani. *Bebop: A Path-Sensitive Interprocedural Dataflow Engine*. In Proceedings of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), 97-103, 2001.
-  Clark W. Barret, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, Lenore Zuck. *TVOC: A Translation Validator for Optimizing Compilers*. In Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005), Springer-V., LNCS 3576, 291-295, 2005.

IV Articles (4)

-  Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, Jean Souyris. *Towards Formally Verified Optimizing Compilation in Flight Control Software*. In Proceedings of the Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011), 59-68, 2011.
-  Marc Berndt, Ondřej Lhoták, Feng Qian, Laurie Hendren, Navindra Umanee. *Points-to Analysis using BDDs*. In Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003), ACM SIGPLAN Notices 38:103-114, 2003.
-  Garret Birkhoff. *Applications of Lattice Algebra*. Mathematical Proceedings of the Cambridge Philosophical Society 30(2):115-122, 1934.




IV Articles (5)

-  Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeny, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Nathan Keynes, Nathaniel Nystrom, Andreas Zeller. *The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations*. *ACM Transactions on Programming Languages and Systems* 38(4), Article 15:1-20, 2016.
-  Ras Bodik, Rajiv Gupta. *Partial Dead Code Elimination using Slicing Transformations*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, *ACM SIGPLAN Notices* 32:159-170, 1997.



IV Articles (6)

-  Ras Bodik, Rajiv Gupta. *Register Pressure Sensitive Redundancy Elimination*. In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 107-121, 1999.
-  Ras Bodik, Rajiv Gupta, Mary Lou Soffa. *Complete Removal of Redundant Expressions*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), ACM SIGPLAN Notices 33(5):1-14, 1998.
-  Martin Bravenboer, Yannis Smaragdakis. *Strictly Declarative Specification of Sophisticated Points-to Analyses*. In Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009), 243-262, 2009.




IV Articles (7)

-  Preston Briggs, Keith D. Cooper. *Effective Partial Redundancy Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):159-170, 1994.
-  Preston Briggs, Keith D. Cooper, L. Taylor Simpson. *Value Numbering*. Software: Practice and Experience 27(6):701-724, 1997.
-  Michael G. Burke, Paul R. Carini, Jong-Deok Choi, Michael Hind. *Flow-insensitive Interprocedural Alias Analysis in the Presence of Pointers*. In Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC'94), Springer-V., LNCS 892, 234-350, 1994.



IV Articles (8)

-  Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard von Karger, Yassine Lakhnech, Hans Langmaack, Markus Müller-Olm. *Provably Correct Compiler Development and Implementation*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-V., LNCS 641, 141-155, 1992.
-  Qiong Cai, Jingling Xue. *Optimal and Efficient Speculation-based Partial Redundancy Elimination*. In Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2003), 91-104, 2003.



IV Articles (9)

-  David Callahan, Steve Carr, Ken Kennedy. *Improving Register Allocation for Subscripted Variables*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), ACM SIGPLAN Notices 25:53-65, 1990.
-  Steve Carr, Ken Kennedy. *Scalar Replacement in the Presence of Conditional Control Flow*. *Software: Practice and Experience* 24(1):51-77, 1994.
-  Venkatesan T. Chakaravarthy. *New Results on the Computability and Complexity of Points-to-Analysis*. In Conference Record of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 115-125, 2003.




IV Articles (10)

-  Larry Carter, Jeanne Ferrante, Clark Thomborson. *Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger*. In Conference Record of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 106-114, 2003.
-  David R. Chase, Mark N. Wegmann, F. Kenneth Zadeck. *Analysis of Pointers and Structures*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), ACM SIGPLAN Notices 25:296-310, 1990.
-  Li-Ling Chen, Youfeng Wu. *Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation*. In Proceedings of the 2003 IEEE International Conference on Parallel Processing (ICPP 2003), 607-614, 2003.



IV Articles (11)

-  Jong-Deok Choi, Michael G. Burke, Paul R. Carini. *Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects*. In Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), 49-61, 1993.
-  John-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. *Escape Analysis for Java*. In Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99), 1-19, 1999.

IV Articles (12)

-  Fred C. Chow, Sun Chan, Robert Kennedy, Shing-Ming Liu, Raymond Lo, Peng Tu. *A New Algorithm for Partial Redundancy Elimination based upon SSA Form*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM SIGPLAN Notices 32(5):273-286, 1997.
-  Cliff Click. *Global Code Motion, Global Value Numbering*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):246-257, 1995.
-  Cliff Click, Keith D. Cooper. *Combing Analyses, Combining Optimizations*. ACM Transactions on Programming Languages and Systems 17(2):181-196, 1995.

IV Articles (13)

-  John Cocke, Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 2nd Revised Version, 771 pages, 1970.
-  Robert Colwell. *The Chip Design Game at the End of Moore's Law*. In Proceedings of the IEEE/ACM Symposium on High-Performance Chips (Hot Chips 2013), 2013. http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.15-keynote1-Chipdesign-epub/HC25.26.190-Keynote1-ChipDesignGame-Colwell-DARPA.pdf

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11




Chap. 12

Chap. 13




Part IV

1426/18




IV Articles (14)

-  Melvin E. Conway. *Proposal for an UNCOL*. Communications of the ACM 1(3):5, 1958.
-  Keith D. Cooper, Jason Eckhardt, Ken Kennedy. *Redundancy Elimination Revisited*. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008), 12-21, 2008.
-  Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *ACME: Adaptive Compilation Made Efficient*. In Proceedings of the 2005 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005), 69-77, 2005.





IV Articles (15)

-  Keith D. Cooper, L. Taylor Simpson, Christopher A. Vick. *Operator Strength Reduction*. ACM Transactions on Programming Languages and Systems 23(5):603-625, 2001.
-  Keith D. Cooper, Devika Subramanian, Linda Torczon. *Adaptive Optimizing Compilers for the 21st Century*. The Journal of Supercomputing 23(1):7-22, 2002.
-  Chris Cummins, Pavlos Petoumenos, Zheng Wang, Hugh Leather. *End-to-end Deep Learning of Optimization Heuristics*. In Proceedings of the 26th ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT 2017), 219-232, 2017.

IV Articles (16)

-  Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. *ACM Transactions on Programming Languages and Systems* 13(4):451-490, 1991.
-  Martin Davis. *Hilbert's Tenth Problem is Unsolvable*. *American Mathematical Monthly* 80:33-269, 1973.
-  Martin Davis, Yuri Matijasevič, Julia Robinson. *Hilbert's Tenth Problem. Diophantine Equations: Positive Aspects of a Negative Solution*. In *Proceedings of the Symposium on the Hilbert Problems (De Kalb, Illinois), May 1974*, American Mathematical Society, Providence, R.I., 323-378, 1976.




IV Articles (17)

-  Peter J. Denning, Ted G. Lewis. *Exponential Laws of Computing Growth*. *Communications of the ACM* 60(1):54-65, 2017.
-  Dhananjay M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. *International Journal of Computer Mathematics* 27:1-14,31-32, 1989.
-  Dhananjay M. Dhamdhere. *Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise*. *ACM Transactions on Programming Languages and Systems* 13(2):291-294, 1991, Technical Correspondence.
-  Dhananjay M. Dhamdhere. *E-path_pre: Partial Redundancy Elimination Made Easy*. *ACM SIGPLAN Notices* 37(8):53-65, 2002.




IV Articles (18)

-  Dhananjay M. Dhamdhere, J. R. Isaac. *A Composite Algorithm for Strength Reduction and Code Movement Optimization*. *International Journal of Computer and Information Sciences* 9(3):243-273, 1980.
-  Karl-Heinz Drechsler, Manfred P. Stadel. *A Solution to a Problem with Morel and Renvoise's "Global Optimization by Suppression of Partial Redundancies"*. *ACM Transactions on Programming Languages and Systems* 10(4):635-640, 1988, Technical Correspondence.
-  Karl-Heinz Drechsler, Manfred P. Stadel. *A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION*. *ACM SIGPLAN Notices* 28(5):29-38, 1993.




IV Articles (19)

-  Maryam Emami, Rakesh Ghiya, Laurie J. Hendren. *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):242-256, 1994.
-  Andrei P. Ershov. *On Programming of Arithmetic Operations*. Communications of the ACM 1(8):3-6, 1958. (Three figures from this article are in CACM 1(9):16).
-  Christian Fecht, Helmut Seidl. *An Even Faster Solver for General Systems of Equations*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 189-204, 1996.




IV Articles (20)

-  Christian Fecht, Helmut Seidl. *Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems*. In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 90-104, 1998.
-  Christian Fecht, Helmut Seidl. *A Faster Solver for General Systems of Equations*. *Science of Computer Programming* 35(2):137-161, 1999.
-  Grigori Fursin, Michael F.P. O'Boyle, Peter M.W. Knijnenburg. *Evaluating Iterative Compilation*. In Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing (LCPC 2002), Revised Papers, Springer-V., LNCS 2481, 362-376, 2005.

IV Articles (21)

-  Gesellschaft für Informatik e.V. (GI), Fachbereich Softwaretechnik. *Verlässlichkeit von Software*, 2014. <https://www.gi.de/themen/grand-challenges/verlaesslichkeit-von-software.html>
-  Alfons Geser, Jens Knoop, Gerald Lüttgen, Oliver Rüthing, Bernhard Steffen. *Non-Monotone Fixpoint Iterations to Resolve Second Order Effects*. In Proceedings of the 6th International Conference on Compiler Construction (CC'96), Springer-V., LNCS 1060, 106-120, 1996.
-  Rakesh Ghiya, Laurie J. Hendren. *Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 1-15, 1996.





IV Articles (22)

-  Sabine Glesner, Gerhard Goos, Wolf Zimmermann. *Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers)*. *it - Information Technology* 46(5):265-276, 2004.
-  Gerhard Goos, Wolf Zimmermann. *Verification of Compilers*. *Correct System Design: Recent Insight and Advances*. Springer-V., LNCS 1710, 201-230, 1999.
-  Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, William M. Waite. *Eli: A Complete, Flexible Compiler Construction System*. *Communications of the ACM* 35(2):121-131, 1992.




IV Articles (23)

-  Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Redundancy Elimination Using Speculation*. In Proceedings of the 6th IEEE International Conference on Computer Languages (ICCL'98), 230-239, 1998.
-  Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Dead Code Elimination using Predication*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 102-115, 1997.
-  Ben Hardekopf, Calvin Lin. *Semi-sparse Flow-sensitive Pointer Analysis*. In Conference Record of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009), 226-238, 2009.

IV Articles (24)

-  Charles A.R. Hoare. *The Verifying Compiler: A Grand Challenge for Computing Research*. *Journal of the ACM* 50(1):63-69, 2003.
-  R. Nigel Horspool, H. C. Ho. *Partial Redundancy Elimination Driven by a Cost-benefit Analysis*. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering (CSSE'97)*, 111-118, 1997.
-  R. Nigel Horspool, David J. Pereira, Bernhard Scholz. *Fast Profile-based Partial Redundancy Elimination*. In *Proceedings of the 7th Joint Modular Languages Conference (JMLC 2006)*, 362-376, 2006.
-  Susan Horwitz. *Precise Flow-insensitive Alias Analysis is NP-hard*. *ACM Transactions on Programming Languages and Systems* 19(1):1-6, 1997.




IV Articles (25)

-  Susan Horwitz, Alan J. Demers, Tim Teitelbaum. *An Efficient General Iterative Algorithm for Dataflow Analysis*. *Acta Informatica* 24(6):679-694, 1987.
-  IEEE Spectrum. *Special Report on 50 Years of Moore's Law*. IEEE Spectrum, April 2015.
<http://spectrum.ieee.org/static/special-report-50-years-of-moores-law>
-  Bertrand Jeannot, Alexey Logivon, Thomas W. Reps, Mooly Sagiv. *A Relational Approach to Interprocedural Shape Analysis*. In *Proceedings of the 11th Static Analysis Symposium (SAS 2004)*, Springer-V., LNCS 3248, 246-264, 2004.

IV Articles (26)

-  S. M. Joshi, Dhananjay M. Dhamdhere. *A Composite Hoisting-strength Reduction Transformation for Global Program Optimization – Part I and Part II*. *International Journal of Computer Mathematics* 11:21-41,111-126, 1982.
-  David Kaeli. *The Road to new Programming Models and Architectures for Future Heterogeneous Systems*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides:
http://old.hipeac.net/system/files/David%20Kaeli_reduced.pdf
-  John B. Kam, Jeffrey D. Ullman. *Global Data Flow Analysis and Iterative Algorithms*. *Journal of the ACM* 23:158-171, 1976.

IV Articles (27)

-  John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. *Acta Informatica* 7:305-317, 1977.
-  Aditya Kanade, Amitabha Sanyal, Uday Khedker. *A PVS based Framework for Validating Compiler Optimizations*. In Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 108-117, 2006.
-  Vini Kanvar, Uday P. Khedker. *Heap Abstractions for Static Analysis*. *ACM Computing Surveys* 49(2):29, 47 pages, 2017.
-  Ken Kennedy. *Safety of Code Motion*. *International Journal of Computer Mathematics* 3(2-3):117-130, 1972.



IV Articles (28)

-  Robert Kennedy, Sun Chan, Shing-Ming Liu, Raymond Lo, Peng Tu, Fred C. Chow. *Partial Redundancy Elimination in SSA Form*. ACM Transactions of Programming Languages and Systems 32(3):627-676, 1999.
-  Robert Kennedy, Fred C. Chow, Peter Dahl, Shing-Ming Liu, Raymond Lo, Mark Streich. *Strength Reduction via SSAPRE*. In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 144-158, 1998.
-  Uday P. Khedker, Bageshri Karkare. *Efficiency, Precision, Simplicity, and Generality in Interprocedural Dataflow Analysis: Resurrecting the Classical Call Strings Method*. In Proceedings of the 17th International Conference on Compiler Construction (CC 2008), Springer-V., LNCS 4959, 213-228, 2008.




IV Articles (29)

-  Uday P. Khedker, Alan Mycroft, Prashant Singh Rawat. *Liveness-based Pointer Analysis*. In Proceedings of the 19th Static Analysis Symposium (SAS 2012), Springer-V., LNCS 7460, 265-282, 2012.
-  Gary A. Kildall. *A Unified Approach to Global Program Optimization*. In Conference Record of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73), 194-206, 1973.
-  Se-Won Kim, Xavier Rival, Sukyoung Ryu. *A Theoretical Framework of Sensitivity in an Abstract Interpretation Framework*. ACM Transactions on Programming Languages and Systems 40(3), 13:1-44, 2018.




IV Articles (30)

-  Toru Kisuki, Peter M.W. Knijnenburg, Michael F.P. O'Boyle, François Bodin, Harry A.G. Wijshoff. *A Feasability Study in Iterative Compilation*. In Proceedings of the 2nd International Symposium on High Performance Computing (ISHPC'99), Springer-V., LNCS 1615, 121-132, 1999.
-  Marion Klein, Jens Knoop, Dirk Koschützki, Bernhard Steffen. *DFA&OPT-METAFrame: A Toolkit for Program Analysis and Optimization*. In Proceedings of the 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), Springer-V., LNCS 1055, 422-426, 1996.

IV Articles (31)

-  Peter M.W. Knijnenburg, Toru Kisuki, Michael F.P. O'Boyle. *Iterative Compilation*. In Proceedings of the 1st Workshop on Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS 2001), Springer-V., LNCS 2268, 171-187, 2002.
-  Kathleen Knobe, Vivek Sarkar. *Conditional Constant Propagation of Scalar and Array References Using Array SSA Form*. In Proceedings of the 5th Static Analysis Symposium (SAS'98), Springer-V., LNCS 1503, 33-56, 1998.
-  Kathleen Knobe, Vivek Sarkar. *Array SSA Form and its Use in Parallelization*. In Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98), 107-120, 1998.

IV Articles (32)

-  Jens Knoop. *Parallel Constant Propagation*. In Proceedings of the 4th European Conference on Parallel Processing (Euro-Par'98), Springer-V., LNCS 1470, 445-455, 1998.
-  Jens Knoop. *Formal Callability and its Relevance and Application to Interprocedural Data Flow Analysis*. In Proceedings of the 6th IEEE International Conference on Computer Languages (ICCL'98), 252-261, 1998.
-  Jens Knoop. *From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Springer-V., LNCS 1579, 360-374, 1999.




IV Articles (33)

-  Jens Knoop, Dirk Koschützki, Bernhard Steffen. *Basic-block Graphs: Living Dinosaurs?* In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 65-79, 1998.
-  Jens Knoop, Eduard Mehofer. *Optimal Distribution Assignment Placement.* In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Springer-V., LNCS 1300, 364-373, 1997.
-  Jens Knoop, Eduard Mehofer. *Interprocedural Distribution Assignment Placement: More than just Enhancing Intraprocedural Placing Techniques.* In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 26-37, 1997.


IV Articles (34)

-  Jens Knoop, Oliver Rüthing. *Constant Propagation on the Value Graph: Simple Constants and Beyond*. In Proceedings of the 9th International Conference on Compiler Construction (CC 2000), Springer-V., LNCS 1781, 94-109, 2000.
-  Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. In Proceedings of the 7th Brazilian Symposium on Programming Languages (SBLP 2003), 135-148, 2003.
-  Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. Journal of Universal Computer Science 9(8):829-850, 2003. (Special issue for SBLP'03)

IV Articles (35)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. Journal of Programming Languages 1(1):71-91, 1993.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.




IV Articles (36)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Code Motion and Code Placement: Just Synonyms?* In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 154-169, 1998.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Expansion-based Removal of Semantic Partial Redundancies.* In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.




IV Articles (37)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Partial Dead Code Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):147-158, 1994.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *The Power of Assignment Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), ACM SIGPLAN Notices 30(6):233-245, 1995.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retrospective: Lazy Code Motion*. In "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection", ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.




IV Articles (38)

-  Jens Knoop, Bernhard Steffen. *The Interprocedural Coincidence Theorem*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-V., LNCS 641, 125-140, 1992.
-  Jens Knoop, Bernhard Steffen. *Code Motion for Explicitly Parallel Programs*. In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), ACM SIGPLAN Notices 34(8):13-24, 1999.
-  Donald E. Knuth. *An Empirical Study of Fortran Programs*. Software: Practice and Experience 1:105-133, 1971.




IV Articles (39)

-  Andreas Krall. *Correct Compilers for Correct Processors*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: <http://old.hipeac.net/system/files/hipeac14.pdf>
-  Sameer Kulkarni, John Cavazos. *Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning*. Proceedings of the ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2012), 147-162, 2012.
-  William Landi. *Undecidability of Static Analysis*. ACM Letters on Programming Languages and Systems 1(4):323-337, 1992.




IV Articles (40)

-  Hans Langmaack. *Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit*. *it+ti - Informationstechnik und Technische Informatik* 39(3):41-47, 1997.
-  Hans Langmaack: *The ProCoS Approach to Correct Systems*. *Real-Time Systems* 13(3):253-275, 1997.
-  Chris Lattner, Vikram S. Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*. In *Proceedings of the 3rd Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004)*, 75-88, 2004.

IV Articles (41)

-  Chris Lattner, Vikram S. Adve. *The LLVM Compiler Framework and Infrastructure Tutorial*. In Proceedings of the 17th International Workshop on Languages and Compilers for High Performance Computing (LCPC 2004), Revised Selected Papers, 15-16, 2004.
-  Xavier Leroy. *Formal Verification of a Realistic Compiler*. Communications of the ACM 52(7):107-115, 2009.
-  Xavier Leroy. *Formally Verifying a Compiler: Why? How? How far?* Invited Talk, In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011), xxxi, 2011.




IV Articles (42)

-  Xavier Leroy. *Compiler Verification for Fun and Profit*. Invited Talk, In Proceedings of the 2014 IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD 2014), 9, 2014.
-  Roman Manevich, Boris Dogadov, Noam Rinetzky. *From Shape Analysis to Termination Analysis in Linear Time*. In Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), Springer-V., LNCS 9779, 426-446, 2016.
-  Roman Manevich, Mooly Sagiv, Ganesan Ramalingam, John Field. *Partially Disjunctive Heap Abstraction*. In Proceedings of the 11th Static Analysis Symposium (SAS 2004), Springer-V., LNCS 3248, 265-279, 2004.




IV Articles (43)

-  Ravi Mangal, Mayur Naik, Hongseok Yang. *A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join*. In *Proceedings of the 23rd European Symposium on Programming (ESOP 2014)*, Springer-V., LNCS 8410, 513-533, 2014.
-  George Markowsky. *Chain-complete Posets and Directed Sets with Applications*. *Algebra Universalis* 6(1):53-68, 1976.
-  Thomas J. Marlowe, Barbara G. Ryder. *Properties of Data Flow Frameworks*. *Acta Informatica* 28(2):121-163, 1990.



IV Articles (44)

-  Florian Martin. *PAG - An Efficient Program Analyzer Generator*. *Journal of Software Tools for Technology Transfer* 2(1):46-67, 1998.
-  Margaret Martonosi. *Power-Aware Computing: Then, Now, and into the Future*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: <http://old.hipeac.net/system/files/MartonosiHIPEACfinal.pdf>
-  Stephen P. Masticola, Thomas J. Marlowe, Barbara G. Ryder. *Lattice Frameworks for Multisource and Bidirectional Data Flow Problems*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17(5):777-803, 1995.




IV Articles (45)

-  Yuri V. Matijasevič. *Enumerable Sets are Diophantine (in Russian)*. Doklady Akademii Nauk SSSR 191:279-282, 1970 (english translation: *Soviet Mathematics Doklady* 11:354-357, 1970).
-  Yuri V. Matijasevič. *On Recursive Unsolvability of Hilbert's Tenth Problem*. In *Proceedings of the 4th International Congress on Logic, Methodology and Philosophy of Science (Bucharest 1971)*, North-Holland, Amsterdam, 89-110, 1973.
-  Yuri V. Matijasevič. *What Should We Do Having Proved a Decision Problem to be Unsolvable?* In *Proceedings of Algorithms in Modern Mathematics and Computer Science*, Springer-V., LNCS 122, 441-448, 1979.




IV Articles (46)

-  John McCarthy, James Painter. *Correctness of a Compiler for Arithmetical Expressions*. Mathematical Aspects of Computer Science, ser. Proceedings of Symposia in Applied Mathematics, Vol. 19, American Mathematical Society, 33-41, 1967.
-  Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Leaf Petersen. *A Verifiable SSA Program Representation for Aggressive Compiler Optimization*. In Conference Record of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), 397-408, 2006.



IV Articles (47)

-  Samuel P. Midkiff, José E. Moreira, Marc Snir. *A Constant Propagation Algorithm for Explicitly Parallel Programs*. International Journal of Computer Science 26(5):563-589, 1998.
-  Matthew Might, Yannis Smaragdakis, David Van Horn. *Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. OO Program Analysis*. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), ACM SIGPLAN Notices 45(6):305-315, 2010.
-  Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to and Side-effect Analyses for JAVA*. In Proceedings of the 6th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), 1-11, 2002.




IV Articles (48)

-  Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to Analysis for JAVA*. ACM Transactions on Software Engineering and Methodology 14(4):431-477, 2005.
-  Robin Milner, R. Weyrauch. *Proving Compiler Correctness in a Mechanized Logic*. In Proceedings of the 7th Annual Machine Intelligence Workshop, ser. Machine Intelligence, B. Meltzer and D. Michie, Eds., Vol. 7., Edinburgh University Press, 51-72, 1972.
-  Don Monroe. *Electronics are Leaving the Plane*. Communications of the ACM 61(8):17-18, 2018.



IV Articles (49)

-  Don Monroe. *And Then, There Were Three — How long can the silicon foundry sector continue to adapt, as physical limits make further shrinkage virtually impossible?* Communications of the ACM 62(6):16-18, 2019.
-  Gordon E. Moore. *Cramming More Components onto Integrated Circuits.* Electronics 38(8), 114-117, 1965. http://web.eng.fiu.edu/npala/eee6397ex/gordon_moore_1965_article.pdf




IV Articles (50)

-  Gordon E. Moore. *Progress in Digital Integrated Electronics*. International Electron Devices Meeting, IEEE, IEDM Tech. Digest, 11–13, 1975.
http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf
-  Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. *Communications of the ACM* 22(2):96-103, 1979.
-  Markus Müller-Olm. *The Complexity of Copy Constant Detection in Parallel Programs*. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2001)*, Springer-V., LNCS 2010, 490-501, 2001.

IV Articles (51)

-  Markus Müller-Olm, Oliver Rüthing. *The Complexity of Constant Propagation*. In Proceedings of the European Symposium on Programming (ESOP 2001), Springer-V., LNCS 2028, 190-205, 2001.
-  Markus Müller-Olm, Helmut Seidl. *Polynomial Constants are Decidable*. In Proceedings of the 9th Static Analysis Symposium (SAS 2002), Springer-V., LNCS 2477, 4-19, 2002.
-  Robert Muth, Saumya K. Debray. *On the Complexity of Flow-Sensitive Dataflow Analyses*. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), 67-80, 2000.




IV Articles (52)

-  George C. Necula. *Translation Validation for an Optimizing Compiler*. In Proceedings of the 20th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000), ACM SIGPLAN Notices 35:83-95, 2000.
-  Flemming Nielson. *Semantics-directed Program Analysis: A Tool-maker's Perspective*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 2-21, 1996.
-  Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92), 96-108, 1992.





IV Articles (53)

-  Hemant D. Pande, Barbara Ryder. *Data-flow-based Virtual Function Resolution*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 238-254, 1996.
-  Viktor Pavlu, Markus Schordan, Andreas Krall. *Computation of Alias Sets from Shape Graphs for Comparison of Shape Analysis Precision*. In Proceedings of the 11th International IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2011), 25-34, 2011. [Best Paper Award SCAM 2011]
-  Amir Pnueli, Michael Siegel, Eli Singerman. *Translation Validation*. In Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Springer-V., LNCS 1384, 151-166, 1998.




IV Articles (54)

-  Amir Pnueli, Ofer Strichman, Michael Siegel. *Translation Validation for Synchronous Languages*. In Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98), Springer-V., LNCS 1443, 235-246, 1998.
-  Amir Pnueli, Ofer Strichman, Michael Siegel. *The Code Validation Tool (CVT) – Automatic Verification of a Compilation Process*. International Journal on Software Tools for Technology Transfer 2(2):192-201, 1998.
-  Todd A. Proebsting. *Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years*. <http://proebsting.cs.arizona.edu/law.html>




IV Articles (55)

-  G. Ramalingam. *The Undecidability of Aliasing*. ACM Transactions on Programming Languages and Systems 16(5):1467-1471, 1994.
-  John H. Reif, Harry R. Lewis. *Symbolic Evaluation and the Global Value Graph*. In Conference Record of the 4th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), 104-118, 1977.
-  John H. Reif, Harry R. Lewis. *Efficient Symbolic Analysis of Programs*. Aiken Computation Laboratory, Harvard University, TR-37-82, 1982.
-  Tom Reps, Susan Horwitz, Mooly Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 49-61, 1995.




IV Articles (56)

-  Barry K. Rosen. *High-level Data Flow Analysis*. Communications of the ACM 20(10):141-156, 1977.
-  Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. *Global Value Numbers and Redundant Computations*. In Conference Record of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 12-27, 1988.
-  Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Detecting Equalities of Variables: Combining Efficiency with Precision*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Springer-V., LNCS 1694, 232-247, 1999.


IV Articles (57)

-  Oliver R uthing, Jens Knoop, Bernhard Steffen. *Sparse Code Motion*. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), 170-183, 2000.
-  Mooly Sagiv, Tom Reps, Susan Horwitz. *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation*. In Proceedings of the 6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95), Springer-V., LNCS 915, 651-665, 1995.
-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-analysis Problems in Languages with Destructive Updating*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 16-31, 1996.



IV Articles (58)

-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-Analysis Problems in Languages with Destructive Updating*. ACM Transactions on Programming Languages and Systems 20(1):1-50, 1998.
-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. In Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), 105-118, 1999.
-  Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-valued Logic*. ACM Transactions on Programming Languages and Systems 24(3):217-298, 2002.




IV Articles (59)

-  Bernhard Scholz, R. Nigel Horspool, Jens Knoop. *Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination*. In Proceedings of the 2004 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), ACM SIGPLAN Notices 39(7):221-230, 2004.
-  Helmut Seidl, Christian Fecht. *Interprocedural Analyses: A Comparison*. The Journal of Logic Programming 43:123-156, 2000.
-  John M. Shalf, Robert Leland. *Computing beyond Moore's Law*. IEEE Computer 48(12):14-23, 2015.

