Optimierende Compiler

LVA 185.A04, VU 2.0, ECTS 3.0
WS 2013/2014

(Stand: 05.01.2014)

Jens Knoop

Technische Universität Wien
Institut für Computersprachen

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Table of Contents

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Table of Contents (1)

Part I: Introduction

- Chap. 1: Motivation
- Chap. 2: Data Flow Analysis in a Nutshell
  - 2.1 Program Analyis
  - 2.2 Forward Analyses
  - 2.3 Backward Analyses
- Chap. 3: Taxonomy of DFA-Analyses

Table of Contents (2)

Part II: Intraprocedural Data Flow Analysis

- Chap. 4: Flow Graphs
- Chap. 5: The Intraprocedural DFA Framework
  - 5.1 The *MOP* Approach
  - 5.2 The *MaxFP* Approach
  - 5.3 Coincidence and Safety Theorem
  - 5.4 Examples: Available Expressions, Simple Constants
    - 5.4.1 Available Expressions
    - 5.4.2 Simple Constants
- Chap. 6: Partial Redundancy Elimination
  - 6.1 Motivation
  - 6.2 The PRE Algorithm of Morel&Renvoise
- Chap. 7: Busy Code Motion
  - 7.1 Preliminaries
  - 7.2 The *BCM*-Transformation

Table of Contents (3)

- Chap. 8: Lazy Code Motion
  - 8.1 Preliminaries
  - 8.2 The *ALCM*-Transformation
  - 8.3 The *LCM*-Transformation
  - 8.4 An Extended Example
- Chap. 9: Implementing Busy and Lazy Code Motion
  - 9.1 Implementing *BCM* and *LCM* on SI-Graphs
    - 9.1.1 Preliminaries
    - 9.1.2 Implementing BCM_ι
    - 9.1.3 Implementing LCM_ι
  - 9.2 Implementing *BCM* and *LCM* on BB-Graphs
    - 9.2.1 Preliminaries
    - 9.2.2 Implementing BCM_β
    - 9.2.3 Implementing LCM_β
  - 9.3 An Extended Example

Table of Contents (4)

- Chap. 10: Sparse Code Motion
- Chap. 11: Lazy Strength Reduction
- Chap. 12: More on Code Motion

12.1 Motivation
12.2 Code Motion vs. Code Placement
12.3 Interactions of Elementary Transformations
12.4 Paradigm Impacts
12.5 Further Code Motion Transformations

Table of Contents (5)

Part III: Interprocedural Data Flow Analysis

- Chap. 13: The Context Information Approach
- Chap. 14: The Functional Approach

14.1 The Base Setting

14.1.1 Local Abstract Semantics
14.1.2 The *IMOP* Approach
14.1.3 The *IMaxFP* Approach
14.1.4 Main Results
14.1.5 Algorithms
14.1.6 Applications

Table of Contents (6)

- Chap. 14 (Cont'd)
  - 14.2 The General Setting
    - 14.2.1 Local Abstract Semantics
    - 14.2.2 The $IMOP_{Stk}$ Approach
    - 14.2.3 The $IMaxFP_{Stk}$ Approach
    - 14.2.4 Main Results
    - 14.2.5 Algorithms
  - 14.3 Further Extensions
  - 14.4 Applications
  - 14.5 Interprocedural DFA: Framework and Toolkit

Part IV: Extensions, Other Settings

- Chap. 15: Alias Analysis
  - 15.1 Sources of Aliasing
  - 15.2 Relevance of Aliasing for Program Optimization
  - 15.3 Shape Analysis

Table of Contents (7)

- Chap. 16: Optimizations for Object-Oriented Languages
  - 16.1 Object Layout and Method Invocation
    - 16.1.1 Single Inheritance
    - 16.1.2 Multiple Inheritance
  - 16.2 Devirtualization of Method Invocations
    - 16.2.1 Class Hierarchy Analysis
    - 16.2.2 Rapid Type Analysis
    - 16.2.3 Inlining
  - 16.3 Escape Analysis
    - 16.3.1 Connection Graphs
    - 16.3.2 Intraprocedural Setting
    - 16.3.3 Interprocedural Setting
- Chap. 17: Program Slicing

Table of Contents (8)

Part V: Conclusions and Prospectives

- Chap. 18: Summary and Outlook
- Bibliography
- Appendix
  - A Mathematical Foundations
    - A.1 Sets and Relations
    - A.2 Partially Ordered Sets
    - A.3 Lattices
    - A.4 Complete Partially Ordered Sets
    - A.5 Fixed Point Theorems
  - B Intricacies of Basic Block Graphs
    - B.1 Motivation
    - B.2 Availability of Expressions
    - B.3 Constant Propagation and Folding
    - B.4 Faint Variables
    - B.5 Conclusion

Part I

Introduction

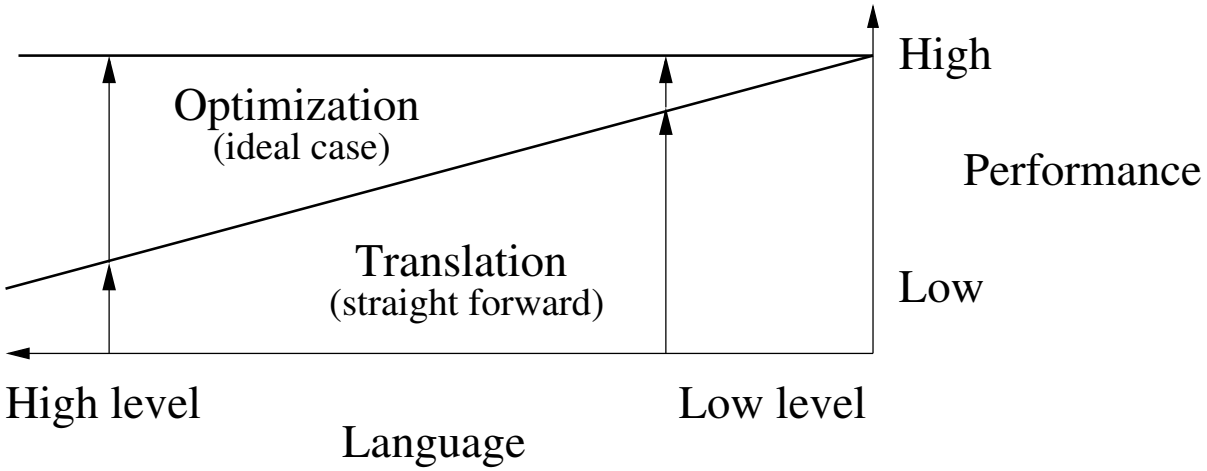
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter 1

Motivation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

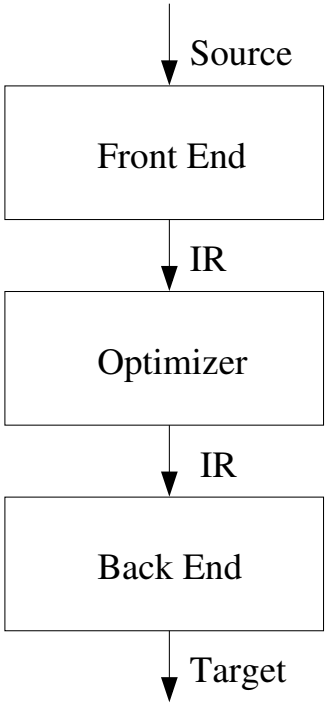
Languages and Their Perceived Performance



- Common perception is that high level languages/abstrac- tion gives low level of performance.
- Translation (straight forward) preserves semantics but does not exploit specific opportunities of lower level language with respect to performance.
- Optimization improves performance (misnomer: usually we do not achieve an "optimal" solution - but it is the ideal case)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Generic Structure of an Optimizing Compiler