IV Articles (60)

-  Marc Shapiro, Susan Horwitz. *Fast and Accurate Flow-insensitive Points-to Analysis*. In Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), 1-14, 1997.
-  Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.3, The Functional Approach to Interprocedural Analysis, 196-209.




IV Articles (61)

-  Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.4, The Call-String Approach to Interprocedural Analysis, 210-217; Chapter 7.6, An Approximative Call-String Approach, 225-230.
-  Olin Shivers. *Control-Flow Analysis in Scheme*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88), ACM SIGPLAN Notices 23:164-174, 1988.
-  Tom Simonite. *Moore's Law Is Dead. Now What?* MIT Technology Review, 2016.
<https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/>


IV Articles (62)

-  Yannis Smaragdakis, Martin Bravenboer, Ondřej Lhoták. *Pick Your Contexts Well: Understanding Object-sensitivity*. In Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), 17-30, 2011.
-  Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Proceedings of the 2nd Joint Conference on Theory and Practice of Software Development (TAPSOFT'87), Springer-V., LNCS 249, 52-68, 1987.
-  Bernhard Steffen. *Property-Oriented Expansion*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 22-41, 1996.

IV Articles (63)

-  Bernhard Steffen, Andreas Claßen, Marion Klein, Jens Knoop, Tiziana Margaria. *The Fixpoint Analysis Machine*. In Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95), Springer-V., LNCS 962, 72-87, 1995.
-  Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. In Proceedings of the 14th International Conference on Mathematical Foundations of Computer Science (MFCS'89), Springer-V., LNCS 379, 481-490, 1989.
-  Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. *Theoretical Computer Science* 80(2):303-318, 1991.




IV Articles (64)

-  Bernhard Steffen, Jens Knoop, Oliver Rüthing. *The Value Flow Graph: A Program Representation for Optimal Program Transformations*. In Proceedings of the 3rd European Symposium on Programming (ESOP'90), Springer-V., LNCS 432, 389-405, 1990.
-  Bernhard Steffen, Jens Knoop, Oliver Rüthing. *Efficient Code Motion and an Adaption to Strength Reduction*. In Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), Springer-V., LNCS 494, 394-415, 1991.
-  Bjarne Steensgaard. *Points-to Analysis in Almost Linear Time*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 32-41, 1996.

IV Articles (65)

-  Rishi Surendran, Rajkishore Barik, Jisheng Zhao, Vivek Sarkar. *Inter-iteration Scalar Replacement using Array SSA Form*. In Proceedings of the 23rd International Conference on Compiler Construction (CC 2014), Springer-V., LNCS 8409, 40-60, 2014.
-  Munehiro Takimoto, Kenichi Harada. *Effective Partial Redundancy Elimination based on Extended Value Graph*. Information Processing Society of Japan 38(11):2237-2250, 1990.
-  Munehiro Takimoto, Kenichi Harada. *Partial Dead Code Elimination Using Extended Value Graph*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Springer-V., LNCS 1694, 179-193, 1999.

IV Articles (66)

-  Robert E. Tarjan. *Fast Algorithms for Solving Path Problems*. Journal of the ACM 28(3):594-614, 1981.
-  Jean-Baptiste Tristan, Xavier Leroy. *Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations*. In Conference Record of the 35th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2008), 17-27, 2008.
-  Jean-Baptiste Tristan, Xavier Leroy. *Verified Validation of Lazy Code Motion*. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), ACM SIGPLAN Notices 44:316-326, 2009.

IV Articles (67)



M. Mitchell Waldrop. *More than Moore*.

Nature 530(7589):144-147, 2016.

http://www.nature.com/polopoly_fs/1.19338!/menu/main/topColumns/topLeftColumn/pdf/530144a.pdf






Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. In Conference Record of the 12th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'85), 291-299, 1985.






Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. ACM Transactions on Programming Languages and Systems 13(2):181-210, 1991.



IV Articles (68)

-  John Whaley, Monica S. Lam. *Cloning-based Context-sensitive Pointer Alias Analysis using Binary Decision Diagrams*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:131-144, 2004.
-  Jingling Xue, Qiong Cai. *A Lifetime Optimal Algorithm for Speculative PRE*. ACM Transactions on Architecture and Code Optimization 3(2):115-155, 2006.
-  Jingling Xue, Jens Knoop. *A Fresh Look at PRE as a Maximum Flow Problem*. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006), Springer-V., LNCS 3923, 139-154, 2006.




IV Articles (69)

-  Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn. *Scalable Shape Analysis for Systems Code*. In Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), Springer-V., LNCS 5123, 385-398, 2008.
-  Anna Zaks, Amir Pnueli. *CovaC: Compiler Validation by Program Analysis of the Cross-product*. In Proceedings of the 15th International Symposium on Formal Methods (FM 2008), Springer-V., LNCS 5014, 35-51, 2008.
-  Sean Zhang, Barbara G. Ryder, William Landi. *Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses*. In Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'96), 81-92, 1996.

IV Articles (70)

-  Hucheng Zhou, Wenguang Chen, Fred C. Chow. *An SSA-based Algorithm for Optimal Speculative Code Motion under an Execution Profile*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), ACM SIGPLAN Notices 46(6):98-108, 2011.
-  Jianwen Zhu, Silvian Calman. *Symbolic Pointer Analysis Revisited*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:145-157, 2004.

IV Articles (71)

-  Wolf Zimmermann. *On the Correctness of Transformations in Compiler Back-Ends*. In Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004), Springer-V., LNCS 4313, 74-95, 2004.
-  Lenore Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg. *VOC: A Methodology for Translation Validation of Optimizing Compilers*. Journal of Universal Computer Science 9(3):223-247, 2003.
-  International Roadmap for Devices and Systems 2017 Edition, IEEE. <https://irds.ieee.org/roadmap-2017>

Appendices

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1485/18

Appendix A

Mathematical Foundations

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1486/18

A.1

Relations

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1487/18

Relations

Let M_i , $1 \leq i \leq k$, be sets.

Definition A.1.1 (k -ary Relation)

A (k -ary) relation is a set R of ordered tuples of elements of M_1, \dots, M_k , i.e., $R \subseteq M_1 \times \dots \times M_k$ is a subset of the cartesian product of the sets M_i , $1 \leq i \leq k$.

Examples

- ▶ \emptyset is the smallest relation on $M_1 \times \dots \times M_k$.
- ▶ $M_1 \times \dots \times M_k$ is the biggest relation on $M_1 \times \dots \times M_k$.

Binary Relations

Let M, N be sets.

Definition A.1.2 (Binary Relation)

A (binary) relation is a set R of ordered pairs of elements of M and N , i.e., R is a subset of the cartesian product of M and N , $R \subseteq M \times N$, called a relation from M to N .

Examples

- ▶ \emptyset is the smallest relation from M to N .
- ▶ $M \times N$ is the biggest relation from M to N .

Note

- ▶ If R is a relation from M to N , it is common to write $m R n$, $R(m, n)$, or $R m n$ instead of $(m, n) \in R$.

Between, On

Definition A.1.3 (Between, On)

A relation R from M to N is called a **relation between M and N** (or a **relation on $M \times N$**).

If M equals N , then R is called a **relation on M** , in symbols: (M, R) .

Domain and Range of a Binary Relation

Definition A.1.4 (Domain and Range)

Let R be a relation from M to N .

The sets

- ▶ $dom(R) =_{df} \{m \mid \exists n \in N. (m, n) \in R\}$
- ▶ $ran(R) =_{df} \{n \mid \exists m \in M. (m, n) \in R\}$

are called the **domain** and the **range** of R , respectively.

Properties of Relations on a Set M

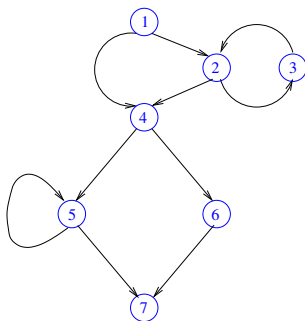
Definition A.1.5 (Properties of Relations on M)

A relation R on a set M is called

- ▶ **reflexive** iff $\forall m \in M. m R m$
- ▶ **irreflexive** iff $\forall m \in M. \neg m R m$
- ▶ **transitive** iff $\forall m, n, p \in M. m R n \wedge n R p \Rightarrow m R p$
- ▶ **intransitive** iff $\forall m, n, p \in M. m R n \wedge n R p \Rightarrow \neg m R p$
- ▶ **symmetric** iff $\forall m, n \in M. m R n \iff n R m$
- ▶ **antisymmetric** iff $\forall m, n \in M. m R n \wedge n R m \Rightarrow m = n$
- ▶ **asymmetric** iff $\forall m, n \in M. m R n \Rightarrow \neg n R m$
- ▶ **linear** iff $\forall m, n \in M. m R n \vee n R m \vee m = n$
- ▶ **total** iff $\forall m, n \in M. m R n \vee n R m$

(Anti-) Example

Let $G = (N, E, s \equiv 1, e \equiv 7)$ be the below (flow) graph, and let R be the relation ' \cdot is linked to \cdot via a (directed) edge' on N of G (e.g., node 4 is linked to node 6 but not vice versa).



The relation R is not reflexive, not irreflexive, not transitive, not intransitive, not symmetric, not antisymmetric, not asymmetric, not linear, and not total.

Equivalence Relation

Let R be a relation on M .

Definition A.1.6 (Equivalence Relation)

R is an **equivalence relation** (or **equivalence**) iff R is reflexive, transitive, and symmetric.

Exercise A.1.7

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or disprove: The divisibility relation $|$ on \mathbb{N}_0 is

1. reflexive
2. irreflexive
3. transitive
4. intransitive
5. symmetric
6. antisymmetric
7. asymmetric
8. linear
9. total
10. equivalence (relation)

Proof or counterexample.

A.2

Ordered Sets

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1496/18

A.2.1

Pre-Orders, Partial Orders, and More

Ordered Sets

Let R be a relation on M .

Definition A.2.1.1 (Pre-Order)

R is a **pre-order** (or **quasi-order**) iff R is reflexive and transitive.

Definition A.2.1.2 (Partial Order)

R is a **partial order** (or **poset** or **order**) iff R is reflexive, transitive, and antisymmetric.

Definition A.2.1.3 (Strict Partial Order)

R is a **strict partial order** iff R is asymmetric and transitive.

Examples of Ordered Sets

Pre-order (reflexive, transitive)

- ▶ The relation \Rightarrow on logical formulas.

Partial order (reflexive, transitive, antisymmetric)

- ▶ The relations $=$, \leq and \geq on \mathbb{IN} .
- ▶ The relation $m \mid n$ (m is a divisor of n) on \mathbb{IN} .

Strict partial order (asymmetric, transitive)

- ▶ The relations $<$ and $>$ on \mathbb{IN} .
- ▶ The relations \subset and \supset on sets.

Equivalence relation (reflexive, transitive, symmetric)

- ▶ The relation \iff on logical formulas.
- ▶ The relation 'have the same prime number divisors' on \mathbb{IN} .
- ▶ The relation 'are citizens of the same country' on people.

Note

- ▶ An antisymmetric pre-order is a partial order; a symmetric pre-order is an equivalence relation.
- ▶ For convenience, also the pair (M, R) is called a pre-order, partial order, and strict partial order, respectively.
- ▶ More accurately, we could speak of the pair (M, R) as of a set M which is pre-ordered, partially ordered, and strictly partially ordered by R , respectively.
- ▶ Synonymously, we also speak of M as a pre-ordered, partially ordered, and a strictly partially ordered set, respectively, or of M as a set which is equipped with a pre-order, partial order and strict partial order, respectively.
- ▶ On any set, the equality relation $=$ is a partial order, called the discrete (partial) order.

The Strict Part of an Ordering

Let \sqsubseteq be a pre-order (reflexive, transitive) on P .

Definition A.2.1.4 (Strict Part of \sqsubseteq)

The relation \sqsubset on P defined by

$$\forall p, q \in P. p \sqsubset q \iff_{df} p \sqsubseteq q \wedge p \neq q$$

is called the **strict part** of \sqsubseteq .

Corollary A.2.1.5 (Strict Partial Order)

Let (P, \sqsubseteq) be a partial order, let \sqsubset be the strict part of \sqsubseteq .

Then: (P, \sqsubset) is a **strict partial order**.

Useful Results

Let \sqsubset be a strict partial order (asymmetric, transitive) on P .

Lemma A.2.1.6

The relation \sqsubset is irreflexive.

Lemma A.2.1.7

The pair (P, \sqsubseteq) , where \sqsubseteq is defined by

$$\forall p, q \in P. p \sqsubseteq q \iff_{df} p \sqsubset q \vee p = q$$

is a **partial order**.

Induced (or Inherited) Partial Order

Definition A.2.1.8 (Induced Partial Order)

Let (P, \sqsubseteq_P) be a partially ordered set, let $Q \subseteq P$ be a subset of P , and let \sqsubseteq_Q be the relation on Q defined by

$$\forall q, r \in Q. q \sqsubseteq_Q r \iff_{df} q \sqsubseteq_P r$$

Then: \sqsubseteq_Q is called the **induced partial order** on Q (or the **inherited order** from P on Q).

Exercise A.2.1.9

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ' \cdot divides \cdot ' (w/out remainder), e.g. $5 | 35$.

Prove or disprove: The divisibility relation $|$ on \mathbb{N}_0 is a

1. pre-order
2. partial order
3. strict partial order
4. equivalence (relation)

Proof or counterexample.

A.2.2

Hasse Diagrams

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

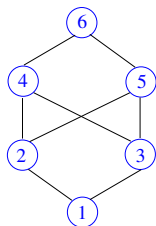
Chap. 13

Part IV

1505/18

Hasse Diagrams

...are a sparse graphical representation of partial orders.



The links of a **Hasse diagram**

- ▶ are read from **below** to **above** (lower means smaller).
- ▶ represent the relation R of ' \cdot is an immediate predecessor of \cdot ' defined by

$$p R q \iff_{df} p \sqsubset q \wedge \nexists r \in P. p \sqsubset r \sqsubset q$$

of a **partial order** (P, \sqsubseteq) , where \sqsubset is the strict part of \sqsubseteq .

Reading Hasse Diagrams

The **Hasse diagram** representation of a **partial order**

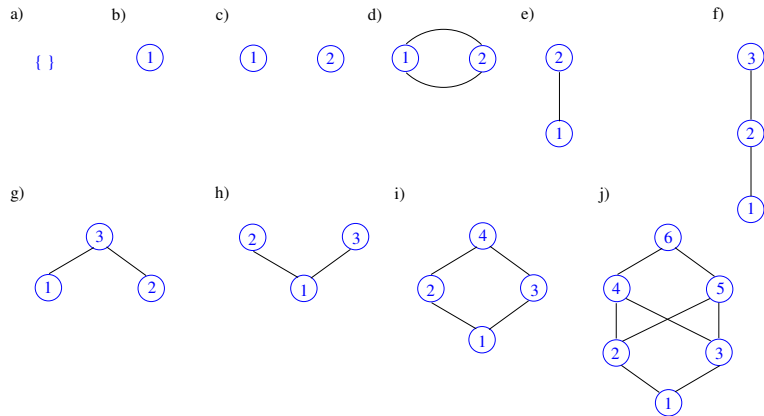
- ▶ omits links which express reflexive and transitive relations explicitly
- ▶ focuses on the 'immediate predecessor' relation.

The representation of a **partial order** by its **Hasse diagram**

- ▶ is sparse and thus economical (in the number of links).
- ▶ while preserving all relevant information of the partial order it represents:
 - ▶ $p \sqsubseteq q \wedge p = q$ (**reflexivity**): trivially represented (just without an explicit link)
 - ▶ $p \sqsubseteq q \wedge p \neq q$ (**transitivity**): represented by ascending paths (with at least one link) from p to q .

Exercise A.2.2.1

Which of the below diagrams are Hasse diagrams representing a partial order?



Exercise A.2.2.2

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Draw an expressive section of the [Hasse diagram](#) of the divisibility relation $|$ on \mathbb{N}_0 .

A.2.3

Bounds and Extremal Elements

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1510/18

Bounds in Pre-Orders

Definition A.2.3.1 (Bounds in Pre-Orders)

Let (Q, \sqsubseteq) be a pre-order, let $q \in Q$ and $Q' \subseteq Q$.

q is called a

- ▶ **lower bound** of Q' , in signs: $q \sqsubseteq Q'$, if $\forall q' \in Q'. q \sqsubseteq q'$
- ▶ **upper bound** of Q' , in signs: $Q' \sqsubseteq q$, if $\forall q' \in Q'. q' \sqsubseteq q$
- ▶ **greatest lower bound (glb)** (or **infimum**) of Q' , in signs: $\sqcap Q'$, if q is a lower bound of Q' and for every other lower bound \hat{q} of Q' holds: $\hat{q} \sqsubseteq q$.
- ▶ **least upper bound (lub)** (or **supremum**) of Q' , in signs: $\sqcup Q'$, if q is an upper bound of Q' and for every other upper bound \hat{q} of Q' holds: $q \sqsubseteq \hat{q}$.

Extremal Elements in Pre-Orders

Definition A.2.3.2 (Extremal Elements in Pre-Ord's)

Let (Q, \sqsubseteq) be a pre-order, let \sqsubset be the strict part of \sqsubseteq , and let $Q' \subseteq Q$ and $q \in Q'$.

q is called a

- ▶ **minimal element** of Q' , if there is no $q' \in Q'$ with $q' \sqsubset q$.
- ▶ **maximal element** of Q' , if there is no $q' \in Q'$ with $q \sqsubset q'$.
- ▶ **least (or minimum) element** of Q' , if $q \sqsubseteq Q'$.
- ▶ **greatest (or maximum) element** of Q' , if $Q' \sqsubseteq q$.

Note: Least and greatest elements of Q itself are usually denoted by \perp and \top (**bottom, top** (in German: **Tief, Hoch**)), respectively, if they exist. **Least (greatest)** elements of Q are always **minimal (maximal)** elements of Q .

Existence and Uniqueness

...of **bounds** and **extremal elements** in **partially ordered sets**.

Let (P, \subseteq) be a partial order, and let $Q \subseteq P$ be a subset of P .

Lemma A.2.3.3 (lub/glb: Unique if Existent)

Least upper bounds, greatest lower bounds, least elements, and greatest elements in Q are **unique**, if they exist.

Lemma A.2.3.4 (Minimal/Maximal El.: Not Unique)

Minimal and maximal elements in Q are usually **not unique**.

Note: Lemma A.2.3.3 suggests considering \sqcup and \sqcap partial maps $\sqcup, \sqcap : \mathcal{P}(P) \rightarrow P$ from the powerset $\mathcal{P}(P)$ of P to P .

Lemma A.2.3.3 does not hold for pre-orders.

Characterization of Least, Greatest Elements

...in terms of *infima* and *suprema* of sets.

Let (P, \sqsubseteq) be a partial order.

Lemma A.2.3.5 (Characterization of \perp and \top)

The *least element* \perp and the *greatest element* \top of P are given by the *supremum* and the *infimum* of the *empty set*, and the *infimum* and the *supremum* of P , respectively, i.e.,

$$\perp = \bigsqcup \emptyset = \bigsqcap P \quad \text{and} \quad \top = \bigsqcap \emptyset = \bigsqcup P$$

if they exist.

Lower and Upper Bound Sets

Considering \sqcup and \sqcap partial functions $\sqcup, \sqcap : \mathcal{P}(P) \rightarrow P$ on the powerset of a partial order (P, \sqsubseteq) suggests introducing two further maps $LB, UB : \mathcal{P}(P) \rightarrow \mathcal{P}(P)$ on $\mathcal{P}(P)$:

Definition A.2.3.6 (Lower and Upper Bound Sets)

Let (P, \sqsubseteq) be a partial order. Then:

$LB, UB : \mathcal{P}(P) \rightarrow \mathcal{P}(P)$ denote two maps, which map a subset $Q \subseteq P$ to the set of its **lower bounds** and **upper bounds**, respectively:

1. $\forall Q \subseteq P. LB(Q) =_{df} \{lb \in P \mid lb \sqsubseteq Q\}$
2. $\forall Q \subseteq P. UB(Q) =_{df} \{ub \in P \mid Q \sqsubseteq ub\}$

Properties of Lower and Upper Bound Sets

Lemma A.2.3.7

Let (P, \sqsubseteq) be a partial order, and let $Q \subseteq P$. Then:

$$\bigsqcup Q = \bigsqcap UB(Q) \quad \text{and} \quad \bigsqcap Q = \bigsqcup LB(Q)$$

if the supremum and the infimum of Q exist.

Lemma A.2.3.8

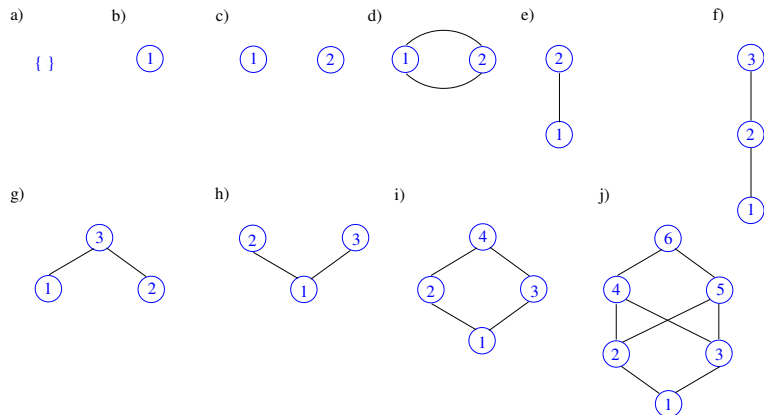
Let (P, \sqsubseteq) be a partial order, and let $Q, Q_1, Q_2 \subseteq P$. Then:

1. $Q_1 \subseteq Q_2 \Rightarrow LB(Q_1) \supseteq LB(Q_2) \wedge UB(Q_1) \supseteq UB(Q_2)$
2. $UB(LB(UB(Q))) = UB(Q)$
3. $LB(UB(LB(Q))) = LB(Q)$

Note: Lemma A.2.3.8(1) shows that LB and UB are antitonic maps (cf. Chapter A.2.7).

Exercise A.2.3.9

Which of the elements of the below diagrams are minimal, maximal, least or greatest?



Exercise A.2.3.10

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Write down the sets of elements of \mathbb{N}_0 , which are

1. minimal
2. maximal
3. least
4. greatest

wrt the divisibility relation $|$ on \mathbb{N}_0 .

A.2.4

Noetherian and Artinian Orders

Noetherian and Artinian Orders

Let (P, \sqsubseteq) be a partial order.

Definition A.2.4.1 (Noetherian Order)

(P, \sqsubseteq) is called a **Noetherian order**, if every non-empty subset $\emptyset \neq Q \subseteq P$ contains a minimal element.

Definition A.2.4.2 (Artinian Order)

(P, \sqsubseteq) is called an **Artinian order**, if the dual order (P, \supseteq) of (P, \sqsubseteq) is a Noetherian order.

Lemma A.2.4.3

(P, \sqsubseteq) is an **Artinian order** iff every non-empty subset $\emptyset \neq Q \subseteq P$ contains a maximal element.

Well-founded Orders

Let (P, \sqsubseteq) be a partial order.

Definition A.2.4.4 (Well-founded Order)

(P, \sqsubseteq) is called a **well-founded order**, if (P, \sqsubseteq) is a Noetherian order and totally ordered.

Lemma A.2.4.5

(P, \sqsubseteq) is a **well-founded order** iff every non-empty subset $\emptyset \neq Q \subseteq P$ contains a least element.

Noetherian Induction

Theorem A.2.4.6 (Noetherian Induction)

Let (N, \sqsubseteq) be a Noetherian order, let $N_{min} \subseteq N$ be the set of minimal elements of N , and let $\phi : N \rightarrow \mathbb{B}$ be a predicate on N . Then:

If

1. $\forall n \in N_{min}. \phi(n)$ (Induction base)
2. $\forall n \in N \setminus N_{min}. (\forall m \sqsubset n. \phi(m)) \Rightarrow \phi(n)$ (Induction step)

then:

$$\forall n \in N. \phi(n)$$

A.2.5

Chains

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1523/18

Chains, Antichains

Let (P, \sqsubseteq) be a partial order.

Definition A.2.5.1 (Chain)

A set $C \subseteq P$ is called a **chain**, if the elements of C are totally ordered, i.e., $\forall c_1, c_2 \in C. c_1 \sqsubseteq c_2 \vee c_2 \sqsubseteq c_1$.

Definition A.2.5.2 (Antichain)

A set $C \subseteq P$ is called an **antichain**, if $\forall c_1, c_2 \in C. c_1 \sqsubseteq c_2 \Rightarrow c_1 = c_2$.

Definition A.2.5.3 (Finite, Infinite (Anti-) Chain)

Let $C \subseteq P$ be a chain or an antichain. C is called **finite**, if the number of its elements is finite; C is called **infinite** otherwise.

Note: Any set P may be converted into an antichain by giving it the discrete order: $(P, =)$.

Ascending Chains, Descending Chains

Definition A.2.5.4 (Ascending, Descending Chain)

Let $C \subseteq P$ be a chain. C given in the form of

$$\blacktriangleright C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \dots\}$$

$$\blacktriangleright C = \{c_0 \supseteq c_1 \supseteq c_2 \supseteq \dots\}$$

is called an **ascending chain** and **descending chain**, respectively.

Examples of Chains

The set

- ▶ $S =_{df} \{n \in \mathbb{IN} \mid n \text{ even}\}$ is a chain in \mathbb{IN} .
- ▶ $S =_{df} \{z \in \mathbb{Z} \mid z \text{ odd}\}$ is a chain in \mathbb{Z} .
- ▶ $S =_{df} \{\{k \in \mathbb{IN} \mid k < n\} \mid n \in \mathbb{IN}\}$ is a chain in the powerset $\mathcal{P}(\mathbb{IN})$ of \mathbb{IN} .

Note: A chain can always be given in the form of an ascending or descending chain.

- ▶ $\{0 \leq 2 \leq 4 \leq 6 \leq \dots\}$: \mathbb{IN} as ascending chain.
- ▶ $\{\dots \geq 6 \geq 4 \geq 2 \geq 0\}$: \mathbb{IN} as descending chain.
- ▶ $\{\dots \leq -3 \leq -1 \leq 1 \leq 3 \leq \dots\}$: \mathbb{Z} as ascending chain.
- ▶ $\{\dots \geq 3 \geq 1 \geq -1 \geq -3 \geq \dots\}$: \mathbb{Z} as descending chain.
- ▶ ...

Eventually Stationary Sequences

Definition A.2.5.5 (Stationary Sequence)

1. An ascending sequence of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots$$

is called **eventually stationary**, if

$$\exists n \in \mathbb{N}. \forall j \in \mathbb{N}. p_{n+j} = p_n$$

2. A descending sequence of the form

$$p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

is called **eventually stationary**, if

$$\exists n \in \mathbb{N}. \forall j \in \mathbb{N}. p_{n+j} = p_n$$

Chains and Sequences

Lemma A.2.5.6

An ascending or descending sequence of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

1. is a finite chain iff it is eventually stationary.
2. is an infinite chain iff it is not eventually stationary.

Note the subtle difference between the notion of **chains** in terms of sets

$$\{p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots\} \quad \text{or} \quad \{p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots\}$$

and in terms of sequences

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

Sequences may contain **duplicates**, which would correspond to defining **chains** in terms of **multisets**.

Ascending, Descending Chain Condition

Let (P, \sqsubseteq) be a partial order.

Definition A.2.5.7 (Asc./Desc. Chain Condition)

(P, \sqsubseteq) satisfies the

1. **ascending chain condition** (in German: **aufsteigende Kettenbedingung**), if every ascending chain is eventually stationary, i.e., for every chain $p_1 \sqsubseteq p_2 \sqsubseteq \dots \sqsubseteq p_n \sqsubseteq \dots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.
2. **descending chain condition** (in German: **absteigende Kettenbedingung**), if every descending chain is eventually stationary, i.e., for every chain $p_1 \sqsupseteq p_2 \sqsupseteq \dots \sqsupseteq p_n \sqsupseteq \dots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.

Chains and Noetherian Orders

Let (P, \sqsubseteq) be a partial order.

Lemma A.2.5.8 (Noetherian Order)

The following statements are equivalent:

1. (P, \sqsubseteq) is a Noetherian order.
2. (P, \sqsubseteq) satisfies the descending chain condition.
3. Every chain of the form

$$p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

is eventually stationary, i.e.: $\exists n \in \mathbb{N}. \forall j \in \mathbb{N}. p_{n+j} = p_n$.

4. Every chain of the form

$$p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

is finite.

Chains and Artinian Orders

Let (P, \sqsubseteq) be a partial order.