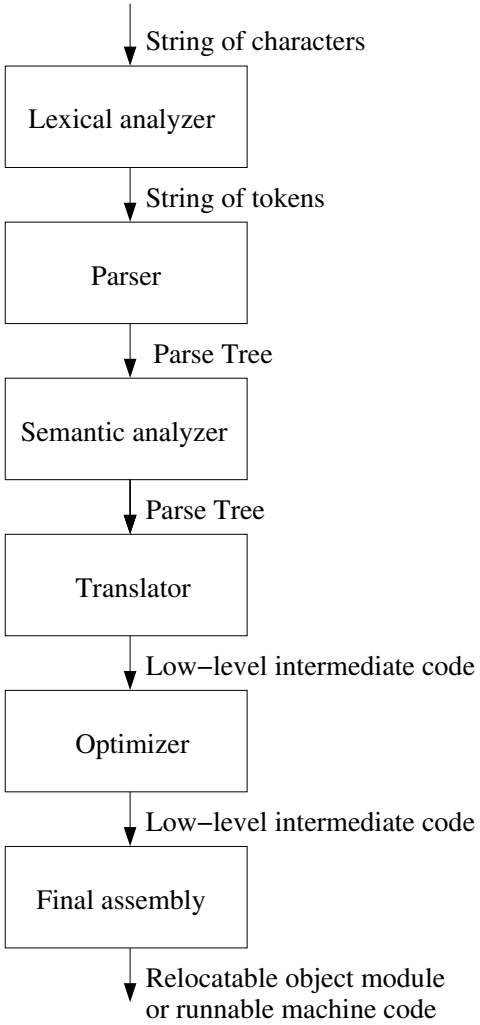


Goal of code optimization

- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

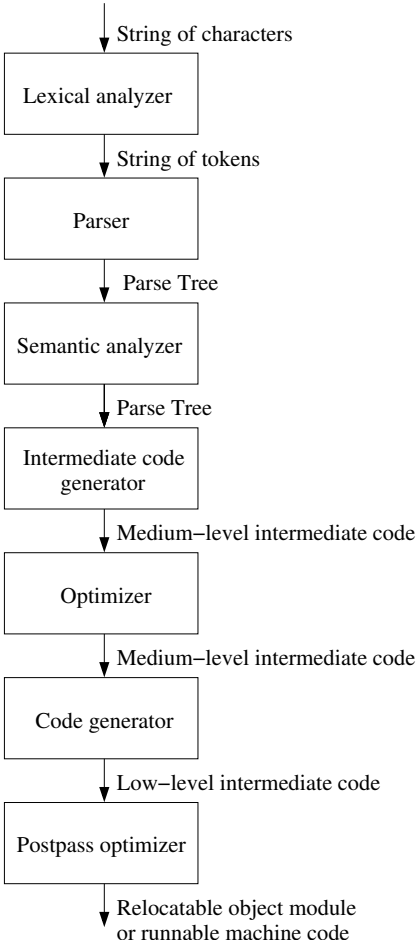
Model of a Low Level Optimizer



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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

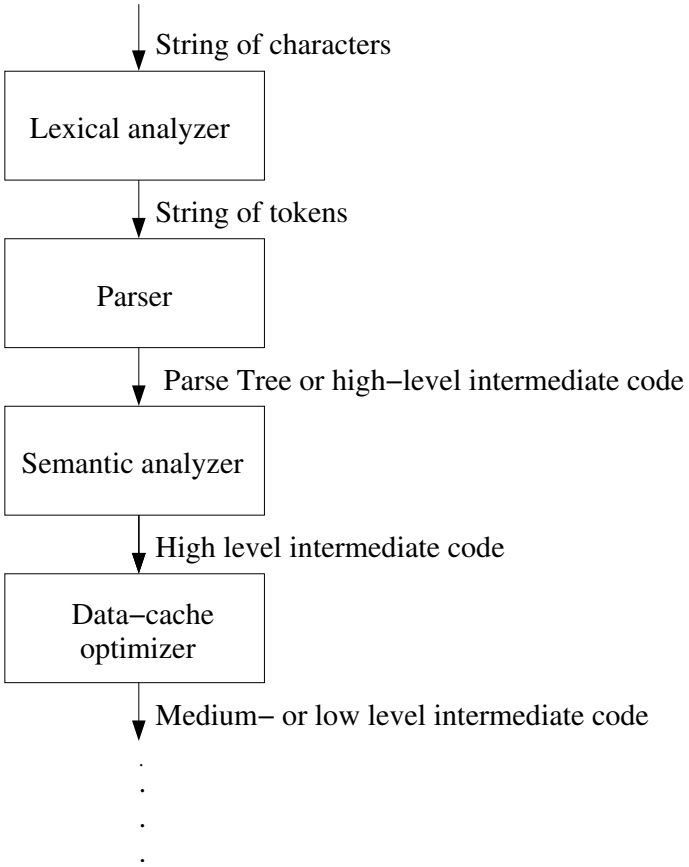
Model of a Mixed Level Optimizer



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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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.