Lemma A.2.5.9 (Artinian Order)

The following statements are equivalent:

1. (P, \sqsubseteq) is an Artinian order.
2. (P, \sqsubseteq) satisfies the ascending chain condition.
3. Every chain of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots$$

is eventually stationary, i.e.: $\exists n \in \mathbb{N}. \forall j \in \mathbb{N}. p_{n+j} = p_n$.

4. Every chain of the form

$$p_0 \sqsubset p_1 \sqsubset p_2 \sqsubset \dots$$

is finite.

Chains and Noetherian, Artinian Orders

Let (P, \subseteq) be a partial order.

Lemma A.2.5.10 (Noetherian and Artinian Order)

The following statements are equivalent:

1. (P, \subseteq) is a Noetherian and an Artinian order.
2. (P, \subseteq) satisfies the descending and the ascending chain condition.
3. Every chain $C \subseteq P$ is finite.

A.2.6

Directed Sets

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1533/18

Directed Sets

Let (P, \sqsubseteq) be a partial order, and let $\emptyset \neq D \subseteq P$.

Definition A.2.6.1 (Directed Set)

$D (\neq \emptyset)$ is called a **directed set** (in German: **gerichtete Menge**), if

$$\forall d, e \in D. \exists f \in D. f \in UB(\{d, e\})$$

i.e., for any two elements d and e there is a common upper bound of d and e in D , i.e., $UB(\{d, e\}) \cap D \neq \emptyset$.

Properties of Directed Sets

Let (P, \sqsubseteq) be a partial order, and let $D \subseteq P$.

Lemma A.2.6.2

D is a **directed set** iff any finite subset $D' \subseteq D$ has an upper bound in D , i.e., $\exists d \in D. d \in UB(D')$, i.e., $UB(D') \cap D \neq \emptyset$.

Lemma A.2.6.3

If D has a greatest element, then D is a directed set.

Properties of Finite Directed Sets

Let (P, \sqsubseteq) be a partial order, and let $D \subseteq P$.

Corollary A.2.6.4

Let D be a *finite directed set*. Then: $\bigsqcup D$ exists $\in D$ and is the greatest element of D .

Proof. Since D a directed set, we have:

$$\exists d \in D. d \in UB(D), \text{ i.e., } UB(D) \cap D \neq \emptyset.$$

This means $D \sqsubseteq d$. The antisymmetry of \sqsubseteq yields that the element enjoying this property is unique. Thus, d is the (unique) greatest element of D given by $\bigsqcup D$, i.e., $d = \bigsqcup D$.

Note: If D is infinite, the statement of [Corollary A.2.6.4](#) does usually not hold.

Strongly Directed Sets

Let (P, \sqsubseteq) be a partial order with least element \perp , and let $D \subseteq P$.

Definition A.2.6.5 (Strongly Directed Set)

$D \neq \emptyset$ is called a **strongly directed set** (in German: **stark gerichtete Menge**), if

1. $\perp \in D$
2. $\forall d, e \in D. \exists f \in D. f = \bigsqcup\{d, e\}$, i.e., for any two elements d and e the supremum $\bigsqcup\{d, e\}$ of d and e exists in D .

Properties of Strongly Directed Sets

Let (P, \sqsubseteq) be a partial order with least element \perp , and let $D \subseteq P$.

Lemma A.2.6.6

D is a strongly directed set iff every finite subset $D' \subseteq D$ has a supremum in D , i.e., $\exists d \in D. d = \bigsqcup D'$.

Lemma A.2.6.7

Let D be a finite strongly directed set. Then: $\bigsqcup D$ exists $\in D$ and is the greatest element of D .

Note: The statement of Lemma A.2.6.7 does usually not hold, if D is infinite.

Directed Sets, Strongly Directed Sets, Chains

Let (P, \sqsubseteq) be a partial order with least element \perp .

Lemma A.2.6.8

Let $\emptyset \neq D \subseteq P$ be a non-empty subset of P . Then:

1. D is a directed set, if D is a strongly directed set.
2. D is a strongly directed set, if $\perp \in D$ and D is a chain.

Corollary A.2.6.9

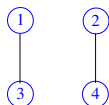
Let $\emptyset \neq D \subseteq P$ be a non-empty subset of P . Then:

$\perp \in D \wedge D$ chain $\Rightarrow D$ strongly directed set $\Rightarrow D$ directed set

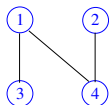
Exercise A.2.6.10

Which of the below partial orders are (strongly) directed sets?
Which of their subsets are (strongly) directed sets?

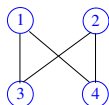
a)



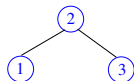
b)



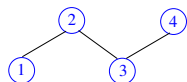
c)



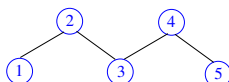
d)



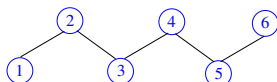
e)



f)



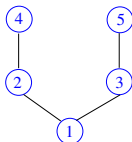
g)



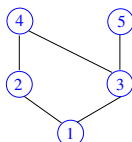
Exercise A.2.6.11

Which of the below **partial orders** are (strongly) directed sets?
Which of their **subsets** are (strongly) directed sets?

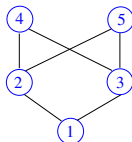
a)



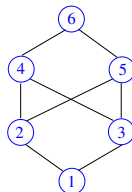
b)



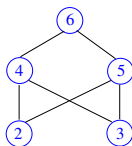
c)



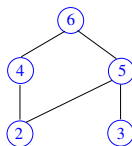
d)



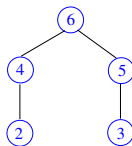
e)



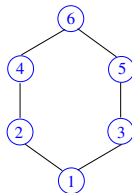
f)



g)



h)



Exercise A.2.6.12

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Is the set \mathbb{N}_0

1. directed?
2. strongly directed?

What subsets of \mathbb{N}_0 are

1. directed?
2. strongly directed?

Proof or counterexample.

A.2.7

Maps on Partial Orders

Monotonic and Antitonic Maps on POs

Let (C, \sqsubseteq_C) and (D, \sqsubseteq_D) be partial orders, and let $f \in [C \rightarrow D]$ be a map from C to D .

Definition A.2.7.1 (Monotonic Maps on POs)

f is called **monotonic** (or **order preserving**) iff

$$\forall c, c' \in C. c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$$

(Preservation of the ordering of elements)

Definition A.2.7.2 (Antitonic Maps on POs)

f is called **antitonic** (or **order inverting**) iff

$$\forall c, c' \in C. c \sqsubseteq_C c' \Rightarrow f(c') \sqsupseteq_D f(c)$$

(Inversion of the ordering of elements)

Expanding and Contracting Maps on POs

Let (C, \sqsubseteq_C) be a partial order, let $f \in [C \rightarrow C]$ be a map on C , and let $\hat{c} \in C$ be an element of C .

Definition A.2.7.3 (Expanding Maps on POs)

f is called

- ▶ **expanding** (or **inflationary**) for \hat{c} iff $\hat{c} \sqsubseteq f(\hat{c})$
- ▶ **expanding** (or **inflationary**) iff $\forall c \in C. c \sqsubseteq f(c)$

Definition A.2.7.4 (Contracting Maps on POs)

f is called

- ▶ **contracting** (or **deflationary**) for \hat{c} iff $f(\hat{c}) \sqsubseteq \hat{c}$
- ▶ **contracting** (or **deflationary**) iff $\forall c \in C. f(c) \sqsubseteq c$

A.2.8

Order Homomorphisms and Order Isomorphisms

PO Homomorphisms, PO Isomorphisms

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be partial orders, and let $f \in [P \rightarrow R]$ be a map from P to R .

Definition A.2.8.1 (PO Hom. & Isomorphism)

f is called an

1. **order homomorphism** between P and R , if f is monotonic (or order preserving), i.e.,

$$\forall p, q \in P. p \sqsubseteq_P q \Rightarrow f(p) \sqsubseteq_R f(q)$$

2. **order isomorphism** between P and R , if f is a bijective order homomorphism between P and R and the inverse f^{-1} of f is an order homomorphism between R and P .

Definition A.2.8.2 (Order Isomorphic)

(P, \sqsubseteq_P) and (R, \sqsubseteq_R) are called **order isomorphic**, if there is an order isomorphism between P and R .

PO Embeddings

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be partial orders, and let $f \in [P \rightarrow R]$ be a map from P to R .

Definition A.2.8.3 (PO Embedding)

f is called an **order embedding** of P in R iff

$$\forall p, q \in P. p \sqsubseteq_P q \iff f(p) \sqsubseteq_R f(q)$$

Lemma A.2.8.4 (PO Embeddings and Isomorphisms)

f is an order isomorphism between P and R iff f is an order embedding of P in R and f is surjective.

Intuitively: Partial orders, which are order isomorphic, are ‘essentially the same.’

A.3

Complete Partially Ordered Sets

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1549/18

A.3.1

Chain and Directly Complete Partial Orders

Complete Partially Ordered Sets

...or **Complete Partial Orders**:

- ▶ a slightly weaker ordering notion than that of a lattice (cf. [Appendix A.4](#)), which is often more adequate for the modelling of problems in computer science, where full lattice properties are often not required.
- ▶ come in two different flavours as so-called
 - ▶ Chain Complete Partial Orders (CCPOs)
 - ▶ Directedly Complete Partial Orders (DCPOs)based on the notions of **chains** and **directed sets**, respectively, which, however, are equivalent (cf. [Theorem 3.1.7](#)).

Complete Partial Orders: CCPO View

Definition A.3.1.1 (Chain Complete Partial Order)

A partial order (P, \sqsubseteq) is a

1. **chain complete partial order (pre-CCPO)**, if every non-empty (ascending) chain $\emptyset \neq C \subseteq P$ has a least upper bound $\bigsqcup C$ in P , i.e., $\bigsqcup C \text{ exists} \in P$.
2. **pointed chain complete partial order (CCPO)**, if every (ascending) chain $C \subseteq P$ has a least upper bound $\bigsqcup C$ in P , i.e., $\bigsqcup C \text{ exists} \in P$.

Note: Some authors use **CCPO** and **CCPPO** instead of **pre-CCPO** and **CCPO**, respectively.

Complete Partial Orders: DCPO View

Definition A.3.1.2 (Directedly Complete Partial Ord.)

A partial order (P, \sqsubseteq) is a

1. **directedly complete partial order (pre-DCPO)**, if every directed subset $D \subseteq P$ has a least upper bound $\bigsqcup D$ in P , i.e., $\bigsqcup D \text{ exists} \in P$.
2. **pointed directedly complete partial order (DCPO)**, if it is a **pre-DCPO** and has a least element \perp .

Note: Some authors use **DCPO** and **DCPPO** instead of **pre-DCPO** and **DCPO**, respectively.

Remarks on CCPOs and DCPOs

On **CCPOs**:

- ▶ A **CCPO** is often called a **domain**.
- ▶ ‘**Ascending chain**’ and ‘**chain**’ can equivalently be used in **Definition A.3.1.1**, since a chain can always be given in ascending order. ‘**Ascending**’ chain is just more intuitive.

On **DCPOs**:

- ▶ A **directed set** S , in which by definition every finite subset has an upper bound in S , does not need to have a supremum in S , if S is infinite. Therefore, the **DCPO property does not trivially follow** from the directed set property (cf. **Corollary A.2.6.4**).

Existence of Least Elements in CPOs

Lemma A.3.1.3 (Least Elem. Existence in CPOs)

Let (C, \sqsubseteq) be a CPO, i.e., a CCPO or DCPO. Then there is a unique least element in C , denoted by \perp , which is given by the supremum of the empty chain or set, i.e.: $\perp = \bigsqcup \emptyset$.

Corollary A.3.1.4 (Non-Emptiness of CPOs)

Let (C, \sqsubseteq) be a CPO, i.e., a CCPO or DCPO. Then: $C \neq \emptyset$.

Note: Lemma A.3.1.3 does not hold for pre-CPOs, i.e., a pre-CPO (P, \sqsubseteq) does not need to have a least element.

Relating Finite POs, CCPOs and DCPOs

Let P be a finite set, and let \sqsubseteq be a relation on P .

Lemma A.3.1.5 (Fin. POs, pre-CCPOs, pre-DCPOs)

The following statements are equivalent:

1. (P, \sqsubseteq) is a partial order.
2. (P, \sqsubseteq) is a pre-CCPO.
3. (P, \sqsubseteq) is a pre-DCPO.

Lemma A.3.1.6 (Finite POs, CCPOs, DCPOs)

Let $p \in P$ with $p \sqsubseteq P$. Then the following statements are equivalent:

1. (P, \sqsubseteq) is a partial order.
2. (P, \sqsubseteq) is a CCPO.
3. (P, \sqsubseteq) is a DCPO.

Equivalence of CCPOs and DCPOs

Theorem A.3.1.7 (Equivalence)

Let (P, \sqsubseteq) be a partial order. Then the following statements are equivalent:

1. (P, \sqsubseteq) is a CCPO.
2. (P, \sqsubseteq) is a DCPO.

Note: We simply speak of a CPO, if its flavour based on chains (CCPO) or directed sets (DCPO) does not matter; analogously, this applies to pre-CPOs.

Examples of pre-CPOs and CPOs (1)

- ▶ $(\mathcal{P}(\mathbb{N}), \subseteq)$ is a CPO (i.e., a CCPO and a DCPO).
 - ▶ Least element: \emptyset
 - ▶ Least upper bound $\bigsqcup C$ of C chain $C \subseteq \mathcal{P}(\mathbb{N})$: $\bigcup_{C' \in C} C'$
- ▶ The set of finite and infinite strings S partially ordered by the prefix relation $\sqsubseteq_{\text{pref}}$ defined by

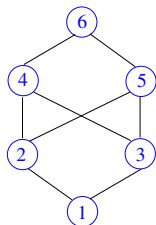
$$\forall s, s'' \in S. s \sqsubseteq_{\text{pref}} s'' \iff_{df} s = s'' \vee (s \text{ finite} \wedge \exists s' \in S. s ++ s' = s'')$$

is a CPO.

- ▶ $(\{-n \mid n \in \mathbb{N}\}, \leq)$ is a pre-CPO (i.e., a pre-CCPO and a pre-DCPO) but not a CPO (i.e., not a CCPO and DCPO).

Examples of pre-CPOs and CPOs (2)

- ▶ (\emptyset, \emptyset) is a pre-CPO (i.e., a pre-CCPO and a pre-DCPO) but not a CPO (i.e., not a CCPO and DCPO).
(Both the pre-CCPO (absence of non-empty chains in \emptyset) and the pre-DCPO (\emptyset is the only subset of \emptyset and is not directed by definition) property holds trivially. Note also that $P = \emptyset$ implies $\sqsubseteq = \emptyset \subseteq P \times P$).
- ▶ The partial order (P, \sqsubseteq) given by the below Hasse diagram is a CPO.



Examples of pre-CPOs and CPOs (3)

- ▶ The set of finite and infinite strings S partially ordered by the lexicographical order \sqsubseteq_{lex} defined by

$$\forall s, t \in S. s \sqsubseteq_{lex} t \iff_{df}$$

$$s = t \vee (\exists p \text{ finite}, s', t' \in S. s = p ++ s' \wedge t = p ++ t' \wedge (s' = \varepsilon \vee s'_1 < t'_1))$$

where ε denotes the empty string, $w \downarrow_1$ denotes the first character of a string w , and $<$ the lexicographical ordering on characters, is a CPO (i.e., a CCPO and a DCPO).

(Anti-) Examples of CPOs

- ▶ (\mathbb{N}, \leq) is not a CPO (i.e., not a CCPO and DCPO).

- ▶ The set of finite strings S_{fin} partially ordered by the

- ▶ prefix relation \sqsubseteq_{pref} defined by

$$\forall s, s' \in S_{fin}. s \sqsubseteq_{pref} s' \iff_{df} \exists s'' \in S_{fin}. s ++ s'' = s'$$

is not a CPO (i.e., not a CCPO and DCPO).

- ▶ lexicographical order \sqsubseteq_{lex} defined by

$$\forall s, t \in S_{fin}. s \sqsubseteq_{lex} t \iff_{df}$$

$$\exists p, s', t' \in S_{fin}. s = p ++ s' \wedge t = p ++ t' \wedge (s' = \varepsilon \vee s' \downarrow_1 < t' \downarrow_1)$$

where ε denotes the empty string, $w \downarrow_1$ denotes the first character of a string w , and $<$ the lexicographical ordering on characters, is not a CPO (i.e., not a CCPO and DCPO).

- ▶ $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is not a CPO (i.e., not a CCPO and DCPO).

Exercise A.3.1.8

Which of the **partial orders** given by the below **Hasse diagrams** are **(pre-) CCPOs**? Which ones are **(pre-) DCPOs**?

a)

{}

b)



c)



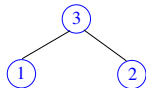
d)



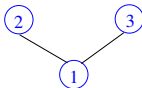
e)



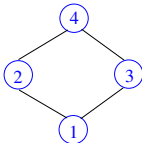
f)



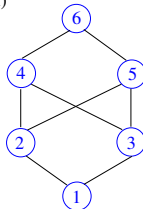
g)



h)



i)



Strongly Directed CPOs: A DCPO Variant

On DCPOs based on Strongly Directed Sets

- ▶ Replacing directed sets by strongly directed sets in Definition A.3.1.2 leads to SDCPOs.
- ▶ Recalling that strongly directed sets are not empty (cf. Definition A.2.6.5), there is no analogue of pre-DCPOs for strongly directed sets.
- ▶ A strongly directed set S , in which by definition every finite subset has a supremum in S , does not need to have a supremum itself in S , if S is infinite. Therefore, the SDCPO property does not trivially follow from the strongly directed property of sets (cf. Corollary A.2.6.4).

Exercise A.3.1.9

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or disprove: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. pre-CCPO
2. CCPO
3. pre-DCPO
4. DCPO
5. SDCPO

Proof or counterexample.

A.3.2

Maps on Complete Partial Orders

Continuous Maps on CCPOs

Let (C, \sqsubseteq_C) and (D, \sqsubseteq_D) be CCPOs, and let $f \in [C \rightarrow D]$ be a map from C to D .

Definition A.3.2.1 (Continuous Maps on CCPOs)

f is called **continuous** iff f is monotonic and

$$\forall C' \neq \emptyset \text{ chain} \subseteq C. f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$$

(Preservation of least upper bounds)

Note: $\forall S \subseteq C. f(S) =_{df} \{f(s) \mid s \in S\}$

Continuous Maps on DCPOs

Let (D, \sqsubseteq_D) and (E, \sqsubseteq_E) be DCPOs, and let $f \in [D \rightarrow E]$ be a map from D to E .

Definition A.3.2.2 (Continuous Maps on DCPOs)

f is called **continuous** iff

$$\forall D' \neq \emptyset \text{ directed set } \subseteq D. f(D') \text{ directed set } \subseteq E \wedge \\ f(\bigsqcup_D D') =_E \bigsqcup_E f(D')$$

(Preservation of least upper bounds)

Note: $\forall S \subseteq D. f(S) =_{df} \{ f(s) \mid s \in S \}$

Characterizing Monotonicity

Let $(C, \sqsubseteq_C), (D, \sqsubseteq_D)$ be CCPOs, let $(E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be DCPOs.

Lemma A.3.2.3 (Characterizing Monotonicity)

1. $f : C \rightarrow D$ is monotonic

iff $\forall C' \neq \emptyset$ *chain* $\subseteq C$.

$$f(C') \text{ *chain* } \subseteq D \wedge f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$$

2. $g : E \rightarrow F$ is monotonic

iff $\forall E' \neq \emptyset$ *directed set* $\subseteq E$.

$$g(E') \text{ *directed set* } \subseteq F \wedge g(\bigsqcup_E E') \sqsupseteq_F \bigsqcup_F g(E')$$

Strict Maps on CCPOs and DCPOs

Let (C, \sqsubseteq_C) , (D, \sqsubseteq_D) be CCPOs with least elements \perp_C and \perp_D , respectively, let (E, \sqsubseteq_E) , (F, \sqsubseteq_F) be DCPOs with least elements \perp_E and \perp_F , respectively, and let $f \in [C \xrightarrow{\text{con}} D]$ and $g \in [E \xrightarrow{\text{con}} F]$ be continuous maps.

Definition A.3.2.4 (Strict Functions on CPOs)

f and g are called **strict**, if the equalities

$$\blacktriangleright f(\bigsqcup_C C') =_D \bigsqcup_D f(C'), \quad g(\bigsqcup_E E') =_F \bigsqcup_F g(E')$$

also hold for $C' = \emptyset$ and $E' = \emptyset$, i.e., if the equalities

$$\blacktriangleright f(\bigsqcup_C \emptyset) =_C f(\perp_C) =_D \perp_D =_D \bigsqcup \emptyset$$

$$\blacktriangleright f(\bigsqcup_E \emptyset) =_E g(\perp_E) =_F \perp_F =_F \bigsqcup \emptyset$$

are valid.

A.3.3

Mechanisms for Constructing Complete Partial Orders

Common CCPO and DCPO Constructions

The following construction principles hold for

- ▶ CCPOs
- ▶ DCPOs

Therefore, we simply write CPO.

Common CPO Constructions: Flat CPOs

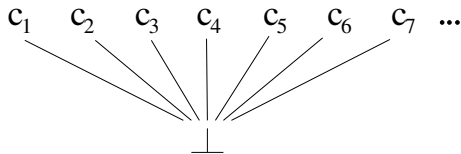
Lemma A.3.3.1 (Flat CPO Construction)

Let C be a set. Then:

$(C \dot{\cup} \{\perp\}, \sqsubseteq_{flat})$ with \sqsubseteq_{flat} defined by

$$\forall c, d \in C \dot{\cup} \{\perp\}. c \sqsubseteq_{flat} d \iff_{df} c = \perp \vee c = d$$

is a CPO, a so-called flat CPO.



Common CPO Constructions: Flat pre-CPOs

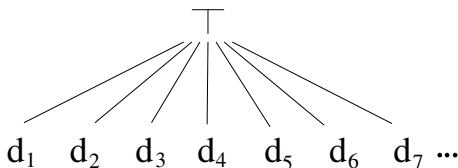
Lemma A.3.3.2 (Flat Pre-CPO Construction)

Let D be a set. Then:

$(D \dot{\cup} \{T\}, \sqsubseteq_{flat})$ with \sqsubseteq_{flat} defined by

$$\forall d, e \in D \dot{\cup} \{T\}. d \sqsubseteq_{flat} e \iff_{df} e = T \vee d = e$$

is a pre-CPO, a so-called flat pre-CPO.



Common CPO Constructions: Products (1)

Lemma A.3.3.3 (Non-strict Product Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **non-strict product** $(\times P_i, \sqsubseteq_\times)$, where

▶ $\times P_i =_{df} P_1 \times P_2 \times \dots \times P_n$ is the cartesian product of all P_i , $1 \leq i \leq n$

▶ \sqsubseteq_\times is defined pointwise by

$$\forall (p_1, \dots, p_n), (q_1, \dots, q_n) \in \times P_i.$$

$$(p_1, \dots, p_n) \sqsubseteq_\times (q_1, \dots, q_n) \iff_{df}$$

$$\forall i \in \{1, \dots, n\}. p_i \sqsubseteq_i q_i$$

is a CPO.

Common CPO Constructions: Products (2)

Lemma A.3.3.4 (Strict Product Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **strict** (or **smash**) **product** $(\bigotimes P_i, \sqsubseteq_{\otimes})$, where

- ▶ $\bigotimes P_i =_{df} \times P_i$ is the the cartesian product of all P_i
- ▶ $\sqsubseteq_{\otimes} =_{df} \sqsubseteq_{\times}$ defined pointwise with the additional setting
$$(p_1, \dots, p_n) = \perp \iff_{df} \exists i \in \{1, \dots, n\}. p_i = \perp_i$$

is a CPO.

Common CPO Constructions: Sums (1)

Lemma A.3.3.5 (Separated Sum Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **separated** (or **direct**) **sum** $(\bigoplus_{\perp} P_i, \sqsubseteq_{\bigoplus_{\perp}})$, where

▶ $\bigoplus_{\perp} P_i =_{df} P_1 \dot{\cup} P_2 \dot{\cup} \dots \dot{\cup} P_n \dot{\cup} \{\perp\}$ is the disjoint union of all P_i , $1 \leq i \leq n$, and a fresh bottom element \perp

▶ $\sqsubseteq_{\bigoplus_{\perp}}$ is defined by

$$\forall p, q \in \bigoplus_{\perp} P_i. p \sqsubseteq_{\bigoplus_{\perp}} q \iff_{df}$$

$$p = \perp \vee (\exists i \in \{1, \dots, n\}. p, q \in P_i \wedge p \sqsubseteq_i q)$$

is a CPO.

Common CPO Constructions: Sums (2)

Lemma A.3.3.6 (Coalesced Sum Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **coalesced sum** $(\bigoplus_{\vee} P_i, \sqsubseteq_{\bigoplus_{\vee}})$, where

- ▶ $\bigoplus_{\vee} P_i =_{df} P_1 \setminus \{\perp_1\} \dot{\cup} P_2 \setminus \{\perp_2\} \dot{\cup} \dots \dot{\cup} P_n \setminus \{\perp_n\} \dot{\cup} \{\perp\}$ is the disjoint union of all P_i , $1 \leq i \leq n$, and a fresh bottom element \perp , which is identified with and replaces the least elements \perp_i of the sets P_i , i.e., $\perp =_{df} \perp_i$, $i \in \{1, \dots, n\}$

- ▶ $\sqsubseteq_{\bigoplus_{\vee}}$ is defined by

$$\forall p, q \in \bigoplus_{\vee} P_i. p \sqsubseteq_{\bigoplus_{\vee}} q \iff_{df}$$

$$p = \perp \vee (\exists i \in \{1, \dots, n\}. p, q \in P_i \wedge p \sqsubseteq_i q)$$

is a CPO.

Common CPO Constructions: Function Space

Lemma A.3.3.7 (Continuous Function Space Con.)

Let (C, \sqsubseteq_C) and (D, \sqsubseteq_D) be pre-CPOs. Then:

The **continuous function space** $([C \xrightarrow{\text{con}} D], \sqsubseteq_{\text{cfs}})$, where

▶ $[C \xrightarrow{\text{con}} D]$ is the set of continuous maps from C to D

▶ \sqsubseteq_{cfs} is defined pointwise by

$$\forall f, g \in [C \xrightarrow{\text{con}} D]. f \sqsubseteq_{\text{cfs}} g \iff_{df} \forall c \in C. f(c) \sqsubseteq_D g(c)$$

is a **pre-CPO**. It is a **CPO**, if (D, \sqsubseteq_D) is a **CPO**.

Note: The definition of \sqsubseteq_{cfs} does not make use of C being a pre-CPO. This requirement is only to allow us tailoring the definition to continuous maps.

A.4

Lattices

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1579/18

A.4.1

Lattices, Complete Lattices

Lattices and Complete Lattices

Let (P, \sqsubseteq) be a partial order, $P \neq \emptyset$.

Definition A.4.1.1 (Lattice)

(P, \sqsubseteq) is a **lattice** (in German: **Verband**), if every **non-empty finite** subset P' of P has a least upper bound and a greatest lower bound in P .

Definition A.4.1.2 (Complete Lattice)

(P, \sqsubseteq) is a **complete lattice** (in German: **vollständiger Verband**), if **every** subset P' of P has a least upper bound and a greatest lower bound in P .

Note: Lattices and complete lattices are special partial orders.

Properties of Complete Lattices

Lemma A.4.1.3 (Existence of Extremal Elements)

Let (P, \sqsubseteq) be a complete lattice. Then there is

1. a least element in P , denoted by \perp , satisfying:
$$\perp = \bigsqcup \emptyset = \bigsqcap P.$$
2. a greatest element in P , denoted by \top , satisfying:
$$\top = \bigsqcap \emptyset = \bigsqcup P.$$

Lemma A.4.1.4 (Characterization Lemma)

Let (P, \sqsubseteq) be a partial order. Then the following statements are equivalent:

1. (P, \sqsubseteq) is a complete lattice.
2. Every subset of P has a least upper bound.
3. Every subset of P has a greatest lower bound.

Properties of Finite Lattices

Lemma A.4.1.5 (Finiteness implies Completeness)

If (P, \sqsubseteq) is a finite lattice, then (P, \sqsubseteq) is a complete lattice.

Corollary A.4.1.6 (Finiteness impl. Ex. of ext. Elem.)

If (P, \sqsubseteq) is a finite lattice, then (P, \sqsubseteq) has a least element and a greatest element.

Complete Semi-Lattices

Let (P, \sqsubseteq) be a partial order, $P \neq \emptyset$.

Definition A.4.1.7 (Complete Semi-Lattice)

(P, \sqsubseteq) is a **complete**

1. **join semi-lattice** (in German: **Vereinigungshalbverband**) iff $\forall \emptyset \neq S \subseteq P. \bigsqcup S \text{ exists } \in P$.
2. **meet semi-lattice** (in German: **Schnitthalbverband**) iff $\forall \emptyset \neq S \subseteq P. \bigsqcap S \text{ exists } \in P$.

Properties of Complete Semi-Lattices (1)

Proposition A.4.1.8 (Extr. Bounds in C. Semi-Lat.)

If (P, \sqsubseteq) is a complete

1. join semi-lattice, then $\bigsqcup P$ exists $\in P$ (whereas $\bigsqcup \emptyset (\hat{=} \perp)$ does usually not exist in P).
2. meet semi-lattice, then $\bigsqcap P$ exists $\in P$ (whereas $\bigsqcap \emptyset (\hat{=} \top)$ does usually not exist in P).

Informally: Least elements need not exist in complete join semi-lattices, greatest elements need not exist in complete meet semi-lattices.

Properties of Complete Semi-Lattices (2)

Lemma A.4.1.9 (Ex. great. El. in C. Join Semi-Lat.)

Let (P, \sqsubseteq) be a complete join semi-lattice. Then:

$\bigsqcup P$ exists $\in P$ and is the (unique) greatest element in P that is usually denoted by \top , i.e., $\top = \bigsqcup P$.

Lemma A.4.1.10 (Ex. least El. in C. Meet Semi-Lat.)

Let (P, \sqsubseteq) be a complete meet semi-lattice. Then:

$\bigsqcap P$ exists $\in P$ and is the (unique) least element in P that is usually denoted by \perp , i.e., $\perp = \bigsqcap P$.

Characterizing Upper and Lower Bounds (1)

...in complete semi-lattices.

Lemma A.4.1.11 (Char. u./l. Bounds in C. Semi-L.)

1. Let (P, \sqsubseteq) be a complete join semi-lattice, and let $Q \subseteq P$ be a subset of P .

If there is a lower bound for Q in P , i.e, if

$\{p \in P \mid p \sqsubseteq Q\} \neq \emptyset$, then $\bigcap Q$ exists $\in P$ satisfying

$$\bigcap Q = \bigsqcup \{p \in P \mid p \sqsubseteq Q\}$$

2. Let (P, \sqsubseteq) be a complete meet semi-lattice, and let $Q \subseteq P$ be a subset of P .

If there is an upper bound for Q in P , i.e, if

$\{p \in P \mid Q \sqsubseteq p\} \neq \emptyset$, then $\bigsqcup Q$ exists $\in P$ satisfying

$$\bigsqcup Q = \bigcap \{p \in P \mid Q \sqsubseteq p\}$$

Characterizing Upper and Lower Bounds (2)

Lemma A.4.1.12 (L./gr. Elements in C. Semi-L.)

If (P, \sqsubseteq) is a complete

1. join semi-lattice and $\bigsqcup \emptyset$ exists $\in P$, then $\bigsqcup \emptyset$ is the (unique) least element in P , denoted by \perp , i.e., $\perp = \bigsqcup \emptyset$.
2. meet semi-lattice and $\bigsqcap \emptyset$ exists $\in P$, then $\bigsqcap \emptyset$ is the (unique) greatest element in P , denoted by \top , i.e., $\top = \bigsqcap \emptyset$.

Relating Complete Semi-Lattices and Lattices

Lemma A.4.1.13 (Complete Semi-Lattices&Lattices)

If (P, \sqsubseteq) is a complete

1. join semi-lattice and $\bigsqcup \emptyset$ exists $\in P$
2. meet semi-lattice and $\bigsqcap \emptyset$ exists $\in P$

then (P, \sqsubseteq) is a complete lattice.

Exercise A.4.1.14

Prove or disprove:

If (P, \sqsubseteq) is a complete lattice, then

1. $(P \setminus \{\perp\}, \sqsubseteq_{\setminus \perp})$ is a complete join semi-lattice.
2. $(P \setminus \{\top\}, \sqsubseteq_{\setminus \top})$ is a complete meet semi-lattice.

where $\sqsubseteq_{\setminus \perp}$ and $\sqsubseteq_{\setminus \top}$ denote the restrictions of \sqsubseteq from P to $P \setminus \{\perp\}$ and $P \setminus \{\top\}$, respectively. Proof or counterexample.

Relating Lattices and Complete Partial Orders

Lemma A.4.1.15 (Complete Lattices and CPOs)

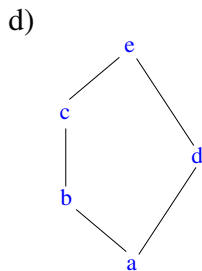
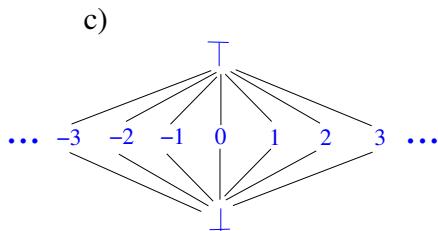
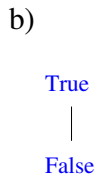
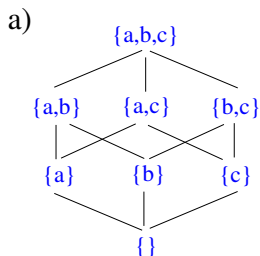
If (P, \sqsubseteq) is a complete lattice, then (P, \sqsubseteq) is a CPO (i.e., a CCPO and DCPO).

Corollary A.4.1.16 (Finite Lattices and CPOs)

If (P, \sqsubseteq) is a finite lattice, then (P, \sqsubseteq) is a CPO (i.e., a CCPO and DCPO).

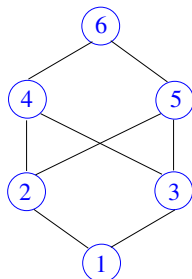
Note: Lemma A.4.1.15 does not hold for lattices.

Examples of Complete Lattices



(Anti-) Examples

- ▶ The partial order (P, \sqsubseteq) given by the below Hasse diagram is not a lattice (whereas it is a CPO).



- ▶ $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is not a complete lattice (and not a CPO).

Exercise A.4.1.17

Which of the **partial orders** given by the below **Hasse diagrams** are **lattices**? Which ones are **complete lattices**?

a)

{ }

b)



c)



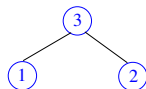
d)



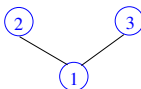
e)



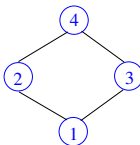
f)



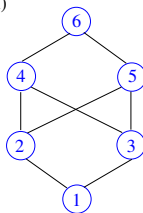
g)



h)



i)



Exercise A.4.1.18

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or disprove: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. lattice
2. complete lattice
3. complete join semi-lattice
4. complete meet semi-lattice

Proof or counterexample.

Summary, Overview

Corollary A.4.1.19

Let $P \neq \emptyset$ be a non-empty set, and \sqsubseteq a relation on P . Then:

(P, \sqsubseteq) finite lattice (L. A.4.1.5) \vee

(P, \sqsubseteq) complete join semi-lattice and

$\bigsqcup \emptyset \text{ exists } \in P$ (L. A.4.1.13(1)) \vee

(P, \sqsubseteq) complete meet semi-lattice and

$\bigsqcap \emptyset \text{ exists } \in P$ (L. A.4.1.13(2))

$\Rightarrow (P, \sqsubseteq)$ complete lattice

(D. A.4.1.2 and

L. A.4.1.14) $\Rightarrow (P, \sqsubseteq)$ lattice and complete partial order

(D. A.4.1.1 and

D. A.3.1.1/2) $\Rightarrow (P, \sqsubseteq)$ partial order

(D. A.2.1.2) $\Rightarrow (P, \sqsubseteq)$ pre-order

Exercise A.4.1.20

Let

$QO, PO, \mathcal{L}, CPO, \mathcal{CL}, \mathcal{FL}, CJSL, CJSL_{\perp}, CMSL, CMSL^{\top}$

denote the sets of all quasi-orders QO , partial orders PO , lattices \mathcal{L} , complete partial orders CPO , complete lattices \mathcal{CL} , finite lattices \mathcal{FL} , complete join semi-lattices without/with least element $CJSL/CJSL_{\perp}$, and meet semi-lattices without/with greatest element $CMSL/CMSL^{\top}$.

1. What further implications or equivalences hold in addition to those listed in [Corollary A.4.1.19](#)? (Proof or counterexample)
2. What inclusions or (set) equalities hold among QO, PO, \mathcal{L} , etc.? (Proof or counterexample)

A.4.2

Distributive, Additive Maps on Lattices

Distributive, Additive Maps on Lattices

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a map on P .

Definition A.4.2.1 (Distributive, Additive Map)

f is called

- ▶ **distributive** (or \sqcap -continuous) iff

$$\forall \emptyset \neq P' \subseteq P. f(\sqcap P') = \sqcap f(P')$$

(Preservation of greatest lower bounds)

- ▶ **additive** (or \sqcup -continuous) iff

$$\forall \emptyset \neq P' \subseteq P. f(\sqcup P') = \sqcup f(P')$$

(Preservation of least upper bounds)

Note: $\forall S \subseteq P. f(S) =_{df} \{ f(s) \mid s \in S \}$

Characterizing Monotonicity

...in terms of the preservation of greatest lower and least upper bounds:

Lemma A.4.2.2 (Characterizing Monotonicity)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a map on P . Then:

$$\begin{aligned} f \text{ is monotonic} &\iff \forall P' \subseteq P. f(\bigsqcap P') \sqsubseteq \bigsqcap f(P') \\ &\iff \forall P' \subseteq P. f(\bigsqcup P') \supseteq \bigsqcup f(P') \end{aligned}$$

Note: $\forall S \subseteq P. f(S) =_{df} \{ f(s) \mid s \in S \}$

Useful Results on Mon., Distr., and Additivity

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a map on P .

Lemma A.4.2.3

f is distributive iff f is additive.

Lemma A.4.2.4

f is monotonic, if f is distributive (or additive).
(i.e., distributivity (or additivity) implies monotonicity)

A.4.3

Lattice Homomorphisms, Lattice Isomorphisms

Lattice Homomorphisms, Lattice Isomorphisms

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be two lattices, and let $f \in [P \rightarrow R]$ be a map from P to R .

Definition A.4.3.1 (Lattice Homomorphism)

f is called a **lattice homomorphism**, if

$$\forall p, q \in P. f(p \sqcup_P q) = f(p) \sqcup_Q f(q) \wedge \\ f(p \sqcap_P q) = f(p) \sqcap_Q f(q)$$

Definition A.4.3.2 (Lattice Isomorphism)

1. f is called a **lattice isomorphism**, if f is a lattice homomorphism and bijective.
2. (P, \sqsubseteq_P) and (R, \sqsubseteq_R) are called **isomorphic**, if there is lattice isomorphism between P and R .

Useful Results (1)

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be two lattices, and let $f \in [P \rightarrow R]$ be a map from P to R .

Lemma A.4.3.3

$$f \in [P \xrightarrow{hom} R] \Rightarrow f \in [P \xrightarrow{mon} R]$$

The reverse implication of [Lemma A.4.3.3](#) does not hold, however, the following weaker relation holds:

Lemma A.4.3.4

$$\begin{aligned} f \in [P \xrightarrow{mon} R] \Rightarrow \\ \forall p, q \in P. f(p \sqcup_P q) \sqsupseteq_Q f(p) \sqcup_Q f(q) \wedge \\ f(p \sqcap_P q) \sqsubseteq_Q f(p) \sqcap_Q f(q) \end{aligned}$$

Useful Results (2)

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be two lattices, and let $f \in [P \rightarrow R]$ be a map from P to R .

Lemma A.4.3.5

$$f \in [P \xrightarrow{iso} R] \Rightarrow f^{-1} \in [R \xrightarrow{iso} P]$$

Lemma A.4.3.6

$$f \in [P \xrightarrow{iso} R] \iff f \in [P \xrightarrow{po-hom} R] \text{ wrt } \sqsubseteq_P \text{ and } \sqsubseteq_Q$$

A.4.4

Modular, Distributive, and Boolean Lattices

Modular Lattices

Let (P, \sqsubseteq) be a lattice with meet operation \sqcap and join operation \sqcup .

Lemma A.4.4.1

$$\forall p, q, r \in P. p \sqsubseteq r \Rightarrow p \sqcup (q \sqcap r) \sqsubseteq (p \sqcup q) \sqcap r$$

Definition A.4.4.2 (Modular Lattice)

(P, \sqsubseteq) is called **modular**, if

$$\forall p, q, r \in P. p \sqsubseteq r \Rightarrow p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap r$$

Characterizing Modular Lattices

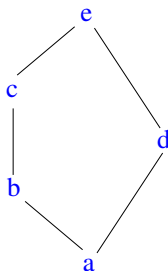
Theorem A.4.4.3 (Characterizing Modular Lattices)

A lattice (P, \sqsubseteq) is

1. **modular** iff

$$\forall p, q, r \in P. p \sqsubseteq q, p \sqcap r = q \sqcap r, p \sqcup r = q \sqcup r \Rightarrow p = q$$

2. **not modular** iff (P, \sqsubseteq) contains a sublattice, which is isomorphic to the lattice:



Distributive Lattices

Let (P, \sqsubseteq) be a lattice with meet operation \sqcap and join operation \sqcup .

Lemma A.4.4.4

1. $\forall p, q, r \in P. p \sqcup (q \sqcap r) \sqsubseteq (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall p, q, r \in P. p \sqcap (q \sqcup r) \sqsupseteq (p \sqcap q) \sqcup (p \sqcap r)$

Definition A.4.4.5 (Distributive Lattice)

(P, \sqsubseteq) is called **distributive**, if

1. $\forall p, q, r \in P. p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall p, q, r \in P. p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$

Towards Characterizing Distributive Lattices

Lemma A.4.4.6

The following two statements are equivalent:

1. $\forall p, q, r \in P. p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall p, q, r \in P. p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$

Hence, it is sufficient to require the validity of [property \(1\)](#) or of [property \(2\)](#) in [Definition A.4.4.5](#).

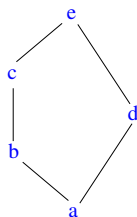
Characterizing Distributive Lattices

Let (P, \sqsubseteq) be a lattice.

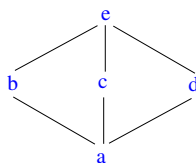
Theorem A.4.4.7 (Characterizing Distributive Lat.)

(P, \sqsubseteq) is not distributive iff (P, \sqsubseteq) contains a sublattice that is isomorphic to one of the below two lattices:

a)



b)



Corollary A.4.4.8

If (P, \sqsubseteq) is distributive, then (P, \sqsubseteq) is modular.

Boolean Lattices

Let (P, \sqsubseteq) be a lattice with meet operation \sqcap , join operation \sqcup , least element \perp , and greatest element \top .

Definition A.4.4.9 (Complement)

Let $p, q \in P$. Then:

1. q is called a **complement** of p , if $p \sqcup q = \top$ and $p \sqcap q = \perp$.
2. P is called **complementary**, if every element in P has a complement.

Definition A.4.4.10 (Boolean Lattice)

(P, \sqsubseteq) is called **Boolean**, if it is complementary, distributive, and $\perp \neq \top$.

Note: If (P, \sqsubseteq) is Boolean, then every element $p \in P$ has an unambiguous unique complement in P , which is denoted by \bar{p} .

Useful Result

Lemma A.4.4.11

Let (P, \sqsubseteq) be a Boolean lattice, and let $p, q, r \in P$. Then:

$$1. \quad \bar{\bar{p}} = p \quad (\text{Involution Law})$$

$$2. \quad \overline{p \sqcup q} = \bar{p} \sqcap \bar{q}, \quad \overline{p \sqcap q} = \bar{p} \sqcup \bar{q} \quad (\text{De Morgan Laws})$$

$$3. \quad p \sqsubseteq q \iff \bar{p} \sqcup q = \top \iff p \sqcap \bar{q} = \perp$$

$$4. \quad p \sqsubseteq q \sqcup r \iff p \sqcap \bar{q} \sqsubseteq r \iff \bar{q} \sqsubseteq \bar{p} \sqcup r$$

Boolean Lat. Homomorphisms/Isomorphisms

Let (P, \sqsubseteq_P) and (Q, \sqsubseteq_Q) be two Boolean lattices, and let $f \in [P \rightarrow Q]$ be a function from P to Q .

Definition A.4.4.12 (Boolean Lattice Homomorphism)

f is called a **Boolean lattice homomorphism**, if f is a lattice homomorphism and

$$\forall p \in P. f(\bar{p}) = \overline{f(p)}$$

Definition A.4.4.13 (Boolean Lattice Isomorphism)

f is called a **Boolean lattice isomorphism**, if f is a Boolean lattice homomorphism and bijective.

Useful Results

Let (P, \sqsubseteq_P) and (Q, \sqsubseteq_Q) be two Boolean lattices, and let $f \in [P \xrightarrow{bhom} Q]$ be a Boolean lattice homomorphism from P to Q .

Lemma A.4.4.14

$$f(\perp) = \perp \wedge f(\top) = \top$$

Lemma A.4.4.15

f is a Boolean lattice isomorphism iff $f(\perp) = \perp \wedge f(\top) = \top$

Summary, Overview

Corollary A.4.4.16

Let $P \neq \emptyset$ be a non-empty set, and \sqsubseteq a relation on P . Then:

- (P, \sqsubseteq) Boolean lattice
- (Def. A.4.4.10) $\Rightarrow (P, \sqsubseteq)$ distributive lattice
- (Cor. A.4.4.8) $\Rightarrow (P, \sqsubseteq)$ modular lattice
- (Def. A.4.4.2) $\Rightarrow (P, \sqsubseteq)$ lattice
- (Def. A.4.1.1) $\Rightarrow (P, \sqsubseteq)$ partial order
- (Def. A.2.1.2) $\Rightarrow (P, \sqsubseteq)$ pre-order

Corollary A.4.4.17

$$\mathcal{QO} \supset \mathcal{PO} \supset \mathcal{L} \supset \mathcal{ML} \supset \mathcal{DL} \supset \mathcal{BL}$$

where all inclusions are proper and \mathcal{QO} , \mathcal{PO} , \mathcal{L} , \mathcal{ML} , \mathcal{DL} , and \mathcal{BL} denote the sets of all quasi- (or pre-) orders, partial orders, lattices, modular lattices, distributive lattices, and Boolean lattices, respectively.

Exercise A.4.4.18

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or disprove: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. modular lattice
2. distributive lattice
3. Boolean lattice

Proof or counterexample.

A.4.5

Mechanisms for Constructing Lattices

Common Lattice Constructions: Flat Lattices

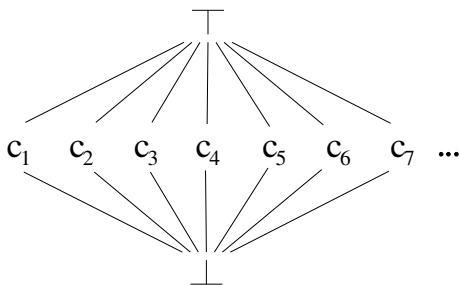
Lemma A.4.5.1 (Flat Lattice Construction)

Let C be a set. Then:

$(C \dot{\cup} \{\perp, \top\}, \sqsubseteq_{flat})$ with \sqsubseteq_{flat} defined by

$$\forall c, d \in C \dot{\cup} \{\perp, \top\}. c \sqsubseteq_{flat} d \iff_{df} c = \perp \vee c = d \vee d = \top$$

is a **complete lattice**, a so-called **flat lattice** (or **diamond lattice**).



Lattice Constructions: Products, Sums,...

Like the principle for constructing flat CPOs also the principles for constructing

- ▶ non-strict products
- ▶ strict products
- ▶ separate sums
- ▶ coalesced sums
- ▶ continuous (here: additive, distributive) function spaces

carry over from CPOs to (complete) lattices (cf. App. A.3.3).

A.4.6

Order-theoretic and Algebraic View of Lattices

Motivation

In [Definition A.4.1.1](#), we introduced [lattices](#) as special

- ▶ [ordered sets](#) (P, \sqsubseteq)

which induces an

- ▶ [order-theoretic](#) view of lattices.

Alternatively, [lattices](#) can be introduced as special

- ▶ [algebraic structures](#) (P, \sqcap, \sqcup)

which induces an

- ▶ [algebraic](#) view of lattices.

Next, we will show that both views are equivalent:

- ▶ [Order-theoretically](#) defined lattices can be considered [algebraically](#) and vice versa.

Lattices as Algebraic Structures

Definition A.4.6.1 (Algebraic Lattice)

An **algebraic lattice** is an algebraic structure (P, \sqcap, \sqcup) , where

- ▶ $P \neq \emptyset$ is a non-empty set.
- ▶ $\sqcap, \sqcup : P \times P \rightarrow P$ are two maps such that for all elements $p, q, r \in P$ the following laws hold (infix notation):
 - ▶ **Commutative Laws:** $p \sqcap q = q \sqcap p$
 $p \sqcup q = q \sqcup p$
 - ▶ **Associative Laws:** $(p \sqcap q) \sqcap r = p \sqcap (q \sqcap r)$
 $(p \sqcup q) \sqcup r = p \sqcup (q \sqcup r)$
 - ▶ **Absorption Laws:** $(p \sqcap q) \sqcup p = p$
 $(p \sqcup q) \sqcap p = p$

Properties of Algebraic Lattices

Let (P, \sqcap, \sqcup) be an algebraic lattice.

Lemma A.4.6.2 (Idempotency Laws)

For all $p \in P$, the maps $\sqcap, \sqcup : P \times P \rightarrow P$ satisfy the following laws:

- ▶ **Idempotency Laws:** $p \sqcap p = p$
 $p \sqcup p = p$

Lemma A.4.6.3

For all $p, q \in P$, the maps $\sqcap, \sqcup : P \times P \rightarrow P$ satisfy:

1. $p \sqcap q = p \iff p \sqcup q = q$
2. $p \sqcap q = p \sqcup q \iff p = q$

Induced (Partial) Order

Let (P, \sqcap, \sqcup) be an algebraic lattice.

Lemma A.4.6.4

The relation $\sqsubseteq \subseteq P \times P$ on P defined by

$$\forall p, q \in P. p \sqsubseteq q \iff_{df} p \sqcap q = p$$

is a partial order relation on P , i.e., \sqsubseteq is reflexive, transitive, and antisymmetric.

Definition A.4.6.5 (Induced Partial Order)

The relation \sqsubseteq defined in Lemma A.4.6.4 is called the **induced (partial) order** of (P, \sqcap, \sqcup) .

Properties of the Induced Partial Order

Let (P, \sqcap, \sqcup) be an algebraic lattice, and let \sqsubseteq be the induced partial order of (P, \sqcap, \sqcup) .

Lemma A.4.6.6

For all $p, q \in P$, the infimum ($\hat{=}$ greatest lower bound) and the supremum ($\hat{=}$ least upper bound) of the set $\{p, q\}$ exist and are given by the images of \sqcap and \sqcup applied to p and q , respectively, i.e.:

$$\forall p, q \in P. \sqcap\{p, q\} = p \sqcap q \wedge \sqcup\{p, q\} = p \sqcup q$$

Lemma A.4.6.6 can inductively be extended yielding:

Lemma A.4.6.7

Let $\emptyset \neq Q \subseteq P$ be a non-empty finite subset of P . Then:

$$\exists glb, lub \in P. glb = \sqcap Q \wedge lub = \sqcup Q$$

Algebraic Lattices Order-theoretically

Corollary A.4.6.8 (From (P, \sqcap, \sqcup) to (P, \sqsubseteq))

Let (P, \sqcap, \sqcup) be an algebraic lattice. Then:

(P, \sqsubseteq) , where \sqsubseteq is the induced partial order of (P, \sqcap, \sqcup) , is an order-theoretic lattice in the sense of [Definition A.4.1.1](#).

Induced Algebraic Maps

Let (P, \sqsubseteq) be an order-theoretic lattice.

Definition A.4.6.9 (Induced Algebraic Maps)

The partial order \sqsubseteq of (P, \sqsubseteq) induces two maps \sqcap and \sqcup from $P \times P$ to P defined by:

1. $\forall p, q \in P. p \sqcap q =_{df} \sqcap\{p, q\}$
2. $\forall p, q \in P. p \sqcup q =_{df} \sqcup\{p, q\}$

Properties of the Induced Algebraic Maps (1)

Let (P, \sqsubseteq) be an order-theoretic lattice, and let \sqcap and \sqcup be the induced algebraic maps of (P, \sqsubseteq) .

Lemma A.4.6.10

Let $p, q \in P$. Then the following statements are equivalent:

1. $p \sqsubseteq q$
2. $p \sqcap q = p$
3. $p \sqcup q = q$

Properties of the Induced Algebraic Maps (2)

Let (P, \sqsubseteq) be an order-theoretic lattice, and let \sqcap and \sqcup be the induced algebraic maps of (P, \sqsubseteq) .

Lemma A.4.6.11

For all $p, q, r \in P$, the induced maps \sqcap and \sqcup satisfy the following laws:

- ▶ **Commutative Laws:** $p \sqcap q = q \sqcap p$
 $p \sqcup q = q \sqcup p$
- ▶ **Associative Laws:** $(p \sqcap q) \sqcap r = p \sqcap (q \sqcap r)$
 $(p \sqcup q) \sqcup r = p \sqcup (q \sqcup r)$
- ▶ **Absorption Laws:** $(p \sqcap q) \sqcup p = p$
 $(p \sqcup q) \sqcap p = p$
- ▶ **Idempotency Laws:** $p \sqcap p = p$
 $p \sqcup p = p$

Order-theoretic Lattices Algebraically

Corollary A.4.6.12 (From (P, \sqsubseteq) to (P, \sqcap, \sqcup))

Let (P, \sqsubseteq) be an order-theoretic lattice. Then:

(P, \sqcap, \sqcup) , where \sqcap and \sqcup are the induced maps of (P, \sqsubseteq) , is an algebraic lattice in the sense of [Definition A.4.6.1](#).

Equivalence (1)

...of the [order-theoretic](#) and the [algebraic view](#) of lattices.

From [order-theoretic](#) to [algebraic lattices](#):

- ▶ An order-theoretic lattice (P, \sqsubseteq) can be considered algebraically by switching from (P, \sqsubseteq) to (P, \sqcap, \sqcup) , where \sqcap and \sqcup are the induced maps of (P, \sqsubseteq) .

From [algebraic](#) to [order-theoretic lattices](#):

- ▶ An algebraic lattice (P, \sqcap, \sqcup) can be considered order-theoretically by switching from (P, \sqcap, \sqcup) to (P, \sqsubseteq) , where \sqsubseteq is the induced partial order of (P, \sqcap, \sqcup) .

Equivalence (2)

Together, this allows us to simply speak of a lattice P , and to speak only more precisely of P as an

- ▶ order-theoretic lattice (P, \sqsubseteq)
- ▶ algebraic lattice (P, \sqcap, \sqcup)

if we want to emphasize that we think of P as a special **ordered set** or as a special **algebraic structure**.

Bottom and Top vs. Zero and One (1)

Let P be a lattice with a least and a greatest element.

Considering P

- ▶ **order-theoretically** as (P, \sqsubseteq) , it is appropriate to think of its least and greatest element in terms of bottom \perp and top \top with
 - ▶ Bottom $\perp \in P$: $\perp = \bigsqcup \emptyset$
 - ▶ Top $\top \in P$: $\top = \bigsqcap \emptyset$
- ▶ **algebraically** as (P, \sqcap, \sqcup) , it is appropriate to think of its least and greatest element in terms of Zero $\mathbf{0}$ and One $\mathbf{1}$, where (P, \sqcap, \sqcup) is said to have a (if existent, uniquely determined)
 - ▶ Zero $\mathbf{0} \in P$: $\forall p \in P. p \sqcup \mathbf{0} = p$
 - ▶ One $\mathbf{1} \in P$: $\forall p \in P. p \sqcap \mathbf{1} = p$

Bottom and Top vs. Zero and One (2)

Lemma A.4.6.13

Let P be a lattice. Then:

- ▶ (P, \sqsubseteq) has a bottom element \perp iff (P, \sqcap, \sqcup) has a zero element $\mathbf{0}$, and in that case:

$$(\bigsqcup \emptyset =) \perp = \mathbf{0}$$

- ▶ (P, \sqsubseteq) has a top element \top iff (P, \sqcap, \sqcup) has a one element $\mathbf{1}$, and in that case:

$$(\bigsqcap \emptyset =) \top = \mathbf{1}$$

On the Adequacy of the two Lattice Views

In **mathematics**, usually the

- ▶ **algebraic view** of a lattice is more appropriate as it is in line with other algebraic structures ('a set together with some maps satisfying a number of laws'), e.g., **groups, rings, fields, vector spaces, categories**, etc., which are investigated and dealt with in mathematics.