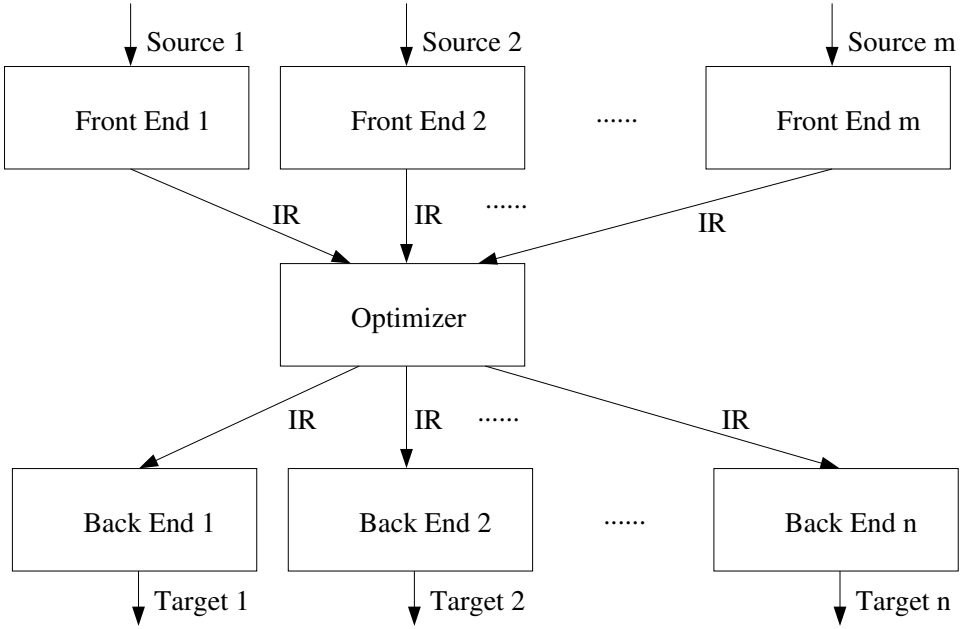
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Practice: m-2-n Compilers and Optimizers



Idea: Decoupling of Compiler Front Ends from Back Ends

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

IR-Decoupling of Compiler Front/Back Ends

...is an application of the well-known UNCOL concept:

```
---------     ---------         --------
|  PL 1 |     |  PL 2 |   ...   |  PL n |
---------      --------         --------
    |             |                |
    ---------------------------------
                  |
                 \/
               ----------          Universal
               |  UNCOL  |      Computer Oriented
               ----------          Language
                  |
    ---------------------------------
    |             |                |
   \/            \/               \/
 ---------     ---------        ---------
 |  ML 1  |    |  ML 2 |   ...  |  ML m |
 ---------      --------        ---------
```

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Different Kinds of Optimizations

...for different purposes, e.g.:

- Speeding up execution of compiled code
- 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
- Energy consumption
- Response to real-time events
- etc.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Scope of Optimization (1)

- Local
  - 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
- Intra-procedural (global)
  - 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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Scope of Optimization (2)

- 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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Machine Independent Optimizations (1)

- Dead code elimination
  - eliminate useless or unreachable code
  - algebraic identities
- Code motion
  - move operation to place where it executes less frequently
  - loop invariant code motion, hoisting, constant propagation
- Specialize
  - to specific context in which an operation will execute
  - operator strength reduction, constant propagation, peephole optimization

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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

Concrete example: 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 EC1-6

Acknowledgement: Joint work of Markus Schordan&Rene Heinzl.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Programming Styles - 1&2

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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Programming Style - 3

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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Programming Style - 4

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: Templatized utility for binding values to function objects

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Programming Styles - 5&6

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Evaluation (EC1-6 without optimization)

- Compiler: GNU g++ 4.0
- Evaluation Cases 1-6
- Time measured in milliseconds, container size: 1,000