In **computer science**, usually the

- ▶ **order-theoretic view** of a lattice is more appropriate, since the order relation can often be interpreted and understood as '**· carries more/less information than ·,**' '**· is more/less defined than ·,**' '**· is stronger/weaker than ·,**' etc., which often fits naturally to problems investigated and dealt with in computer science.

Exercise A.4.6.14

Let $(\mathbb{N}_0, \sqsubseteq)$ be the lattice with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Provide the definition of $(\mathbb{N}_0, \wedge, \vee)$, i.e., write down the algebraically defined counterpart of $(\mathbb{N}_0, \sqsubseteq)$. To this end, provide the definition of the meet and join operation on $\mathbb{N}_0 \times \mathbb{N}_0$:

1. $\wedge : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$
2. $\vee : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$

What is the

1. zero element **0**
2. one element **1**

of $(\mathbb{N}_0, \wedge, \vee)$?

A.5

Fixed Point Theorems

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1638/18

A.5.1

Fixed Points, Towers

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1639/18

Fixed Points of Functions

Definition A.5.1.1 (Fixed Point)

Let M be a set, let $f \in [M \rightarrow M]$ be a function on M , and let $m \in M$ be an element of M . Then:

m is called a **fixed point** of f iff $f(m) = m$.

Least, Greatest Fixed Points in Partial Orders

Definition A.5.1.2 (Least, Greatest Fixed Point)

Let (P, \sqsubseteq) be a partial order, let $f \in [P \rightarrow P]$ be a function on P , and let p be a fixed point of f , i.e., $f(p) = p$. Then:

p is called the

- ▶ **least fixed point** of f , denoted by μf ,
iff $\forall q \in P. f(q) = q \Rightarrow p \sqsubseteq q$
- ▶ **greatest fixed point** of f , denoted by νf ,
iff $\forall q \in P. f(q) = q \Rightarrow q \sqsubseteq p$

Towers in Chain Complete Partial Orders

Definition A.5.1.3 (f -Tower in C)

Let (C, \sqsubseteq) be a CCPO, let $f \in [C \rightarrow C]$ be a function on C , and let $T \subseteq C$ be a subset of C . Then:

T is called an f -tower in C iff

1. $\perp \in T$.
2. If $t \in T$, then also $f(t) \in T$.
3. If $T' \subseteq T$ is a chain in C , then $\bigsqcup T' \in T$.

Least Towers in Chain Complete Partial Orders

Lemma A.5.1.4 (The Least f -Tower in C)

The intersection

$$I =_{df} \bigcap \{T \mid T \text{ } f\text{-tower in } C\}$$

of all f -towers in C is the least f -tower in C , i.e.,

1. I is an f -tower in C .
2. $\forall T$ f -tower in C . $I \subseteq T$.

Lemma A.5.1.5 (Least f -Towers and Chains)

The least f -tower in C is a chain in C , if f is expanding.

A.5.2

Fixed Point Theorems for Complete Partial Orders

Fixed Points of Exp./Monotonic Functions

Fixed Point Theorem A.5.2.1 (Expanding Function)

Let (C, \sqsubseteq) be a CCPO, and let $f \in [C \xrightarrow{\text{exp}} C]$ be an expanding function on C . Then:

The supremum of the least f -tower in C is a fixed point of f .

Fixed Point Theorem A.5.2.2 (Monotonic Function)

Let (C, \sqsubseteq) be a CCPO, and let $f \in [C \xrightarrow{\text{mon}} C]$ be a monotonic function on C . Then:

f has a unique least fixed point μf , which is given by the supremum of the least f -tower in C .

Note

- ▶ [Theorem A.5.2.1](#) and [Theorem A.5.2.2](#) ensure the existence of a fixed point for expanding functions and of a unique least fixed point for monotonic functions, respectively, but do not provide constructive procedures for computing or approximating them.
- ▶ This is in contrast to [Theorem A.5.2.3](#), which does so for continuous functions. In practice, continuous functions are thus more important and considered where possible.

Least Fixed Points of Continuous Functions

Fixed Point Theorem A.5.2.3 (Knaster, Tarski, Kleene)

Let (C, \sqsubseteq) be a CCPO, and let $f \in [C \xrightarrow{\text{con}} C]$ be a continuous function on C . Then:

f has a unique **least fixed point** $\mu f \in C$, which is given by the **supremum** of the (so-called) **Kleene chain** $\{\perp, f(\perp), f^2(\perp), \dots\}$, i.e.:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{\perp, f(\perp), f^2(\perp), \dots\}$$

Note: $f^0 =_{df} Id_C$; $f^i =_{df} f \circ f^{i-1}$, $i > 0$.

Proof of Fixed Point Theorem A.5.2.3 (1)

We have to prove:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{f^i(\perp) \mid i \geq 0\}$$

1. exists,
2. is a fixed point of f ,
3. is the least fixed point of f .

Proof of Fixed Point Theorem A.5.2.3 (2)

1. Existence

- ▶ By definition of \perp as the least element of C and of f^0 as the identity on C we have: $\perp = f^0(\perp) \sqsubseteq f^1(\perp) = f(\perp)$.
- ▶ Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
 $\forall i, j \in \mathbb{N}_0. i < j \Rightarrow f^i(\perp) \sqsubseteq f^{i+1}(\perp) \sqsubseteq f^j(\perp)$.
- ▶ Hence, the set $\{f^i(\perp) \mid i \geq 0\}$ is a (possibly infinite) chain in C .
- ▶ Since (C, \sqsubseteq) is a CCPO and $\{f^i(\perp) \mid i \geq 0\}$ a chain in C , this implies by definition of a CCPO that the least upper bound of the chain $\{f^i(\perp) \mid i \geq 0\}$

$$\bigsqcup \{f^i(\perp) \mid i \geq 0\} = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \text{ exists.}$$

Proof of Fixed Point Theorem A.5.2.3 (3)

2. Fixed point property

$$f\left(\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)\right)$$

$$(f \text{ continuous}) = \bigsqcup_{i \in \mathbb{N}_0} f(f^i(\perp))$$

$$= \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp)$$

$(C' =_{df} \{f^i \perp \mid i \geq 1\})$ is a chain \Rightarrow

$$\bigsqcup C' \text{ exists} = \perp \sqcup \bigsqcup C' = \perp \sqcup \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp)$$

$$(f^0(\perp) =_{df} \perp) = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$$

Proof of Fixed Point Theorem A.5.2.3 (4)

3. Least fixed point property

- ▶ Let c be an arbitrary fixed point of f . Then: $\perp \sqsubseteq c$.
- ▶ Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
 $\forall i \in \mathbb{N}_0. f^i(\perp) \sqsubseteq f^i(c) (= c)$.
- ▶ Since c is a fixed point of f , this implies:
 $\forall i \in \mathbb{N}_0. f^i(\perp) \sqsubseteq c (= f^i(c))$.
- ▶ Thus, c is an upper bound of the set $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$.
- ▶ Since $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$ is a chain, and $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is by definition the least upper bound of this chain, we obtain the desired inclusion

$$\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c.$$

□

Least Conditional Fixed Points

Let (C, \sqsubseteq) be a CCPO, let $f \in [C \rightarrow C]$ be a function on C , and let $d, c_d \in C$ be elements of C .

Definition A.5.2.4 (Least Conditional Fixed Point)

c_d is called the **least conditional fixed point** of f wrt d (in German: *kleinster bedingter Fixpunkt*) iff c_d is the least fixed point of C with $d \sqsubseteq c_d$, i.e.:

$$\forall x \in C. f(x) = x \wedge d \sqsubseteq x \Rightarrow c_d \sqsubseteq x$$

Least Cond. Fixed Points of Cont. Functions

Theorem A.5.2.5 (Conditional Fixed Point Theorem)

Let (C, \sqsubseteq) be a CCPO, let $d \in C$, and let $f \in [C \xrightarrow{\text{con}} C]$ be a continuous function on C which is expanding for d , i.e., $d \sqsubseteq f(d)$. Then:

f has a least conditional fixed point $\mu f_d \in C$, which is given by the supremum of the (generalized) Kleene chain $\{d, f(d), f^2(d), \dots\}$, i.e.:

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \dots\}$$

Finite Fixed Points

Let (C, \sqsubseteq) be a CCPO, let $d \in C$, and let $f \in [C \xrightarrow{\text{mon}} C]$ be a monotonic function on C .

Theorem A.5.2.6 (Finite Fixed Point Theorem)

If two succeeding elements in the Kleene chain of f are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(\perp) = f^{i+1}(\perp)$, then we have: $\mu f = f^i(\perp)$.

Theorem A.5.2.7 (Finite Conditional FP Theorem)

If f is expanding for d , i.e., $d \sqsubseteq f(d)$, and two succeeding elements in the (generalized) Kleene chain of f wrt d are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(d) = f^{i+1}(d)$, then we have: $\mu f_d = f^i(d)$.

Note: Theorems A.5.2.6 and A.5.2.7 do not require continuity of f . Monotonicity (and expandingness) of f suffice(s).

Towards the Existence of Finite Fixed Points

Let (P, \sqsubseteq) be a partial order, and let $p, r \in P$.

Definition A.5.2.8 (Chain-finite Partial Order)

(P, \sqsubseteq) is called **chain-finite** (in German: *kettenendlich*) iff P does not contain an infinite chain.

Definition A.5.2.9 (Finite Element)

p is called

- ▶ **finite** iff the set $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ does not contain an infinite chain.
- ▶ **finite relative to r** iff the set $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ does not contain an infinite chain.

Existence of Finite Fixed Points

...there are numerous sufficient conditions ensuring the existence of a least finite fixed point of a function f , which often hold in practice (cf. Nielson/Nielson 1992), e.g.:

- ▶ the domain or the range of f are finite or chain-finite,
- ▶ the least fixed point of f is finite,
- ▶ f is of the form $f(c) = c \sqcup g(c)$ with g a monotonic function on a chain-finite (data) domain.

Fixed Point Theorems, Lattices, and DCPOs

Note: Complete lattices (cf. [Lemma A.4.1.13](#)) and DCPOs with a least element (cf. [Lemma A.3.1.5](#)) are CCPOs, too.

Thus, we can conclude:

Corollary A.5.2.10 (Fixed Points, Lattices, DCPOs)

The fixed point theorems of [Chapter A.5.2](#) hold for functions on complete lattices and on DCPOs with a least element, too.

A.5.3

Fixed Point Theorems for Lattices

Fixed Points of Monotonic Functions

Fixed Point Theorem A.5.3.1 (Knaster, Tarski)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \xrightarrow{\text{mon}} P]$ be a monotonic function on P . Then:

1. f has a unique least fixed point $\mu f \in P$, which is given by $\mu f = \bigcap \{p \in P \mid f(p) \sqsubseteq p\}$.
2. f has a unique greatest fixed point $\nu f \in P$, which is given by $\nu f = \bigcup \{p \in P \mid p \sqsubseteq f(p)\}$.

Characterization Theorem A.5.3.2 (Davis)

Let (P, \sqsubseteq) be a lattice. Then:

(P, \sqsubseteq) is complete iff every $f \in [P \xrightarrow{\text{mon}} P]$ has a fixed point.

The Fixed Point Lattice of Mon. Functions

Theorem A.5.3.3 (Lattice of Fixed Points)

Let (P, \sqsubseteq) be a complete lattice, let $f \in [P \xrightarrow{\text{mon}} P]$ be a monotonic function on P , and let $\text{Fix}(f) =_{df} \{p \in P \mid f(p) = p\}$ be the set of all fixed points of f . Then:

Every subset $F \subseteq \text{Fix}(f)$ has a supremum and an infimum in $\text{Fix}(f)$, i.e., $(\text{Fix}(f), \sqsubseteq|_{\text{Fix}(f)})$ is a complete lattice.

Theorem A.5.3.4 (Ordering of Fixed Points)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \xrightarrow{\text{mon}} P]$ be a monotonic function on P . Then:

$$\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq \mu f \sqsubseteq \nu f \sqsubseteq \bigsqcap_{i \in \mathbb{N}_0} f^i(\top)$$

Fixed Points of Add./Distributive Functions

For **additive** and **distributive functions**, the leftmost and the rightmost inequality of **Theorem A.5.3.4** become equalities:

Fixed Point Theorem A.5.3.5 (Knaster,Tarski,Kleene)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a function on P . Then: f has a unique

1. least fixed point $\mu f \in P$ given by $\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$, if f is additive, i.e., $f \in [P \xrightarrow{add} P]$.
2. greatest fixed point $\nu f \in P$ given by $\nu f = \bigsqcap_{i \in \mathbb{N}_0} f^i(\top)$, if f is distributive, i.e., $f \in [P \xrightarrow{dis} P]$.

Recall: $f^0 =_{df} Id_C$; $f^i =_{df} f \circ f^{i-1}$, $i > 0$.

A.6

Fixed Point Induction

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1662/18

Admissible Predicates

Fixed point induction allows proving properties of fixed points. Essential is the notion of **admissible predicates**:

Definition A.6.1 (Admissible Predicate)

Let (P, \sqsubseteq) be a complete lattice, and let $\phi : P \rightarrow \mathbb{B}$ be a predicate on P . Then:

ϕ is called **admissible** (or **\sqcup -admissible**) iff for every chain $C \subseteq P$ holds:

$$(\forall c \in C. \phi(c)) \Rightarrow \phi(\bigsqcup C)$$

Lemma A.6.2

Let (P, \sqsubseteq) be a complete lattice, and let $\phi : P \rightarrow \mathbb{B}$ be an admissible predicate on P . Then: $\phi(\perp) = \text{true}$.

Proof. The admissibility of ϕ implies $\phi(\bigsqcup \emptyset) = \text{true}$. Moreover, we have $\perp = \bigsqcup \emptyset$, which completes the proof.

Sufficient Conditions for Admissibility

Theorem A.6.3 (Admissibility Condition 1)

Let (P, \sqsubseteq) be a complete lattice, and let $\phi : P \rightarrow \mathbb{B}$ be a predicate on P . Then:

ϕ is admissible, if there is a complete lattice (Q, \sqsubseteq_Q) and two additive functions $f, g \in [P \xrightarrow{add} Q]$, such that

$$\forall p \in P. \phi(p) \iff f(p) \sqsubseteq_Q g(p)$$

Theorem A.6.4 (Admissibility Condition 2)

Let (P, \sqsubseteq) be a complete lattice, and let $\phi, \psi : P \rightarrow \mathbb{B}$ be two admissible predicates on P . Then:

The conjunction of ϕ and ψ , the predicate $\phi \wedge \psi$ defined by

$$\forall p \in P. (\phi \wedge \psi)(p) =_{df} \phi(p) \wedge \psi(p)$$

is admissible.

Fixed Point Induction on Complete Lattices

Theorem A.6.5 (Fixed Point Induction on C. Lat.)

Let (P, \sqsubseteq) be a complete lattice, let $f \in [P \xrightarrow{\text{add}} P]$ be an additive function on P , and let $\phi : P \rightarrow \mathbb{B}$ be an admissible predicate on P . Then:

The validity of

$$\blacktriangleright \forall p \in P. \phi(p) \Rightarrow \phi(f(p)) \quad (\text{Induction step})$$

implies the validity of $\phi(\mu f)$.

Note: The **induction base**, i.e., the validity of $\phi(\perp)$, is implied by the admissibility of ϕ (cf. [Lemma A.6.2](#)) and proved when verifying the admissibility of ϕ .

Fixed Point Induction on CCPOs

The notion of admissibility of a predicate carries over from complete lattices to CCPOs.

Theorem A.6.6 (Fixed Point Induction on CCPOs)

Let (C, \sqsubseteq) be a CCPO, let $f \in [C \xrightarrow{mon} C]$ be a monotonic function on C , and let $\phi : C \rightarrow \mathbb{B}$ be an admissible predicate on C . Then:

The validity of

$$\blacktriangleright \forall c \in C. \phi(c) \Rightarrow \phi(f(c)) \quad (\text{Induction step})$$

implies the validity of $\phi(\mu f)$.

Note: Theorem A.6.6 holds (of course still), if we replace the CCPO (C, \sqsubseteq) by a complete lattice (P, \sqsubseteq) .

A.7

Completions, Embeddings

A.7.1

Downsets

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1668/18

Downsets

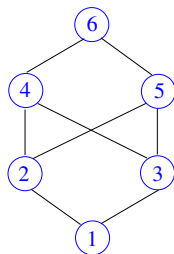
Definition A.7.1.1 (Downset)

Let (P, \sqsubseteq) be a partial order, let $D \subseteq P$ be a subset of P , and let $p, q \in P$ with $p \sqsubseteq q$. Then:

1. D is called a **downset** (or **lower set** or **order ideal**) (in German: *Abwärtsmenge*) of P , if: $q \in D \Rightarrow p \in D$.
2. $\mathcal{D}(P)$ denotes the **set** of all **downsets** of P .

Example

Let (P, \sqsubseteq) be the partial order given by the below Hasse diagram.



Then, e.g.:

1. $\emptyset, P \in \mathcal{D}(P), \forall q \in P. \{p \in P \mid p \sqsubseteq q\} \in \mathcal{D}(P)$
2. $\{1, 3\}, \{1, 2, 3\}, \{1, 2, 3, 4\} \in \mathcal{D}(P)$
3. $\{2, 3\}, \{2, 4, 5\}, \{1, 2, 4, 5\} \notin \mathcal{D}(P)$

Properties of Downsets

Lemma A.7.1.2

Let (P, \sqsubseteq) be a partial order, let $q \in P$, and $Q \subseteq P$. Then:

1. $\emptyset \in \mathcal{D}(P)$, $P \in \mathcal{D}(P)$, are (trivial) downsets of P .
2. $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\} \in \mathcal{D}(P)$.
3. $\downarrow Q =_{df} \{p \in P \mid \exists q \in Q. p \sqsubseteq q\} \in \mathcal{D}(P)$.
4. $Q \in \mathcal{D}(P) \iff Q = \downarrow Q$

Lemma A.7.1.3

Let (P, \sqsubseteq) be a partial order, and let $p, q \in P$. Then the following statements are equivalent:

1. $p \sqsubseteq q$
2. $\downarrow p \subseteq \downarrow q$
3. $\forall D \in \mathcal{D}(P). q \in D \Rightarrow p \in D$.

Characterization of Downsets

Lemma A.7.1.4 (Downsets of a PO)

Let (P, \sqsubseteq) be a partial order. Then:

$$\mathcal{D}(P) = \{\downarrow Q \mid Q \subseteq P\}$$

Corollary A.7.1.5

Let (P, \sqsubseteq) be a partial order, let $D \in \mathcal{D}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in D \Rightarrow p \in D$.

The Lattice of Downsets: Complete & Distr.

Let (P, \sqsubseteq) be a partial order, let $\mathcal{D}(P)$ be the set of downsets of P , and let \subseteq denote set inclusion.

Theorem A.7.1.6 (Complete & Distr. L. of Downsets)

$(\mathcal{D}(P), \subseteq)$ is a complete and distributive lattice, the so-called **downset lattice** of P , with set intersection \cap as meet operation, set union \cup as join operation, least element \emptyset , and greatest element P .

Recall: Complete lattices are CCPOs and DCPOs, too (cf. Lemma A.4.1.13). Thus, we have:

Corollary A.7.1.7 (The CCPO/DCPO of Downsets)

$(\mathcal{D}(P), \subseteq)$ is a **CCPO** and a **DCPO** with least element \emptyset .

From POs to Lattices, CCPOs, and DCPOs

Construction Principle:

Theorem A.7.1.6 and Corollary A.7.1.7 yield a construction principle that shows how to construct

- ▶ a complete lattice and thus also a CCPO and a DCPO from a given partial order (P, \sqsubseteq) (cf. Appendix A.3.3 and Appendix A.4.5).

Principal Downsets

The downsets of the form $\{p \in P \mid p \sqsubseteq q\}$ of a partial order (P, \sqsubseteq) considered in Lemma A.7.1.2(2) are peculiar, and will reoccur as so-called **principal ideals** (cf. Chapter A.7.2) and **principal cuts** (cf. Chapter A.7.3) of lattices. Therefore, we introduce these distinguished downsets explicitly.

Definition A.7.1.8 (Principal Downsets of a PO)

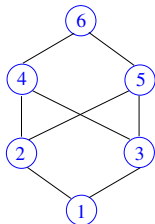
Let (P, \sqsubseteq) be a partial order, and let $q \in P$ be an element of P . Then:

1. $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\}$ denotes the **principal downset** (in German: **Hauptabwärtsmenge**) generated by q .
2. $\mathcal{PD}(P) = \{\downarrow q \mid q \in P\}$ denotes the **set** of all **principal downsets** of P .

Downsets, Directed Sets (1)

...principal downsets of partial orders are **directed** but usually not strongly directed.

Example 1: Consider the below partial order (P, \sqsubseteq) :



- ▶ $\forall p \in P. \downarrow p =_{df} \{r \mid r \sqsubseteq p\}$ *directed* $\in \mathcal{D}(P)$.
- ▶ $\forall p \in P \setminus \{6\}. \downarrow p$ *strongly directed* $\in \mathcal{D}(P)$.
- ▶ $\downarrow 6 =_{df} \{r \mid r \sqsubseteq 6\} = \{1, 2, 3, 4, 5, 6\} = P \in \mathcal{D}(P)$ is a downset of P , however, it is not strongly directed, since its subsets $\{2, 3\}, \{1, 2, 3\} \subseteq \downarrow 6$ do not have a least upper bound in $\downarrow 6 = P$ (though upper bounds: $4, 5, 6$).

Downsets, Directed Sets (2)

Example 2: Consider the below lattice (\mathbb{Z}, \leq) :

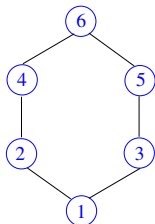


- ▶ $\mathcal{D}(\mathbb{Z}) = \emptyset \cup \mathcal{PD}(\mathbb{Z}) \cup \mathbb{Z} = \emptyset \cup \{\downarrow z =_{df} \{r \in \mathbb{Z} \mid r \leq z\} \mid z \in \mathbb{Z}\} \cup \mathbb{Z}$
- ▶ $\forall S \in \mathcal{D}(\mathbb{Z})$. S directed but not strongly directed (since it lacks a least element).

Downsets, Directed Sets (3)

...arbitrary downsets even of complete lattices are usually **not strongly directed**, though **directed**.

Example 3: Consider the below complete lattice (P, \sqsubseteq) :



► E.g., the downsets

► $\downarrow\{4, 5\} =_{df} \{r \mid r \sqsubseteq 4 \vee r \sqsubseteq 5\} = \{1, 2, 3, 4, 5\} \in \mathcal{D}(P)$

► $\downarrow\{3, 4\} =_{df} \{r \mid r \sqsubseteq 3 \vee r \sqsubseteq 4\} = \{1, 2, 3, 4\} \in \mathcal{D}(P)$

of P are directed but not strongly directed: The subsets $\{2, 3\} \subseteq \downarrow\{4, 5\}$ and $\{1, 2, 3\} \subseteq \downarrow\{3, 4\}$ do not have a least upper bound in $\downarrow\{4, 5\}$ and $\downarrow\{3, 4\}$, respectively.

A.7.2

Ideal Completion: Embedding of Lattices

Lattice Ideals

Definition A.7.2.1 (Lattice Ideal)

Let (P, \sqsubseteq) be a lattice, let $\emptyset \neq I \subseteq P$ be a non-empty subset of P , and let $p, q \in P$. Then:

1. I is called an **ideal** (or **lattice ideal**) of P , if:

- ▶ $p, q \in I \Rightarrow p \sqcup q \in I$.
- ▶ $q \in I \Rightarrow p \sqcap q \in I$.

2. $\mathcal{I}(P)$ denotes the **set** of all **ideals** of P .

Properties of Lattice Ideals

Lemma A.7.2.2 (Ideal Properties 1)

Let (P, \sqsubseteq) be a lattice, let $I \in \mathcal{I}(P)$, and let $q \in I$. Then:

1. $\{p \in P \mid p \sqsubseteq q\} \subseteq I$.
2. $P \in \mathcal{I}(P)$ is a (trivial) ideal of P .

Lemma A.7.2.3 (Ideal Properties 2)

Let (P, \sqsubseteq) be a lattice with least element \perp , and $I \in \mathcal{I}(P)$. Then:

1. $\perp \in I$.
2. $\{\perp\} \in \mathcal{I}(P)$ is a (trivial) ideal of P .

Characterizing Lattice Ideals

Theorem A.7.2.4 (Ideal Characterization)

Let (P, \sqsubseteq) be a lattice, and let $\emptyset \neq I \subseteq P$ be a non-empty subset of P . Then:

$$I \in \mathcal{I}(P) \text{ iff } \forall p, q \in P. p, q \in I \iff p \sqcup q \in I$$

Lattice Ideals and Order Ideals

Lemma A.7.2.5

Let (P, \sqsubseteq) be a lattice, let $I \in \mathcal{I}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in I \Rightarrow p \in I$.

Corollary A.7.1.5 – recalled

Let (P, \sqsubseteq) be a partial order, let $D \in \mathcal{D}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in D \Rightarrow p \in D$.

Corollary A.7.2.6

Let (P, \sqsubseteq) be a lattice, and let $I \subseteq P$. Then:

$$I \in \mathcal{I}(P) \Rightarrow I \in \mathcal{D}(P) \quad (\text{i.e., } \mathcal{I}(P) \subseteq \mathcal{D}(P)).$$

Note: The reverse implication of [Corollary A.7.2.6](#) does not hold.

The Complete Lattice of Ideals

Theorem A.7.2.7 (The Complete Lattice of Ideals)

Let (P, \sqsubseteq) be a lattice with least element \perp , and let $\sqsubseteq_{\mathcal{I}}$ be the following ordering relation on the set $\mathcal{I}(P)$ of ideals of P :

$$\forall I, J \in \mathcal{I}(P). I \sqsubseteq_{\mathcal{I}} J \text{ iff } I \subseteq J$$

Then: $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ is a complete lattice, the so-called **lattice of ideals** of P , with join operation $\sqcup_{\mathcal{I}}$ defined by

$$\forall I, J \in \mathcal{I}(P). I \sqcup_{\mathcal{I}} J =_{df} \{p \in P \mid \exists i \in I, j \in J. p \sqsubseteq i \sqcup j\}$$

and meet operation $\sqcap_{\mathcal{I}}$ defined by

$$\forall I, J \in \mathcal{I}(P). I \sqcap_{\mathcal{I}} J =_{df} I \cap J$$

and with least element $\{\perp\}$ and greatest element P .

Principal Ideals

Lemma A.7.2.8

Let (P, \sqsubseteq) be a lattice, and let $q \in P$ be an element of P .

Then:

$$\downarrow q = \{p \in P \mid p \sqsubseteq q\} \text{ ideal} \in \mathcal{I}(P).$$

Definition A.7.2.9 (Principal Ideal)

Let (P, \sqsubseteq) be a lattice, and let $q \in P$ be an element of P .

Then:

1. $\downarrow q$ is called the **principal ideal** of P generated by q .
2. $\mathcal{PI}(P) =_{df} \{\downarrow q \mid q \in P\}$ denotes the **set** of all **principal ideals** of P .

Towards the Sublattice of Principal Ideals

Lemma A.7.2.10

Let (P, \sqsubseteq) be a lattice with least element, and let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of ideals of P . Then:

$$\forall q, r \in P. \downarrow q \sqcap_{\mathcal{I}} \downarrow r = \downarrow(q \sqcap r) \wedge \downarrow q \sqcup_{\mathcal{I}} \downarrow r = \downarrow(q \sqcup r)$$

The Sublattice of Principal Ideals

Theorem A.7.2.11 (Sublattice of Principal Ideals)

Let (P, \sqsubseteq) be a lattice with least element, let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of ideals of P , let $\mathcal{PI}(P)$ be the set of the principal ideals of P , and let $\sqsubseteq_{\mathcal{PI}}$ be the restriction of $\sqsubseteq_{\mathcal{I}}$ onto $\mathcal{PI}(P)$. Then:

$(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ is a sublattice of $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$.

Note: The sublattice $(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ of $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ is

- ▶ usually not complete, not even if (P, \sqsubseteq) is complete.

(The lattice (\mathbb{Z}, \leq) , e.g., enriched with a least element \perp and a greatest element \top is complete, while the lattice of its principal ideals $(\mathcal{PI}(\mathbb{Z}), \sqsubseteq_{\mathcal{PI}})$ is not.)

Ideal Completion and Embedding of a Lattice

Theorem A.7.2.12 (Ideal Completion & Embedding)

Let (P, \sqsubseteq) be a lattice with least element, and let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of its ideals. Then:

The map

$$e_{\mathcal{I}} : P \rightarrow \mathcal{PI}(P) \text{ defined by } \forall p \in P. e_{\mathcal{I}}(p) =_{df} \downarrow p$$

is a lattice isomorphism between P and the (sub)lattice $\mathcal{PI}(P)$ of its principal ideals.

Intuitively

Theorem A.7.2.12 shows how a lattice (P, \sqsubseteq) with least element

- ▶ can be considered a sublattice of the complete lattice of the ideals of P ; in more detail, how it can be considered the sublattice $(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ of the complete lattice $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$.

A.7.3

Cut Completion: Embedding of Partial Orders and Lattices

Cuts

Definition A.7.3.1 (Cut)

Let (P, \sqsubseteq) be a partial order, and let $Q \subseteq P$ be a subset of P .
Then:

1. Q is called a **cut** (in German: **Schnitt**) of P , if $Q = LB(UB(Q))$.
2. $\mathcal{C}(P)$ denotes the **set** of all **cuts** of P .

Properties of Cuts

Lemma A.7.3.2

Let (P, \sqsubseteq) be a partial order, and let $q \in P$ be an element of P . Then:

1. $LB(\{q\}) =_{df} \downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\} \in \mathcal{C}(P)$
2. $LB(UB(\{q\})) = \{p \in P \mid p \sqsubseteq q\} = LB(\{q\})$

Note: If (P, \sqsubseteq) is a lattice,

1. Lemma A.7.3.2(1) yields that **principal ideals** are **cuts** of P :

$$\forall q \in P. \langle q \rangle =_{df} \{p \in P \mid p \sqsubseteq q\} = LB(\{q\}) \in \mathcal{C}(P)$$

$$(\text{or: } \forall Q \subseteq P. Q \in \mathcal{PI}(P) \Rightarrow Q \in \mathcal{C}(P))$$

2. Lemma A.7.3.2(2) characterizes the principal ideals of P in terms of the function composition $LB \circ UB$.

Principal Cuts

Definition A.7.3.3 (Principal Cut)

Let (P, \sqsubseteq) be a partial order, and let $q \in P$ be an element of P . Then:

1. $\downarrow q =_{df} LB(UB(\{q\}))$ is called the **principal cut** of P generated by q .
2. $\mathcal{PC}(P) =_{df} \{\downarrow q \mid q \in P\}$ denotes the **set** of all **principal cuts** of P .

Properties of Cuts and Ideals of Lattices

Lemma A.7.3.4

Let (P, \sqsubseteq) be a lattice with least element, and let $Q \subseteq P$.
Then:

$$Q \in \mathcal{C}(P) \Rightarrow Q \in \mathcal{I}(P)$$

Corollary A.7.3.5

Let (P, \sqsubseteq) be a lattice with least element, and let $Q \subseteq P$.
Then:

$$Q \in \mathcal{C}(P) \Rightarrow Q \neq \emptyset$$

Note: Corollary A.7.3.5 does not hold for partial orders.

The Complete Lattice of Cuts

Theorem A.7.3.6 (The Complete Lattice of Cuts)

Let (P, \sqsubseteq) be a partial order, and let \sqsubseteq_c be the following ordering relation on the set $\mathcal{C}(P)$ of cuts of P :

$$\forall C, D \in \mathcal{C}(P). C \sqsubseteq_c D \text{ iff } C \subseteq D$$

Then: $(\mathcal{C}(P), \sqsubseteq_c)$ is a complete lattice, the so-called **lattice of cuts** of P , with join operation \sqcup_c defined by

$$\forall C, D \in \mathcal{C}(P). C \sqcup_c D =_{df} \bigcap \{E \in \mathcal{C}(P) \mid C \cup D \subseteq E\}$$

and meet operation \sqcap_c defined by

$$\forall C, D \in \mathcal{C}(P). C \sqcap_c D =_{df} C \cap D$$

and with least element $\{\perp\}$ and greatest element P .

Cut Completion and Embedding of a PO

Theorem A.7.3.7 (PO Cut Completion & Embedd'g)

Let (P, \sqsubseteq) be a partial order, and let $(\mathcal{C}(P), \sqsubseteq_{\mathcal{C}})$ be the complete lattice of its cuts. Then:

The map

$$e_{\mathcal{C}} : P \rightarrow \mathcal{PC}(P) \text{ defined by } \forall p \in P. e_{\mathcal{C}}(p) =_{df} LB(UB(\{p\}))$$

is an **order isomorphism** between P and the partial order $(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ of the principal cuts of P .

Cut Completion and Embedding of a Lattice

Theorem A.7.3.8 (Lattice Cut Completion & Emb'g)

Let (P, \sqsubseteq) be a lattice, let $(\mathcal{C}(P), \sqsubseteq_c)$ be the complete lattice of its cuts, and let $e_c : P \rightarrow \mathcal{PC}(P)$ be the map of Theorem A.7.3.7. Then:

$(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ is a sublattice of $(\mathcal{C}(P), \sqsubseteq)$ and e_c is a lattice isomorphism between P and the sublattice $\mathcal{PC}(P)$ of the principal cuts of P .

A.7.4

Downset Completion: Embedding of Partial Orders

Downsets, Ideals, and Cuts

Lemma A.7.4.1

We have:

1. $\mathcal{C}(P) \subseteq \mathcal{D}(P)$, if (P, \sqsubseteq) is a partial order.
2. $\mathcal{C}(P) \subseteq \mathcal{I}(P) \subseteq \mathcal{D}(P)$, if (P, \sqsubseteq) is a lattice with least element.

Downset Completion and Embedding of a PO

Theorem A.7.4.2 (Downset Completion and Emb.'g)

Let (P, \subseteq) be a partial order, and let $(\mathcal{D}(P), \subseteq)$ be the complete and distributive lattice of its downsets (cf. Theorem A.7.1.6). Then:

The map $e_c : P \rightarrow \mathcal{PC}(P)$ (of Theorem A.7.3.7) defined by

$$\forall p \in P. e_c(p) =_{df} LB(UB(\{p\}))$$

is an order isomorphism between P and the partial order $(\mathcal{PC}(P), \subseteq)$ of the principal cuts of P , or, equivalently, the map $e_c : P \rightarrow \mathcal{D}(P)$ defined as above is a partial order embedding of $(\mathcal{PC}(P), \subseteq)$ into $(\mathcal{D}(P), \subseteq)$.

Intuitively

Theorem A.7.4.2 shows how a partial order (P, \sqsubseteq)

- ▶ can be considered a partial order of the complete and distributive lattice of its downsets; in more detail, how it can be considered the partial order $(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ of the complete and distributive lattice $(\mathcal{D}(P), \sqsubseteq_{\mathcal{D}})$.

A.8

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11




Chap. 12

Chap. 13



Part IV

G 1702/18





Appendix A: Further Reading (1)

-  André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
-  Roland Backhouse, Roy Crole, Jeremy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002. (Chapter 1, Ordered Sets and Complete Lattices by Hilary A. Priestley; Chapter 2, Algebras and Coalgebras by Peter Aczel; Chapter 4, Calculating Functional Programs by Jeremy Gibbons)
-  Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Vieweg+Teubner, 2008.

Appendix A: Further Reading (2)

-  Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012. (Kapitel 1, Ordnungen und Verbände; Kapitel 2.4, Vollständige Verbände; Kapitel 3, Fixpunkttheorie mit Anwendungen; Kapitel 4, Vervollständigung und Darstellung mittels Vervollständigung; Kapitel 5, Wohlgeordnete Mengen und das Auswahlaxiom)
-  Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013. (Kapitel 2, Verbände und Ordnungen; Kapitel 3.4, Vollständige Verbände; Kapitel 4, Fixpunkttheorie mit Anwendungen; Kapitel 5, Vervollständigung und Darstellung mittels Vervollständigung; Kapitel 6, Wohlgeordnete Mengen und das Auswahlaxiom)






Appendix A: Further Reading (3)

-  Garret Birkhoff. *Applications of Lattice Algebra*. Mathematical Proceedings of the Cambridge Philosophical Society 30(2):115-122, 1934.
-  Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.
-  Peter Crawley, Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.
-  Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002. (Chapter 1, Ordered Sets; Chapter 2, Lattices and Complete Lattices; Chapter 8, CPOs and Fixpoint Theorems)





Appendix A: Further Reading (4)

-  Anne C. Davis. *A Characterization of Complete Lattices*. Pacific Journal of Mathematics 5(2):311-319, 1955.
-  Marcel Ern . *Einf hrung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
-  Helmuth Gericke. *Theorie der Verb nde*. Bibliographisches Institut, 2. Auflage, 1967.
-  George Gr tzer. *General Lattice Theory*. Birkh user, 2nd edition, 1998. (Chapter 1, First Concepts; Chapter 2, Distributive Lattices; Chapter 3, Congruences and Ideals; Chapter 5, Varieties of Lattices)
-  George Gr tzer. *Lattice Theory: Foundation*. Birkh user, 2011.




Appendix A: Further Reading (5)

-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. I*. Birkhäuser, 2014.
-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. II*. Birkhäuser, 2016.
-  Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001. (Chapter 6, Ordered Pairs; Chapter 7, Relations; Chapter 8, Functions)
-  Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.
-  Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009. (Chapter 3, Functions, Sequences, and Relations)




Appendix A: Further Reading (6)

-  Stephen C. Kleene. *Introduction to Metamathematics*. North Holland, 1952. (Reprint, North Holland, 1980)
-  Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998. (Chapter 4, Functions; Chapter 6, Relations)
-  David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008. (Chapter 1, Collecting Things Together: Sets; Chapter 2, Comparing Things: Relations)
-  George Markowsky. *Chain-complete Posets and Directed Sets with Applications*. *Algebra Universalis* 6(1):53-68, 1976.




Appendix A: Further Reading (7)

-  Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92), 96-108, 1992.
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
(Chapter 4, Denotational Semantics)
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007.
(Chapter 5, Denotational Semantics)

Appendix A: Further Reading (8)

-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Appendix A, Partially Ordered Sets)
-  Steven Roman. *Lattices and Ordered Sets*. Springer-V., 2008.
-  Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Kapitel 5.1, Ordnungsrelationen; Kapitel 5.2, Ordnungen und Teilstrukturen)

Appendix A: Further Reading (9)

-  Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Chapter 5.1, Order Relations; Chapter 5.2, Orders and Substructures)
-  Alfred Tarski. *A Lattice-theoretical Fixpoint Theorem and its Applications*. Pacific Journal of Mathematics 5(2):285-309, 1955.
-  Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008. (Chapter 5, Fixed Points; Chapter 105, Software Testing; Chapter 106, Formal Methods; Chapter 107, Verification and Validation)

Appendix B

Pragmatics: Variants of Flow Graphs

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1712/18

B.1

Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

G 1713/18

B.1.1

Flow Graph Variants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

G 1714/18

Representing Instructions in Flow Graphs

...representing **programs** by **flow graphs**, instructions (**assignments**, **tests**) can be attached to:

- ▶ nodes
- ▶ edges

as

- ▶ single instructions
- ▶ **basic blocks** (i.e., sequential sequences of instructions)

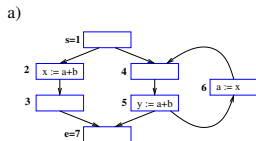
Flow Graph Variants

This leads to **four** flow graph variants:

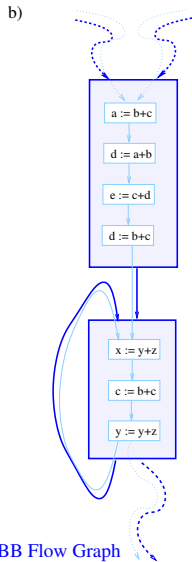
- ▶ **Node-labelled** flow graphs
(in the style of **Kripke structures**)
 - 1) Single instruction graphs (SI graphs)
 - 2) Basic block graphs (BB graphs)
- ▶ **Edge-labelled** flow graphs
(in the style of **transition systems**)
 - 3) Single instruction graphs (SI graphs)
 - 4) Basic block graphs (BB graphs)

Node-labelled Flow Graph Variants

a) Single instruction vs. b) basic block flow graphs:



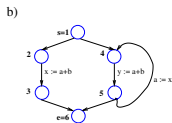
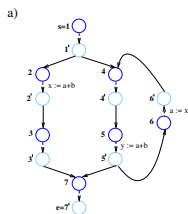
Node-labelled SI Flow Graph



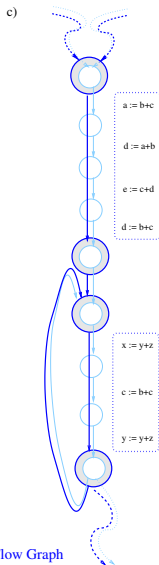
Node-labelled BB Flow Graph

Edge-labelled Flow Graph Variants

a), b) Single instruction vs. c) basic block flow graphs:



Edge-labelled SI Flow Graphs



Edge-labelled BB Flow Graph

Which Flow Graph Variant shall We Select?

Conceptually, there is

- ▶ no difference between the various flow graph variants making the choice of a particular one essentially a matter of taste.

Pragmatically, however,

- ▶ the flow graph variants differ in the ease and hence adequacy of use for specifying and implementing program analyses and optimizations.

This will be [considered](#) in more [detail](#) next.

B.1.2

Flow Graph Variants: Which one to Select?

Basic Block or Single Instruction Graphs

...node or edge-labelled, these are the questions.

We will investigate these questions by comparing the adequacy of different flow graph variants for program analysis and optimization.

To this end we consider node and edge-labelled flow graphs annotated with basic blocks and single instructions, respectively, and investigate their

- ▶ advantages and disadvantages for program analysis

from a pragmatical perspective addressing thereby especially the question:

- ▶ BB or SI graphs: Just a matter of taste?

On the fly we will learn some new data flow analyses such as

- ▶ Faint variable analysis, example of a non-separable real world data flow analysis problem.

Basic Block Graphs: Expected Advantages

Advantages commonly attributed to basic block graphs by 'folk knowledge':

Better scalability and performance because

- ▶ less nodes (edges) are involved in the (potentially) computationally costly fixed point iteration
- ▶ larger programs fit into the main memory.

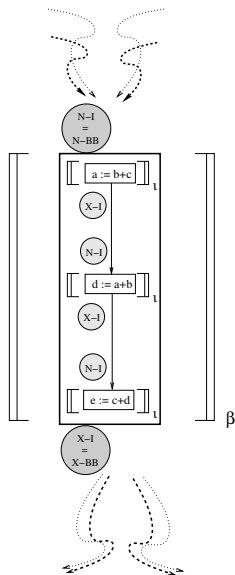
Basic Block Graphs: Definite Disadvantages

Definite disadvantages of basic block graphs in practice:

- ▶ **Higher conceptual complexity:** Basic blocks introduce an undesired *hierarchy* into flow graphs making both theoretical reasoning and practical implementations more difficult.
- ▶ **Need for pre- and post-processes:** These are usually required in order to cope with the additional problems introduced by the hierarchical structure of basic block flow graphs (e.g., in *dead code elimination*, *constant propagation*,...); or which necessitate 'tricky' formulations to avoid them (e.g., in *partial redundancy elimination*).
- ▶ **Limited generality:** Some practically relevant program analyses and optimizations are difficult or not at all expressible on the level of basic block flow graphs (e.g., *faint variable elimination*).

Core Issue

...basic blocks cause a hierarchical graph structure:



Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1724/18

In the following

...we oppose advantages and disadvantages of

- ▶ basic block (BB) and single instructions (SI) graphs

considering DFA problems already discussed:

- ▶ Available expressions
- ▶ Simple constants

and new DFA problems:

- ▶ Faint variables

B.2

MOP and *MaxFP* Approach

B.2.1

Edge-labelled Instruction Graphs

Fixing the Setting

Let

- ▶ $G = (N, E, \mathbf{s}, \mathbf{e})$ be an edge-labelled SI flow graph.
- ▶ $\mathcal{S}_G =_{df} (\hat{C}, \llbracket \cdot \rrbracket_{E,l}, c_s, f_W)$ be a DFA specification.

The *MOP* Approach and *MOP* Solution

...for an edge-labelled single instruction flow graph.

Definition B.2.1.1 (The *MOP* Solution)

The $MOP_{E,\ell}$ solution of S_G is defined by:

$$MOP_{E,\ell}^{S_G} : N \rightarrow \mathcal{C}$$

$$\forall n \in N. MOP_{E,\ell}^{S_G}(n) =_{df} \bigsqcap \{ \llbracket p \rrbracket_{E,\ell}(c_s) \mid p \in \mathbf{P}[s, n] \}$$

The *MaxFP* Approach and *MaxFP* Solution

...for an edge-labelled single instruction flow graph.

Definition B.2.1.2 (The *MaxFP* Solution)

The *MaxFP*_{*E,l*} solution of \mathcal{S}_G is defined by:

$$\text{MaxFP}_{E,l}^{\mathcal{S}_G} : N \rightarrow \mathcal{C}$$

$$\forall n \in N. \text{MaxFP}_{E,l}^{\mathcal{S}_G}(n) =_{df} \nu\text{-inf}_{c_s}(n)$$

where $\nu\text{-inf}_{c_s}$ denotes the **greatest solution** of the *MaxFP* Equation System for instruction graphs:

$$\text{inf}(n) = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \bigcap \{ \llbracket (m, n) \rrbracket_{E,l}(\text{inf}(m)) \mid m \in \text{pred}(n) \} & \text{otherwise} \end{cases}$$

B.2.2

Node-labelled Basic Block Graphs

Fixing the Setting (1)

In the following we denote:

- ▶ basic block nodes by boldface letters ($\mathbf{m}, \mathbf{n}, \dots$)
- ▶ single instruction nodes by normalface letters (m, n, \dots)

We start from:

- ▶ $G = (N, E, \mathbf{s}, \mathbf{e})$, a node-labelled SI flow graph
- ▶ $\mathcal{S}_G = (\hat{\mathcal{C}}, \llbracket \rrbracket_{N, \ell}, c_s, fw)$, a DFA specification

which induce a node-labelled BB flow graph \mathbf{G} and a corresponding DFA specification \mathcal{S}_G .

Fixing the Setting (2)

Given G and \mathcal{S}_G , let

- ▶ $\mathbf{G} = (\mathbf{N}, \mathbf{E}, \mathbf{s}_G, \mathbf{e}_G)$
- ▶ $\mathcal{S}_G = (\widehat{\mathcal{C}}, \llbracket \rrbracket_{\mathbf{N},\beta}, c_s, fw)$

denote the node-labelled BB flow graph and the DFA specification induced by G and \mathcal{S}_G , respectively, where

- ▶ $\llbracket \rrbracket_{\mathbf{N},\beta} : \mathbf{N} \rightarrow \mathcal{C} \rightarrow \mathcal{C}$

denotes the extension of the SI DFA functional $\llbracket \rrbracket_{N,\ell}$ from nodes to basic blocks defined by

- ▶ $\forall \mathbf{n} = \langle n_{\ell_1}, \dots, n_{\ell_k} \rangle \in \mathbf{N}. \llbracket \mathbf{n} \rrbracket_{N,\beta} =_{df} \llbracket \langle n_1, \dots, n_k \rangle \rrbracket_{N,\ell}$

Auxiliary Mappings

- ▶ *bb*: maps a node n to the basic block \mathbf{n} it is included in.
- ▶ *start*: maps a basic block node \mathbf{n} to its entry node n .
- ▶ *end*: maps a basic block node \mathbf{n} to its exit node n .

The *MOP* Approach and *MOP* Solution (1)

...for a node-labelled basic block flow graph.

Definition B.2.2.1 (The *MOP* Solution, Part 1)

The $MOP_{\mathbf{N},\beta}$ solution of $\mathcal{S}_{\mathbf{G}}$ is defined by

$$MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}} : \mathbf{N} \rightarrow (\mathcal{C}, \mathcal{C})$$

$$\forall \mathbf{n} \in \mathbf{N}. MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}) =_{df} (N-MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}), X-MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}))$$

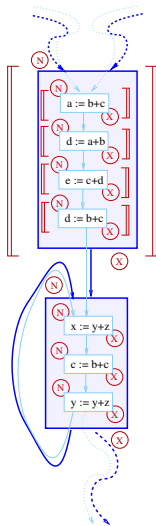
where

$$N-MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}) =_{df} \bigsqcap \{ \llbracket p \rrbracket_{\mathbf{N},\beta}(c_s) \mid p \in \mathbf{P}_{\mathbf{G}}[\mathbf{s}, \mathbf{n}] \}$$

$$X-MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}) =_{df} \bigsqcap \{ \llbracket p \rrbracket_{\mathbf{N},\beta}(c_s) \mid p \in \mathbf{P}_{\mathbf{G}}[\mathbf{s}, \mathbf{n}] \}$$

The *MOP* Approach and *MOP* Solution (2)

Entry (N) and exit (X) information for basic block nodes must be pushed inside of the basic blocks:



The *MOP* Approach and *MOP* Solution (3)

...and their push to the instruction level:

Definition B.2.2.1 (The *MOP* Solution, Part 2)

The *MOP*_{*N,l*} solution of \mathcal{S}_G is defined by

$$MOP_{N,l}^{\mathcal{S}_G} : N \rightarrow (\mathcal{C}, \mathcal{C})$$

$$\forall n \in N. MOP_{N,l}^{\mathcal{S}_G}(n) =_{df} (N\text{-}MOP_{N,l}^{\mathcal{S}_G}(n), X\text{-}MOP_{N,l}^{\mathcal{S}_G}(n))$$

The *MOP* Approach and *MOP* Solution (4)

where

$$N\text{-MOP}_{N,\iota}^{S_G}(n) \stackrel{df}{=} \begin{cases} N\text{-MOP}_{\mathbf{N},\beta}^{S_G}(bb(n)) \\ \quad \text{if } n = \text{start}(bb(n)) \\ \\ \llbracket p \rrbracket_{N,\iota}(N\text{-MOP}_{\mathbf{N},\beta}^{S_G}(bb(n))) \\ \quad \text{otherwise } (p \text{ is the prefix path from} \\ \quad \text{start}(bb(n)) \text{ up to but exclusive of } n) \end{cases}$$

$$X\text{-MOP}_{N,\iota}^{S_G}(n) \stackrel{df}{=} \llbracket p \rrbracket_{N,\iota}(N\text{-MOP}_{\mathbf{N},\beta}^{S_G}(bb(n))) \\ (p \text{ is the prefix path from } \text{start}(bb(n)) \text{ up} \\ \text{to and inclusive of } n)$$

The *MaxFP* Approach and *MaxFP* Solution (1)

...for a node-labelled basic block flow graph.

Definition B.2.2.2 (The *MaxFP* Solution, Part 1)

The *MaxFP*_{N,β} solution of \mathcal{S}_G is defined by

$$\forall \mathbf{n} \in \mathbf{N}. \text{MaxFP}_{\mathbf{N},\beta}^{\mathcal{S}_G}(\mathbf{n}) =_{df} (N\text{-MaxFP}_{\mathbf{N},\beta}^{\mathcal{S}_G}(\mathbf{n}), X\text{-MaxFP}_{\mathbf{N},\beta}^{\mathcal{S}_G}(\mathbf{n}))$$

with

$$N\text{-MaxFP}_{\mathbf{N},\beta}^{\mathcal{S}_G}(\mathbf{n}) =_{df} \nu\text{-pre}_{c_s,\beta}(\mathbf{n})$$

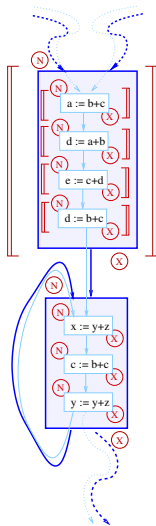
$$X\text{-MaxFP}_{\mathbf{N},\beta}^{\mathcal{S}_G}(\mathbf{n}) =_{df} \nu\text{-post}_{c_s,\beta}(\mathbf{n})$$

where $\nu\text{-pre}_{c_s,\beta}$ and $\nu\text{-post}_{c_s,\beta}$ denote the greatest solution of the *MaxFP* Equation System for basic block graphs:

$$\begin{aligned} \text{pre}(\mathbf{n}) &= \begin{cases} c_s & \text{if } \mathbf{n} = \mathbf{s} \\ \bigcap \{ \text{post}(\mathbf{m}) \mid \mathbf{m} \in \text{pred}_G(\mathbf{n}) \} & \text{otherwise} \end{cases} \\ \text{post}(\mathbf{n}) &= \llbracket \mathbf{n} \rrbracket_{\mathbf{N},\beta}(\text{pre}(\mathbf{n})) \end{aligned}$$

The *MaxFP* Approach and *MaxFP* Solution (2)

Entry (N) and exit (X) information for basic block nodes must be pushed inside of the basic blocks:



The *MaxFP* Approach and *MaxFP* Solution (3)

...and their push to the instruction level:

Definition B.2.2.2 (The *MaxFP* Solution, Part 2)

The *MaxFP*_{*N*,*l*} solution of \mathcal{S}_G is defined by

$$\forall n \in N. \text{MaxFP}_{N,l}^{\mathcal{S}_G}(n) =_{df} (N\text{-MaxFP}_{N,l}^{\mathcal{S}_G}(n), X\text{-MaxFP}_{N,l}^{\mathcal{S}_G}(n))$$

with

$$N\text{-MaxFP}_{N,l}^{\mathcal{S}_G}(n) =_{df} \nu\text{-pre}_{c_s,l}(n)$$

$$X\text{-MaxFP}_{N,l}^{\mathcal{S}_G}(n) =_{df} \nu\text{-post}_{c_s,l}(n)$$

The *MaxFP* Approach and *MaxFP* Solution (4)

...where $\nu\text{-pre}_{c_s, \ell}$ and $\nu\text{-post}_{c_s, \ell}$ denote the greatest solution of the *MaxFP* Equation System for instruction graphs:

$$\text{pre}(n) = \begin{cases} \nu\text{-pre}_{c_s, \beta}(bb(n)) & \text{if } n = \text{start}(bb(n)) \\ \text{post}(m) & \text{otherwise, where } m \text{ is the} \\ & \text{unique predecessor of } n \\ & \text{in } bb(n) \end{cases}$$
$$\text{post}(n) = \begin{cases} \nu\text{-post}_{c_s, \beta}(bb(n)) & \text{if } n = \text{end}(bb(n)) \\ \llbracket n \rrbracket_{N, \ell}(\text{pre}(n)) & \end{cases}$$

B.3

Available Expressions

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1742/18

B.3.1

Basic Block Graphs

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1743/18

Available Expressions (1)

...for node-labelled basic block graphs and a single term t .

Stage I: The Basic Block Level

Local Predicates (associated with basic block nodes):

- ▶ BB-XComp_β^t : β contains a statement ι computing t , and neither ι nor a statement following ι in β modifies an operand of t .
- ▶ BB-Transp_β^t : β does not contain a statement which modifies an operand of t .

The Basic Block *MaxFP* Equation System of Stage I:

$$\text{BB-N-Avail}_\beta = \begin{cases} c_s & \text{if } \beta = \mathbf{s}_G \\ \bigwedge_{\hat{\beta} \in \text{pred}(\beta)} \text{BB-X-Avail}_{\hat{\beta}} & \text{otherwise} \end{cases}$$

$$\text{BB-X-Avail}_\beta = (\text{BB-N-Avail}_\beta \wedge \text{BB-Transp}_\beta^t) \vee \text{BB-XComp}_\beta^t$$

Available Expressions (2)

Stage II: The Instruction Level

Local Predicates (associated with instruction nodes):

- ▶ Comp_ι^t : ι computes t .
- ▶ Transp_ι^t : ι does not modify an operand of t .
- ▶ $\nu\text{-BB-N-Avail}$, $\nu\text{-BB-X-Avail}$: the greatest solution of the *MaxFP* Equation System of Stage I.

Auxiliary Mappings

- ▶ bb : maps an instruction ι to the basic block β it is included in.
- ▶ $start$: maps a basic block β to its entry instruction ι .
- ▶ end : maps a basic block β to its exit instruction ι .

Available Expressions (3)

The Instruction *MaxFP* Equation System of Stage II:

$$\text{N-Avail}_\iota = \begin{cases} \nu\text{-BB-N-Avail}_{bb(\iota)} & \text{if } \iota = \text{start}(bb(\iota)) \\ \text{X-Avail}_{pred(\iota)} & \text{otherwise (note: } |pred(\iota)| = 1) \end{cases}$$

$$\text{X-Avail}_\iota = \begin{cases} \nu\text{-BB-X-Avail}_{bb(\iota)} & \text{if } \iota = \text{end}(bb(\iota)) \\ (\text{N-Avail}_\iota \vee \text{Comp}_\iota^t) \wedge \text{Transp}_\iota^t & \text{otherwise} \end{cases}$$

B.3.2

Node-labelled Instruction Graphs

Available Expressions

...for node-labelled instruction graphs and a single term t .

Local Predicates (associated with instruction nodes):

- ▶ Comp_ι^t : ι computes t .
- ▶ Transp_ι^t : ι does not modify an operand of t .

The Instruction *MaxFP* Equation System:

$$\text{N-Avail}_\iota = \begin{cases} c_s & \text{if } \iota = s \\ \bigwedge_{\hat{\iota} \in \text{pred}(\iota)} \text{X-Avail}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

$$\text{X-Avail}_\iota = (\text{N-Avail}_\iota \vee \text{Comp}_\iota^t) \wedge \text{Transp}_\iota^t$$

B.3.3

Edge-labelled Instruction Graphs

Available Expressions

...edge-labelled instruction graphs and a single term t .

Locale Predicates (associated with instruction edges):

- ▶ $\text{Comp}_\varepsilon^t$: Instruction ι of edge ε computes t .
- ▶ $\text{Transp}_\varepsilon^t$: Instruction ι of edge ε does not modify an operand of t .

The Instruction *MaxFP* Equation System:

$$\text{Avail}_n = \begin{cases} c_s & \text{if } n = \mathbf{s} \\ \bigwedge_{m \in \text{pred}(n)} (\text{Avail}_m \vee \text{Comp}_{(m,n)}^t) \wedge \text{Transp}_{(m,n)}^t & \\ \text{otherwise} & \end{cases}$$

B.3.4

Summary of Findings

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

G
1751/18

Findings

...edge-labelled instruction graphs are conceptually and notationally the most

- ▶ convenient ones.

...node-labelled basic block graphs the most

- ▶ inconvenient ones.

In the following

...we consider two more examples to illustrate the impact of selecting a flow graph variant on the conceptual and practical complexity of data flow analysis:

- ▶ Simple constants analysis (cf. Chap. B.4)
- ▶ Faint variables analysis (cf. Chap. B.5)

To this end we will oppose and investigate *MaxFP* formulations of these problems for

- ▶ node-labelled basic block graphs
- ▶ edge-labelled instruction graphs

which are the *antipodes* of each other.

B.4

Simple Constants

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

G 1754/18

Simple Constants Analysis

...for the formal problem formulation we require two auxiliary functions:

- ▶ Backward substitution δ
- ▶ State transformation θ

together with their extensions to (path) instruction sequences.

Backward Substitution, State Transformation

Let $\iota \equiv (x := t)$ be an instruction. We define:

- ▶ Backward substitution δ_ι

$\delta_\iota : \mathbf{T} \rightarrow \mathbf{T}$ defined by

$$\forall s \in \mathbf{T}. \delta_\iota(s) =_{df} s[t/x]$$

where $s[t/x]$ denotes the simultaneous replacement of all occurrences of x by t in s .

- ▶ State transformation θ_ι

$\theta_\iota : \Sigma \rightarrow \Sigma$ defined by

$$\forall \sigma \in \Sigma \forall v \in \mathbf{V}. \theta_\iota(\sigma)(v) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{if } v = x \\ \sigma(v) & \text{otherwise} \end{cases}$$

The Relationship of δ and θ

Let \mathcal{I} denote the set of all instructions.

Lemma B.4.1 (Substitution Lemma for Instructions)

$$\forall l \in \mathcal{I} \forall t \in \mathbf{T} \forall \sigma \in \Sigma. \mathcal{E}(\delta_l(t))(\sigma) = \mathcal{E}(t)(\theta_l(\sigma))$$

Proof by induction on the structure of t .

B.4.1

Edge-labelled Instruction Graphs

Simple Constants Analysis

...for an edge-labelled instruction graph.

The *MaxFP* Equation System for edge-lab. instruction graphs:

$$SC_n = \begin{cases} \sigma_s & \text{falls } n = s \\ \lambda v. \prod \{ \mathcal{E}(\delta_{(m,n)}(v))(SC_m) \mid m \in \text{pred}(n) \} & \text{sonst} \end{cases}$$

where $\sigma_s \in \Sigma$ start information.

The *Solution* of the *Simple Constants Analysis* is given by:

- ▶ $\nu\text{-SC} : N \rightarrow \Sigma$, the *greatest solution* of the above *EQS*.

B.4.2

Node-labelled Basic Block Graphs

Backward Substitution, State Transformation

...for paths.

Adapting and extending δ and θ from instructions to sequences of instructions on paths (and hence basic blocks) of node-labelled flow graphs:

► Backward Substitution on Path Instruction Sequences

$$\Delta_p : \mathbf{T} \rightarrow \mathbf{T}$$
$$\Delta_p =_{df} \begin{cases} \delta_{n_q} & \text{if } q = 1 \\ \Delta_{(n_1, \dots, n_{q-1})} \circ \delta_{n_q} & \text{if } q > 1 \end{cases}$$

► State Transformation on Path Instruction Sequences

$$\Theta_p : \Sigma \rightarrow \Sigma$$
$$\Theta_p =_{df} \begin{cases} \theta_{n_1} & \text{if } q = 1 \\ \Theta_{(n_2, \dots, n_q)} \circ \theta_{n_1} & \text{if } q > 1 \end{cases}$$

The Relationship of Δ and Θ

Let \mathcal{B} denote the set of all basic blocks.

Lemma B.4.2.1 (Substitution L. for Basic Blocks)

$$\forall \beta \in \mathcal{B} \forall t \in \mathbf{T} \forall \sigma \in \Sigma. \mathcal{E}(\Delta_\beta(t))(\sigma) = \mathcal{E}(t)(\Theta_\beta(\sigma))$$

Proof by induction on the length of β .

Simple Constants Analysis (1)

...for a node-labelled basic block graph.

Stage I: The Basic Block Level

The BB_N *MaxFP* Equation System of Stage I:

$$BB\text{-N-SC}_\beta = \begin{cases} \sigma_s & \text{if } \beta = \mathbf{s} \\ \prod\{BB\text{-X-SC}_{\hat{\beta}} \mid \hat{\beta} \in \text{pred}(\beta)\} & \text{otherwise} \end{cases}$$

$$BB\text{-X-SC}_\beta = \lambda v. \mathcal{E}(\Delta_\beta(v))(BB\text{-N-SC}_\beta)$$

where $\sigma_s \in \Sigma$ start information.

The **Solution** of the BB_N *SC* Analysis is given by:

- ▶ $\nu\text{-BB-N-SC}_\beta, \nu\text{-BB-X-SC}_\beta : \mathbf{N} \rightarrow \Sigma$, the greatest solutions of the above equation system.

Simple Constants Analysis (2)

Stage II: The Instruction Level

Auxiliary Mappings

- ▶ *bb*: maps an instruction ι to the basic block β it is included in.
- ▶ *start*: maps a basic block β to its entry instruction ι .
- ▶ *end*: maps a basic block β to its exit instruction ι .

The SI_N MaxFP Equation System of Stage II:

$$\text{N-SC}_\iota = \begin{cases} \nu\text{-BB-N-SC}_{bb(\iota)} & \text{if } \iota = \text{start}(bb(\iota)) \\ \text{X-SC}_{pred(\iota)} & \text{otherwise (because} \\ & |pred(\iota)| = 1) \end{cases}$$

$$\text{X-SC}_\iota = \begin{cases} \nu\text{-BB-X-SC}_{bb(\iota)} & \text{if } \iota = \text{end}(bb(\iota)) \\ \lambda v. \mathcal{E}(\delta_\iota(v))(\text{N-CP}_\iota) & \text{otherwise} \end{cases}$$

Simple Constants Analysis (3)

The **Solution** of the SI_N SC Analysis is given by:

- ▶ ν -N-SC, ν -X-SC : $N \rightarrow \Sigma$, the **greatest solution** of the preceding equation system.

B.5

Faint Variables

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

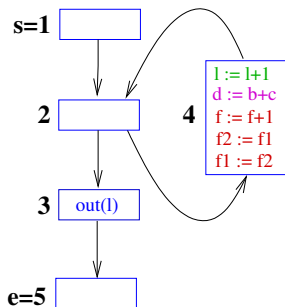
Chap. 13

Part IV

G 1766/18

Faint Variables: Between Life and Death

...consider the program:



Note: Instruction

- ▶ $l := l + 1$ is live,
- ▶ $d := b + c$ is dead,
- ▶ $f := f + 1$, $f1 := f2$, and $f2 := f1$ are live but faint (in German: geisterhaft, schattenhaft).

Faint Variables Analysis (1)

...for an edge-labelled instruction graphs.

Local Predicates (associated with instruction edges):

- ▶ $\text{LifeEnforcingUse}_\varepsilon^v$: Variable v is used by the instruction ι associated with edge ε and 'forced to live' by it (i.e., ι is an output or test operation).
- ▶ Mod_ε^v : The instruction ι at edge ε modifies variable v .
- ▶ $\text{Ass-Used}_\varepsilon^v$: Variable v , occurs in the right-hand side expression of the instruction ι associated with edge ε .

Auxiliary Mapping

- ▶ LhsVar : Maps an edge ε to the left-hand side variable of the instruction ι associated with it.

Faint Variables Analysis (2)

The SI_E MaxFP Equation System:

$\text{Faint}_n^v =$

$$\left\{ \begin{array}{ll} fv_e & \text{if } n = e \\ \bigwedge_{m \in \text{succ}(n)} \neg \text{LifeEnforcingUse}_{(n,m)}^v \wedge \\ & (\text{Faint}_m^v \vee \text{Mod}_{(n,m)}^v) \wedge \\ & (\text{Faint}_m^{\text{LhsVar}_{(n,m)}} \vee \neg \text{Ass-Used}_{(n,m)}^v) & \text{otherwise} \end{array} \right.$$

where $fv_e \in \mathbb{B}^{|V|}$ start information.

The Solution of the SI_E Faint Variables Analysis is given by:

- ▶ $\nu\text{-Faint} : N \rightarrow \mathbb{B}^{|V|}$, the greatest solution of the above equation system.

Informally

...a variable v is **faint** at node n , if v

- ▶ is not forced to live by an instruction at an incoming edge of n (**1-st conjunction term**).
- ▶ is already faint at node n or modified by an instruction at an incoming edge of n and thereby made faint (**2-nd conjunction term**).
- ▶ is not used by an instruction at an incoming edge of n or (at most) used to assign a new value to a variable which is faint itself (**3-rd conjunction term**).

Summing up

Faint variables are an example of a (so-called) **non-separable DFA problem**, where a formulation leading to an **efficient implementation** is

- ▶ **obvious** for (node and edge-labelled) **instruction graphs**,
- ▶ **not at all obvious**, if not impossible at all, for (node and edge-labelled) **basis block graphs**.

(Note that the naive straightforward extension to **basic block graphs** would require for **every basic block n** to compute the full semantic function $\llbracket n \rrbracket_{faint} : \mathbb{B}^{k_n} \rightarrow \mathbb{B}^{k_n}$, where k_n is the number of variables occurring in n , a function with 2^{k_n} arguments. In the worst case, k_n coincides even with the number of all variables in the program under consideration.)

B.6

Conclusions

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

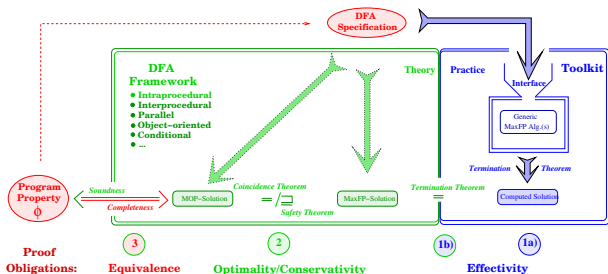
Part IV

G 1772/18

Conclusion

All 4 flow graph variants are conceptually essentially equivalent with in most cases only minor pragmatic advantages and disadvantages.

Thus the general holistic framework and tool kit view of DFA



is conceptually adequate and sufficient when being aware of the differences and their impact on specification, implementation, and proof obligation accomplishment.

B.7

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11




Chap. 12

Chap. 13

Part IV

G 1774/18

Further Reading for Appendix B

-  Larry Carter, Jeanne Ferrante, Clark Thomborson. *Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger*. In Conference Record of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 106-114, 2003.
-  Jens Knoop. *From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Springer-V., LNCS 1579, 360-374, 1999.
-  Jens Knoop, Dirk Koschützki, Bernhard Steffen. *Basic-block Graphs: Living Dinosaurs?* In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 65 - 79, 1998.

Appendix C

Implementing Busy and Lazy Code Motion on SI and BB Flow Graphs

C.1

Implementing *BCM* and *LCM* on SI Graphs

C.1.1

Preliminaries

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

C.1778/18

Busy and Lazy Code Motion

...for node-labelled SI graphs:

- ▶ BCM_ℓ transformation
- ▶ LCM_ℓ transformation

Convention: For the following we assume that only critical edges are split. Therefore, BCM_ℓ and LCM_ℓ require insertions at both node entries and node exits (**N-insertions** and **X-insertions**).

Local Predicates for BCM_ι and LCM_ι

Local Predicates:

- ▶ $COMP_\iota(t)$: t is computed by ι .
- ▶ $TRANSP_\iota(t)$: No operand of t is modified by ι .

C.1.2

Implementing BCM_t

Implementing BCM_ι (1)

1. Analyses for Up-Safety and Down-Safety

The *MaxFP*-Equation System for Up-Safety:

$$\text{N-USAFE}_\iota = \begin{cases} \text{false} & \text{if } \iota = \mathbf{s} \\ \prod_{\hat{\iota} \in \text{pred}(\iota)} \text{X-USAFE}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

$$\text{X-USAFE}_\iota = (\text{N-USAFE}_\iota + \text{COMP}_\iota) \cdot \text{TRANSP}_\iota$$

The *MaxFP*-Equation System for Down-Safety:

$$\text{N-DSAFE}_\iota = \text{COMP}_\iota + \text{X-DSAFE}_\iota \cdot \text{TRANSP}_\iota$$

$$\text{X-DSAFE}_\iota = \begin{cases} \text{false} & \text{if } \iota = \mathbf{e} \\ \prod_{\hat{\iota} \in \text{succ}(\iota)} \text{N-DSAFE}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

Implementing BCM_ι (2)

2. The Transformation: Insertion and Replacement Points

Local Predicates:

- ▶ $N\text{-USAFE}^*$, $X\text{-USAFE}^*$, $N\text{-DSAFE}^*$, $X\text{-DSAFE}^*$: ...denote the greatest solutions of the equation systems for **up-safety** and **down-safety** of step 1.

Computing Earliestness (no data flow analysis!):

$$N\text{-EARLIEST}_\iota =_{df} N\text{-DSAFE}_\iota^* \cdot \prod_{\hat{\iota} \in \text{pred}(\iota)} (\overline{X\text{-USAFE}_{\hat{\iota}}^* + X\text{-DSAFE}_{\hat{\iota}}^*})$$

$$X\text{-EARLIEST}_\iota =_{df} X\text{-DSAFE}_\iota^* \cdot \overline{\text{TRANSP}_\iota}$$

Implementing BCM_ℓ (3)

The BCM_ℓ Transformation:

$N\text{-INSERT}_\ell^{\text{BCM}} =_{df} N\text{-EARLIEST}_\ell$

$X\text{-INSERT}_\ell^{\text{BCM}} =_{df} X\text{-EARLIEST}_\ell$

$\text{REPLACE}_\ell^{\text{BCM}} =_{df} \text{COMP}_\ell$

C.1.3

Implementing LCM_ι

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1785/18

Implementing LCM_ι (1)

3. Analyses for Delayability and Isolation

The *MaxFP*-Equation System for Delayability:

$$\text{N-DELAYED}_\iota = \text{N-EARLIEST}_\iota + \begin{cases} \text{false} & \text{if } \iota = \mathbf{s} \\ \prod_{\iota' \in \text{pred}(\iota)} \text{X-DELAYED}_{\iota'} & \text{otherwise} \end{cases}$$

$$\text{X-DELAYED}_\iota = \text{X-EARLIEST}_\iota + \text{N-DELAYED}_\iota \cdot \overline{\text{COMP}_\iota}$$

Note: The greatest solution of the above equation system equals directly the *Delayed* property of [Definition 8.2.1](#) at the expense of being 'locally bidirectional' and hence lacking a *MOP* characterization.

Implementing LCM_{ι} (2)

Computing Latestness (no data flow analysis!):

$$\text{N-LATEST}_{\iota} =_{df} \text{N-DELAYED}_{\iota}^* \cdot \text{COMP}_{\iota}$$

$$\text{X-LATEST}_{\iota} =_{df} \text{X-DELAYED}_{\iota}^* \cdot \sum_{\iota' \in \text{succ}(\iota)} \overline{\text{N-DELAYED}_{\iota'}^*}$$

where

- ▶ $\text{N-DELAYED}_{\iota}^*$, $\text{X-DELAYED}_{\iota}^*$: ...denote the greatest solutions of the equation system for **delayability**.

Implementing LCM_ι (3)

The $ALCM_\iota$ Transformation:

$N\text{-INSERT}_\iota^{ALCM} =_{df} N\text{-LATEST}_\iota$

$X\text{-INSERT}_\iota^{ALCM} =_{df} X\text{-LATEST}_\iota$

$REPLACE_\iota^{ALCM} =_{df} COMP_\iota$

Implementing LCM_ι (4)

The *MaxFP*-Equation System for Isolation:

$$\text{N-ISOLATED}_\iota = \text{X-EARLIEST}_\iota + \text{X-ISOLATED}_\iota$$

$$\text{X-ISOLATED}_\iota = \prod_{\iota' \in \text{succ}(\iota)} \text{N-EARLIEST}_{\iota'} + \overline{\text{COMP}_{\iota'}} \cdot \text{N-ISOLATED}_{\iota'}$$

Note: The greatest solution of the above equation system equals directly the *Isolated* property of [Definition 8.3.1](#) and [Lemma 8.3.2](#) at the expense of a non-standard definition of the exit property.

Implementing LCM_ι (5)

4. The Transformation: Insertion and Replacement Points

Local Predicates:

- ▶ $N\text{-ISOLATED}^*$, $X\text{-ISOLATED}^*$: ...denote the greatest solutions of the equation system for *isolation* of [step 3](#).

The LCM_ι Transformation:

$$N\text{-INSERT}_\iota^{\text{LCM}} =_{df} N\text{-LATEST}_\iota \cdot \overline{N\text{-ISOLATED}_\iota^*}$$

$$X\text{-INSERT}_\iota^{\text{LCM}} =_{df} X\text{-LATEST}_\iota$$

$$\text{REPLACE}_\iota^{\text{LCM}} =_{df} \text{COMP}_\iota \cdot \overline{N\text{-LATEST}_\iota \cdot N\text{-ISOLATED}_\iota^*}$$

C.2

Implementing *BCM* and *LCM* on BB Graphs

C.2.1

Preliminaries

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

C

Implementing Busy and Lazy Code Motion

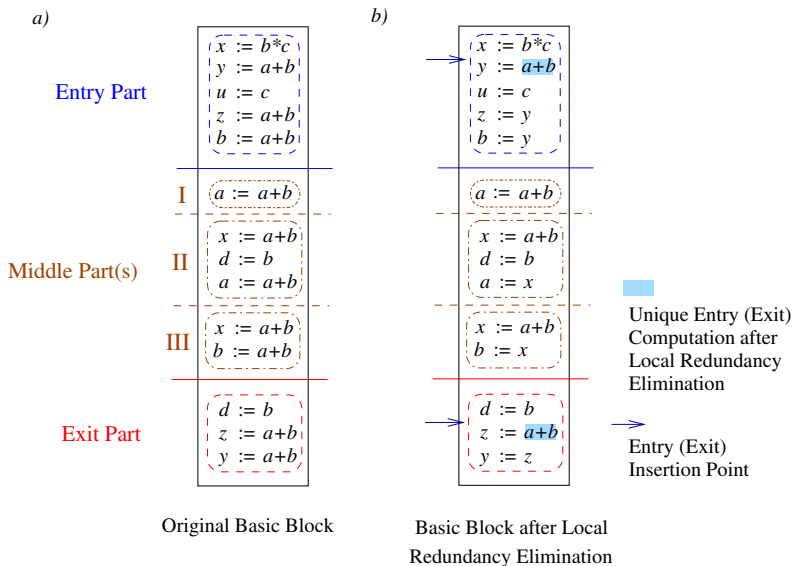
...for node-labelled BB graphs:

- ▶ BCM_{β} Transformation
- ▶ LCM_{β} Transformation

Convention: For the following we assume that (1) only critical edges are split. Therefore, BCM_{β} and LCM_{β} require insertions at both node entries and node exits (**N-insertions** and **X-insertions**), and that (2) all redundancies within a basic block have been removed by a preprocess.

Conceptual Splitting of a Basic Block

...into **entry**, **middle**, and **exit** part.



Entry and Exit Parts of a Basic Block

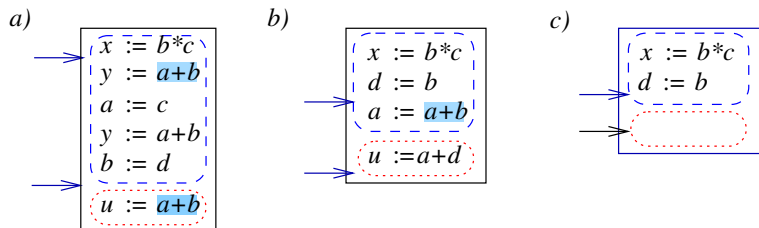
For **PRE**, we do not need to distinguish between **entry** and **middle part(s)**, and can consider them a unit. This gives rise to the following definition:

Given a computation t , a basic block n can be divided into two parts:

- ▶ an **entry part** which consists of all statements up to and including the last modification of t .
- ▶ an **exit part** which consists of the remaining statements of n .

Note: The entry part of a non-empty basic block is always non-empty; in distinction, the exit part of a non-empty basic block can be empty (as illustrated in the following figure).

Illustrating Entry & Exit Part of a Basic Block



- - - Entry Part

..... Exit Part

■ Entry (Exit) Computation

➔ Entry (Exit) Insertion Point

The General Pattern of CM on BB Graphs

1. Introducing temporary

1.1 Define a new temporary variable \mathbf{h}_{CM} for t .

2. Insertions

2.1 Insert assignments $\mathbf{h}_{CM} := t$ at the insertion point of the entry part of all $\beta \in \mathbf{N}$ satisfying N-INSERT^{CM}

2.2 Insert assignments $\mathbf{h}_{CM} := t$ at the insertion point of the exit part of all $\beta \in \mathbf{N}$ satisfying X-INSERT^{CM}

3. Replacements

3.1 Replace the (unique) entry computation of t by \mathbf{h}_{CM} in every $\beta \in \mathbf{N}$ satisfying N-REPLACE^{CM}

3.2 Replace the (unique) exit computation of t by \mathbf{h}_{CM} in every $\beta \in \mathbf{N}$ satisfying X-REPLACE^{CM}

Local Predicates for BCM_β and LCM_β

Local Predicates:

- ▶ $BB-NComp_\beta(t)$: β contains a statement ι that computes t , and that is not preceded by a statement that modifies an operand of t .
- ▶ $BB-XComp_\beta(t)$: β contains a statement ι that computes t and neither ι nor any other statement of β after ι modifies an operand of t .
- ▶ $BB-Transp_\beta(t)$: β contains no statement that modifies an operand of t .

C.2.2

Implementing BCM_{β}

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1799/18

Implementing BCM_{β} (1)

1. Analyses for Up-Safety and Down-Safety

The *MaxFP*-Equation System for Up-Safety:

$$BB\text{-N-USAFF}_{\beta} = \begin{cases} false & \text{if } \beta = \mathbf{s} \\ \prod_{\hat{\beta} \in pred(\beta)} (BB\text{-XComp}_{\hat{\beta}} + BB\text{-X-USAFF}_{\hat{\beta}}) & \text{otherwise} \end{cases}$$

$$BB\text{-X-USAFF}_{\beta} = (BB\text{-N-USAFF}_{\beta} + BB\text{-NComp}_{\beta}) \cdot BB\text{-Transp}_{\beta}$$

The *MaxFP*-Equation System for Down-Safety:

$$BB\text{-N-DSAFF}_{\beta} = BB\text{-NComp}_{\beta} + BB\text{-X-DSAFF}_{\beta} \cdot BB\text{-Transp}_{\beta}$$

$$BB\text{-X-DSAFF}_{\beta} = BB\text{-XComp}_{\beta} + \begin{cases} false & \text{if } \beta = \mathbf{e} \\ \prod_{\hat{\beta} \in succ(\beta)} BB\text{-N-DSAFF}_{\hat{\beta}} & \text{otherwise} \end{cases}$$

Implementing BCM_{β} (2)

2. The Transformation: Insertion and Replacement Points

Local Predicates:

- ▶ $BB-N-USAFE^*$, $BB-X-USAFE^*$, $BB-N-DSAFE^*$, $BB-X-DSAFE^*$: ...denote the greatest solutions of the equation systems for **up-safety** and **down-safety** of **step 1**.

Computing Earliestness (no data flow analysis!):

$$N-EARLIEST_{\beta} =_{df} BB-N-DSAFE_{\beta}^* \cdot \prod_{\hat{\beta} \in pred(\beta)} (\overline{BB-X-USAFE_{\hat{\beta}}^* + BB-X-DSAFE_{\hat{\beta}}^*})$$

$$X-EARLIEST_{\beta} =_{df} BB-X-DSAFE_{\beta}^* \cdot \overline{BB-Transp_{\beta}}$$

Implementing BCM_{β} (3)

The BCM_{β} Transformation:

$N\text{-INSERT}_{\beta}^{BCM} =_{df} N\text{-EARLIEST}_{\beta}$

$X\text{-INSERT}_{\beta}^{BCM} =_{df} X\text{-EARLIEST}_{\beta}$

$N\text{-REPLACE}_{\beta}^{BCM} =_{df} BB\text{-NComp}_{\beta}$

$X\text{-REPLACE}_{\beta}^{BCM} =_{df} BB\text{-XComp}_{\beta}$

C.2.3

Implementing LCM_{β}

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1803/18

Implementing LCM_{β} (1)

3. Analyses for Delayability and Isolation

The *MaxFP*-Equation System for Delayability:

$$\text{N-DELAYED}_{\beta} = \text{N-EARLIEST}_{\beta} + \begin{cases} \text{false} & \text{if } \beta = \mathbf{s} \\ \prod_{\hat{\beta} \in \text{pred}(\beta)} \overline{\text{BB-XComp}_{\hat{\beta}}} \cdot \text{X-DELAYED}_{\hat{\beta}} & \text{otherwise} \end{cases}$$

$$\text{X-DELAYED}_{\beta} = \text{X-EARLIEST}_{\beta} + \text{N-DELAYED}_{\beta} \cdot \overline{\text{BB-NComp}_{\beta}}$$

Implementing LCM_{β} (2)

Computing Latestness (no data flow analysis!):

$$\text{N-LATEST}_{\beta} =_{df} \text{N-DELAYED}_{\beta}^* \cdot \text{BB-NComp}_{\beta}$$

$$\text{X-LATEST}_{\beta} =_{df} \text{X-DELAYED}_{\beta}^* \cdot \left(\text{BB-XComp}_{\beta} + \sum_{\hat{\beta} \in \text{succ}(\beta)} \overline{\text{N-DELAYED}_{\hat{\beta}}^*} \right)$$

where

- ▶ N-DELAYED^* , X-DELAYED^* : ...denote the greatest solutions of the equation system for [delayability](#).

Implementing LCM_{β} (3)

The $ALCM_{\beta}$ Transformation:

$$\text{N-INSERT}_{\beta}^{\text{ALCM}} =_{df} \text{N-LATEST}_{\beta}$$

$$\text{X-INSERT}_{\beta}^{\text{ALCM}} =_{df} \text{X-LATEST}_{\beta}$$

$$\text{N-REPLACE}_{\beta}^{\text{ALCM}} =_{df} \text{BB-NComp}_{\beta}$$

$$\text{X-REPLACE}_{\beta}^{\text{ALCM}} =_{df} \text{BB-XComp}_{\beta}$$

Implementing LCM_{β} (4)

The *MaxFP*-Equation System for Isolation:

$$\text{N-ISOLATED}_{\beta} = \text{X-EARLIEST}_{\beta} + \text{X-ISOLATED}_{\beta}$$

$$\text{X-ISOLATED}_{\beta} = \prod_{\hat{\beta} \in \text{succ}(\beta)} \text{N-EARLIEST}_{\hat{\beta}} + \overline{\text{BB-NComp}_{\hat{\beta}}} \cdot \text{N-ISOLATED}_{\hat{\beta}}$$

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1807/18

Implementing LCM_{β} (5)

4. The Transformation: Insertion and Replacement Points

Local Predicates:

- ▶ $N\text{-ISOLATED}^*$, $X\text{-ISOLATED}^*$: ...denote the greatest solutions of the equation system for **isolation** of **step 3**.

The LCM_{β} Transformation:

$$N\text{-INSERT}_{\beta}^{\text{LCM}} =_{df} N\text{-LATEST}_{\beta} \cdot \overline{N\text{-ISOLATED}_{\beta}^*}$$

$$X\text{-INSERT}_{\beta}^{\text{LCM}} =_{df} X\text{-LATEST}_{\beta} \cdot \overline{X\text{-ISOLATED}_{\beta}^*}$$

$$N\text{-REPLACE}_{\beta}^{\text{LCM}} =_{df} \text{BB-NComp}_{\beta} \cdot \overline{N\text{-LATEST}_{\beta} \cdot N\text{-ISOLATED}_{\beta}^*}$$

$$X\text{-REPLACE}_{\beta}^{\text{LCM}} =_{df} \text{BB-XComp}_{\beta} \cdot \overline{X\text{-LATEST}_{\beta} \cdot X\text{-ISOLATED}_{\beta}^*}$$

C.3

Illustrating Example

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

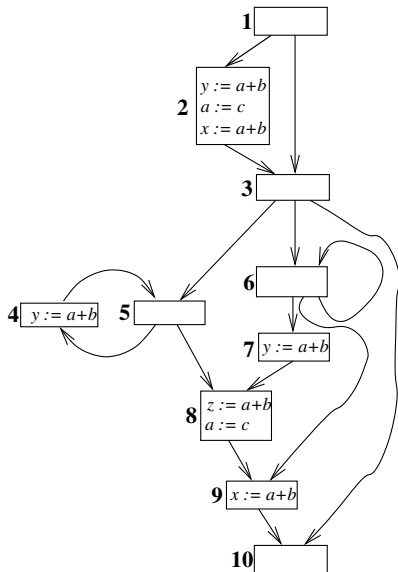
Chap. 12

Chap. 13

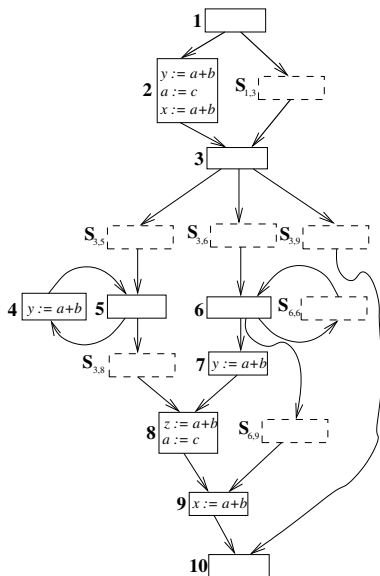
Part IV

1809/18

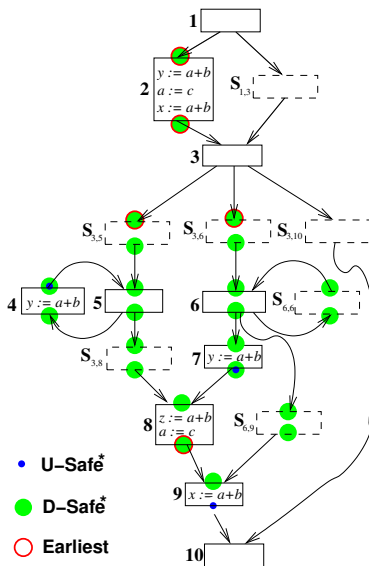
The Original Program



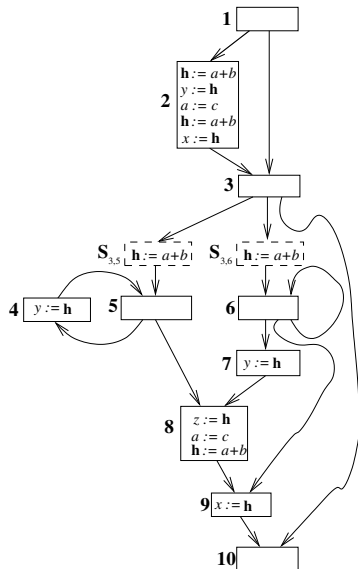
After the Splitting of Critical Edges



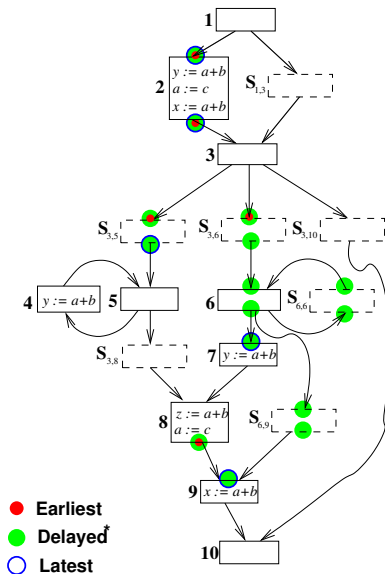
Up/Down-Safe, Earliest Computation Points



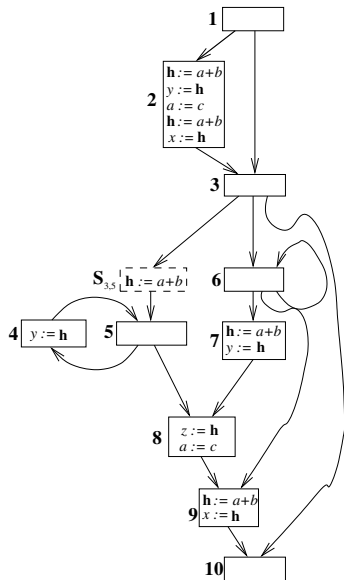
The Result of the BCM_{β} Transformation



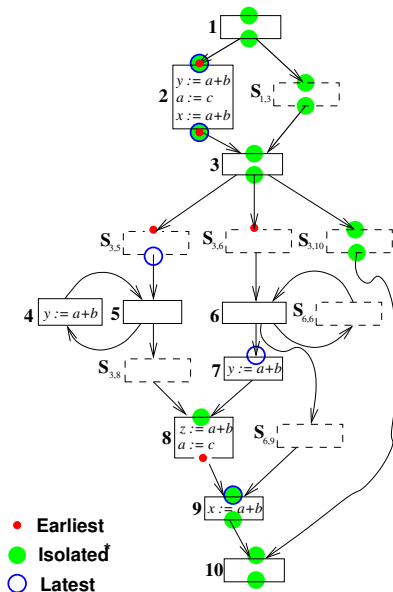
Delayable and Latest Computation Points



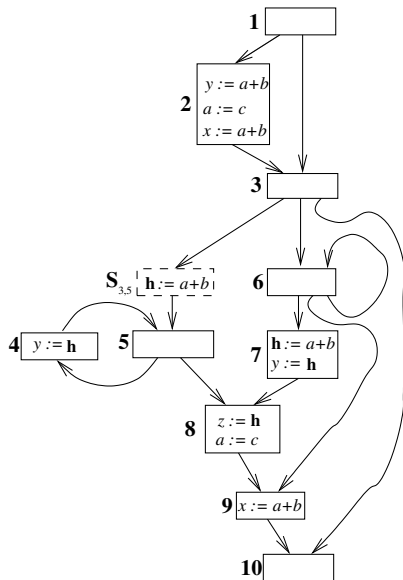
The Result of the $ALCM_{\beta}$ -Transformation



Latest and Isolated Program Points



The Result of the LCM_{β} Transformation



C.4

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11



Chap. 12

Chap. 13

Part IV

1818/18

Further Reading for Appendix C

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.

Appendix D

Lazy Strength Reduction

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

1820/18

D.1

Motivation

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

G 1821/18

From *BCM* to *LSR*

...from *busy code motion* to *lazy strength reduction*.

Objective: Developing a *program optimization* which

- ▶ uniformly covers
 - ▶ Partial Redundancy Elimination (PRE).
 - ▶ Strength Reduction (SR).
- ▶ avoids *superfluous register pressure* caused by *unnecessary code motion*.
- ▶ requires only *uni-directional data flow analyses*.

The Approach

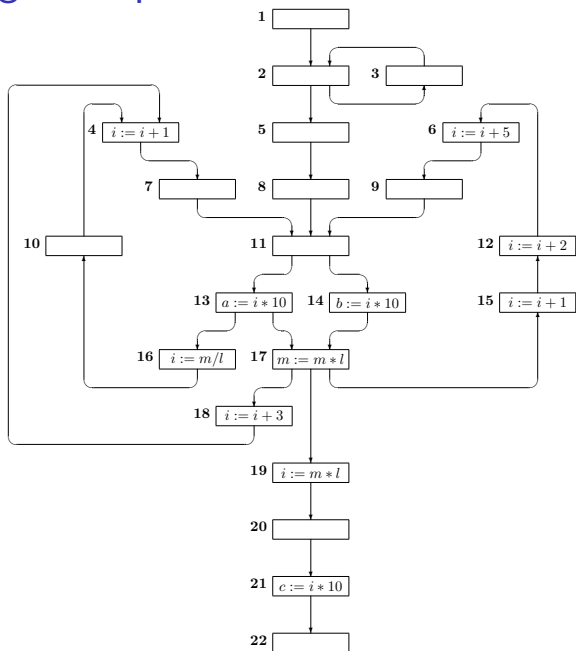
We will **stepwise** and **modularly** extend

- ▶ the *BCM* and the *LCM* to arrive at the
 - ▶ Busy Strength Reduction (*BSR*)
 - ▶ Lazy Strength Reduction (*LSR*)

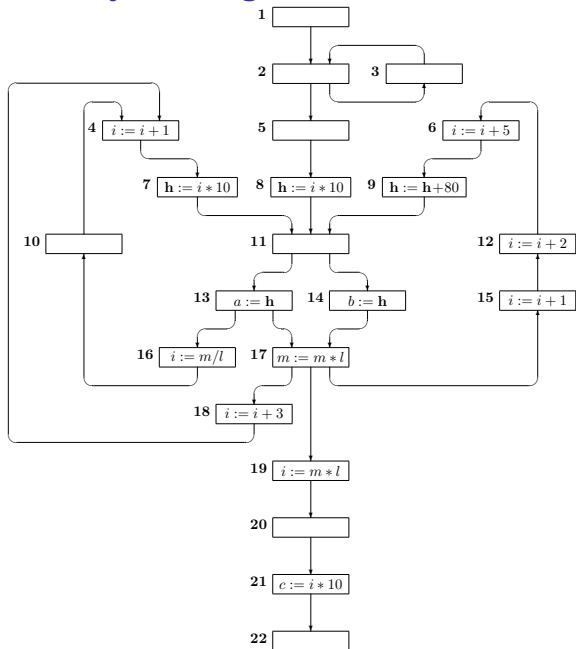
D.2

Running Example

The Running Example



The Result of Lazy Strength Reduction



D.3

Preliminaries

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11

Chap. 12

Chap. 13

Part IV

G 1827/18

Candidate Expressions

...for lazy strength reduction.

Candidate expressions for

- ▶ Code motion: No restrictions, every term $t \in \mathbf{T}$
- ▶ Lazy strength reduction: Every term of the form $v * c \in \mathbf{T}$, where
 - ▶ v is a variable
 - ▶ c is a source code constant

Local Predicates for Lazy Strength Reduction

Local predicates for lazy strength reduction

- ▶ $Used(n) =_{df} v * c \in SubTerms(t)$
- ▶ $Transp(n) =_{df} x \not\equiv v$
- ▶ $SR-Transp(n) =_{df} Transp(n) \vee t \equiv v + d$ with $d \in \mathbf{C}$

Note: The value of a candidate expression $v * c$ is

- ▶ **killed** at a node n , if $\neg(Transp(n) \vee SR-Transp(n))$

while it is

- ▶ **injured** at a node n , if $\neg Transp(n) \wedge SR-Transp(n)$

The Essence of Strength Reduction

Values which are **injured** but **not killed** can be

- ▶ **cured** by inserting an **update instruction** of the form

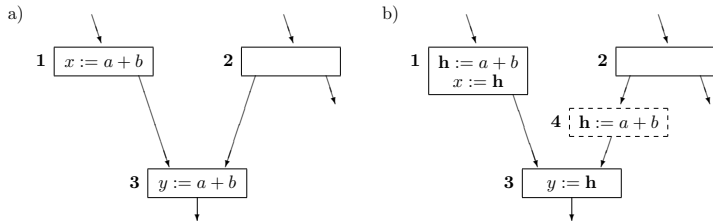
$$\mathbf{h} := \mathbf{h} + \mathit{Cure}(n), \text{ where } \mathit{Cure}(n) =_{df} c * d.$$

Note that the (**correction**) value of $\mathit{Cure}(n)$ can be computed at **compile time** since both c and d are **source code constants**.

Splitting of Critical Edges

As common for code motion transformations like *BCM* and *LCM*, *critical edges* need to be *split* in order to get the full power of

► Lazy Strength Reduction (LSR)



Splitting of Join Edges

In order to even allow *insertions* of statements

- ▶ uniformly at node entries

we assume that even all

- ▶ *join edges* (and not just critical edges)

are *split* as was assumed for the *BCM* and *LCM* transformations (cf. [Chap. 7](#) and [8](#)).

D.4

Extending *BCM* to Strength Reduction: The *BSR* Transformation

Extending *BCM* to Strength Reduction

...straightforward leads to *Busy Strength Reduction (BSR)*.

The *BSR* Transformation

1. Introduce a new auxiliary variable \mathbf{h} for $v * c$
2. Insert at the entry of every node n satisfying
 - 2.1 Ins_{BSR} the assignment $\mathbf{h} := v * c$
 - 2.2 $\text{InsUpd}_{\text{BSR}}$ the assignment $\mathbf{h} := \mathbf{h} + \text{Cure}(n)$
3. Replace every (original) occurrence of $v * c$ in G by \mathbf{h}

Note: If Ins_{BSR} and $\text{InsUpd}_{\text{BSR}}$ hold both at some node n , the initialization statement $\mathbf{h} := v * c$ must precede the update instruction $\mathbf{h} := \mathbf{h} + \text{Cure}(n)$.

Shortcomings of *BSR*

Shortcoming

- ▶ *BSR* can suffer from multiplication-addition-sequences.

Remedy

- ▶ Moving insertions at $(*, +)$ -critical insertion points in the direction of the control flow to the 'earliest' non-critical ones.

Note

- ▶ An insertion point is $(*, +)$ -critical if there is a $v * c$ -free program path from this point to a modification site of v .

D.5

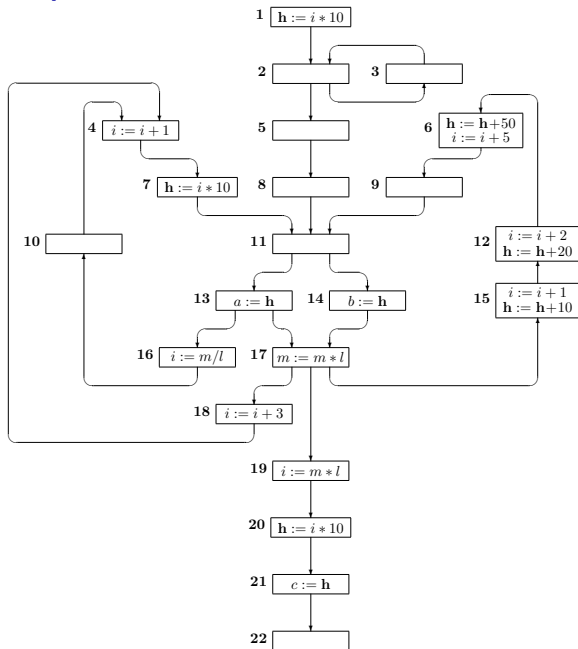
The 1st Refinement: Avoiding Multiplication-Addition Sequences – The BSR_{FstRef} Transformation

BSR_{FstRef} : Avoiding $(*, +)$ Sequences

The BSR_{FstRef} Transformation

1. Introduce a new auxiliary variable \mathbf{h} for $v * c$
2. Insert at the entry of every node n satisfying
 - 2.1 Ins_{FstRef} the assignment $\mathbf{h} := v * c$
 - 2.2 $InsUpd_{FstRef}$ the assignment $\mathbf{h} := \mathbf{h} + Cure(n)$
3. Replace every (original) occurrence of $v * c$ in G by \mathbf{h}

Running Example: The Result of BSR_{FstRef}



Shortcomings of BSR_{FstRef}

Shortcoming

- ▶ BSR_{FstRef} can suffer from unnecessary register pressure due to unnecessary code motion.

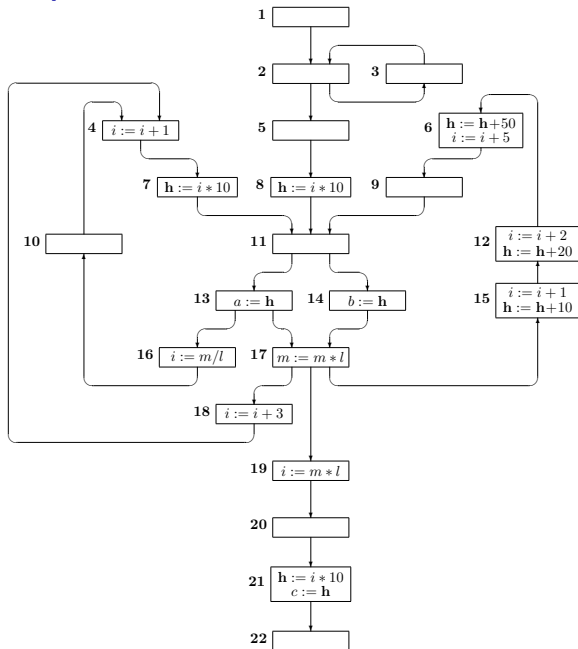
Remedy

- ▶ Adding laziness to BSR_{FstRef} .

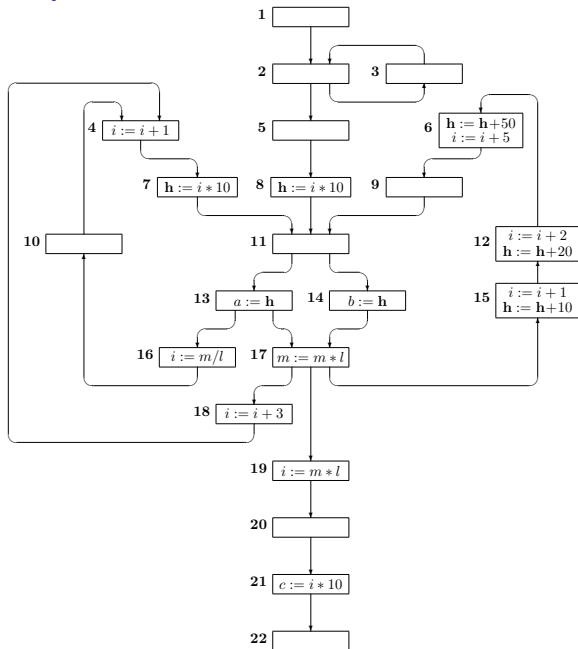
D.6

The 2nd Refinement: Avoiding Unnecessary Register Pressure – The BSR_{SndRef} Transformation

Running Example: Pre-Result of BSR_{SndRef}



Running Example: The Result of BSR_{SndRef}



Shortcomings of BSR_{SndRef}

Shortcoming

- ▶ BSR_{SndRef} can suffer from multiple-addition sequences.

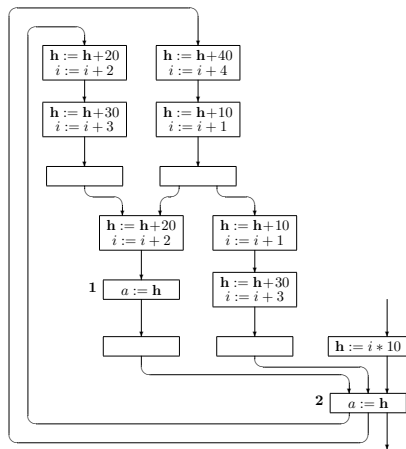
Remedy

- ▶ Replacing of multiple-addition sequences by a single cumulated addition instruction.

Note

- ▶ The resulting third refinement of the BSR transformation, BSR_{ThdRef} , defines the Lazy Strength Reduction (LSR), i.e., $LSR =_{df} BSR_{ThdRef}$.

Illustrating Multiple-Addition Sequences

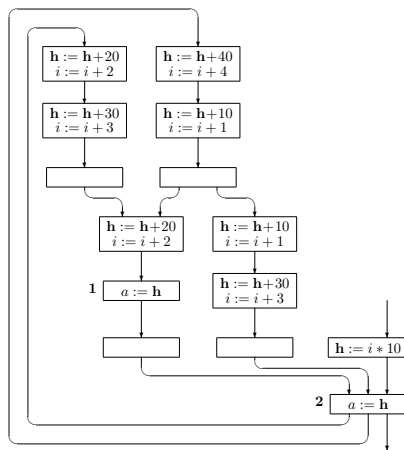


D.7

The 3rd Refinement: Avoiding Multiple-Addition Sequences – The ($BSR_{ThdRef} \equiv$) *LSR* Transformation

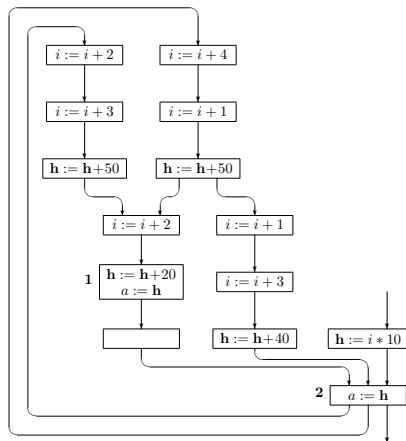
Illustrating Example

...suffering from **multiple-addition sequences**:



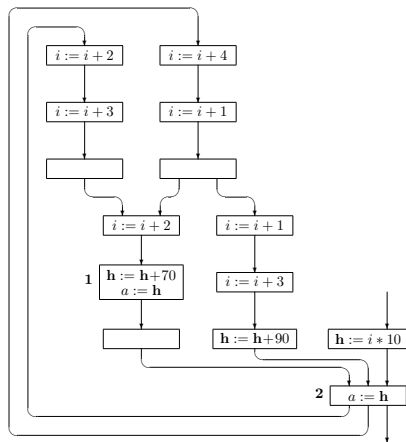
Avoiding Multiple-Addition Sequences (1)

...basic accumulation of the effect of cure instructions across basic blocks:

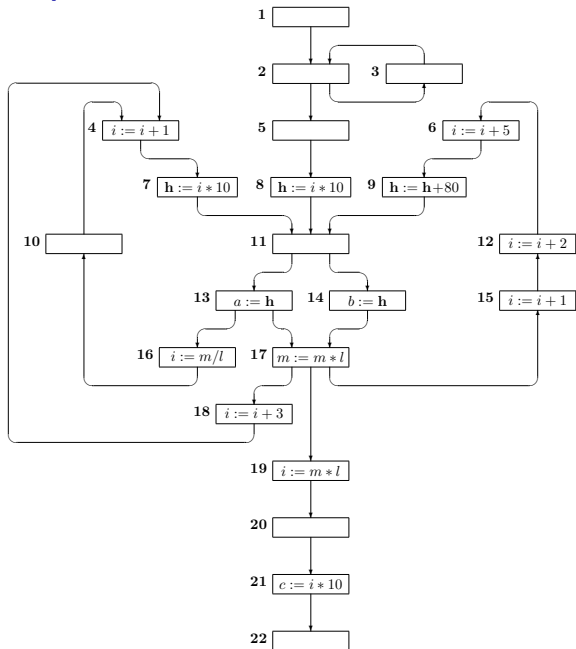


Avoiding Multiple-Addition Sequences (2)

...refined accumulation of the effect of cure instructions across extended basic blocks:



Running Example: The Result of *LSR*



Exercise D.7.1

1. Specify the DFAs and transformations required for
 - ▶ BSR
 - ▶ BSR_{FstRef} (avoiding multiplication-addition sequences)
 - ▶ BSR_{SndRef} (avoiding unnecessary register pressure)
 - ▶ $LSR =_{df} BSR_{ThdRef}$ (avoiding multiple-addition sequ.)
2. Implement the DFAs in PAG.
3. Validate the DFAs on the running example of Appendix D (or an example coming close to it).

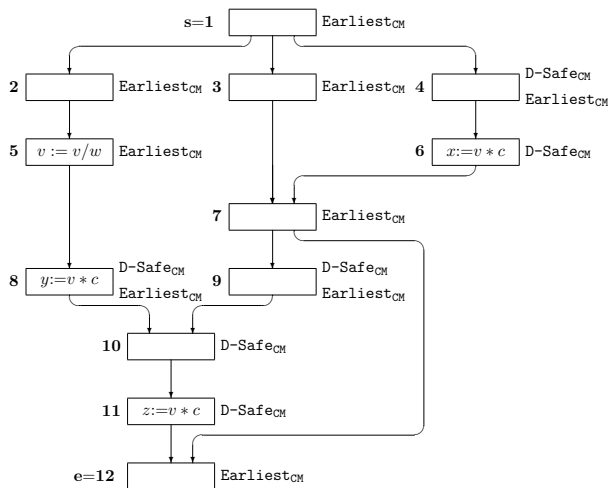
Running Example: Summary

...of the predicate values of the properties required for LSR:

Predicate	Node Number																					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
<i>Safe</i> _{CM}	1	1	1	0	1	0	1	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0
<i>Earliest</i> _{CM}	1	0	0	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0
<i>Insert</i> _{CM}	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
<i>Safe</i> _{SR}	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	0
<i>Earliest</i> _{SR}	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
<i>Insert</i> _{SR}	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
<i>Critical</i>	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0
<i>Subst-Crit</i>	0	0	0	1	0	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0
<i>Insert</i> _{FstRef}	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
<i>Delay</i>	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
<i>Latest</i>	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>Isolated</i>	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1
<i>Update</i> _{SndRef}	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	0	0
<i>Insert</i> _{SndRef}	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>Accumulating</i>	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1	0
<i>Insert</i> _{LSR}	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>InsUpd</i> _{LSR}	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>Delete</i> _{LSR}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Illustrating Down-Safety and Earliestness

...using a new independent example:



D.8

References, Further Reading

Contents

Part I

Chap. 1

Chap. 2

Part II

Chap. 3

Chap. 4

Chap. 5

Chap. 6

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Part III

Chap. 11




Chap. 12

Chap. 13

Part IV

1854/18



Further Reading for Appendix D (1)

-  Frances E. Allen, John Cocke, Ken Kennedy. *Reduction of Operator Strength*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 3, 79-101.
-  Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 10.4.2, Strength Reduction)
-  Dhananjay M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. International Journal of Computer Mathematics 27:1-14&31-32, 1989.

Further Reading for Appendix D (2)

-  Dhananjay M. Dhamdhere, J. R. Isaac. *A Composite Algorithm for Strength Reduction and Code Movement Optimization*. International Journal of Computer and Information Sciences 9(3):243-273, 1980.
-  S. M. Joshi, Dhananjay M. Dhamdhere. *A Composite Hoisting-strength Reduction Transformation for Global Program Optimization – Part I and Part II*. International Journal of Computer Mathematics 11:21-41&111-126, 1982.
-  Robert Kennedy, Fred C. Chow, Peter Dahl, Shing-Ming Liu, Raymond Lo, Mark Streich. *Strength Reduction via SSAPRE*. In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 144-158, 1998.

Further Reading for Appendix D (3)

-  Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. *Journal of Programming Languages* 1(1):71-91, 1993.
-  Bernhard Steffen, Jens Knoop, Oliver Rüthing. *Efficient Code Motion and an Adaption to Strength Reduction*. In *Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, Springer-V., LNCS 494, 394-415, 1991.