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

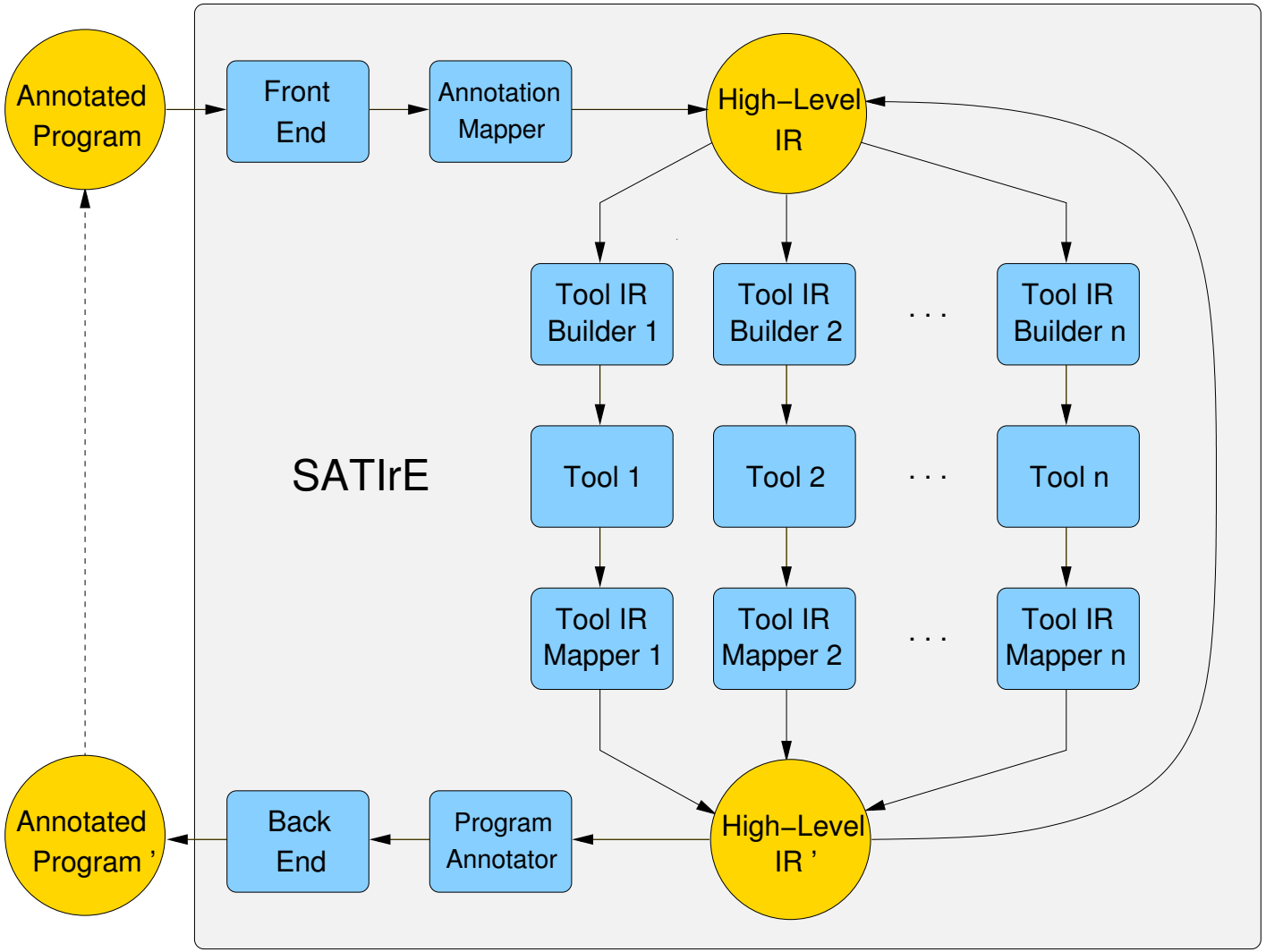
Evaluation: Optimization Levels O1-3

- Compiler: GNU g++ 4.0
- The actual run-time with different optimization levels -O1, -O2, -O3 for each programming style (EC1-6)
- An almost identical run-time is achieved at level -O3.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

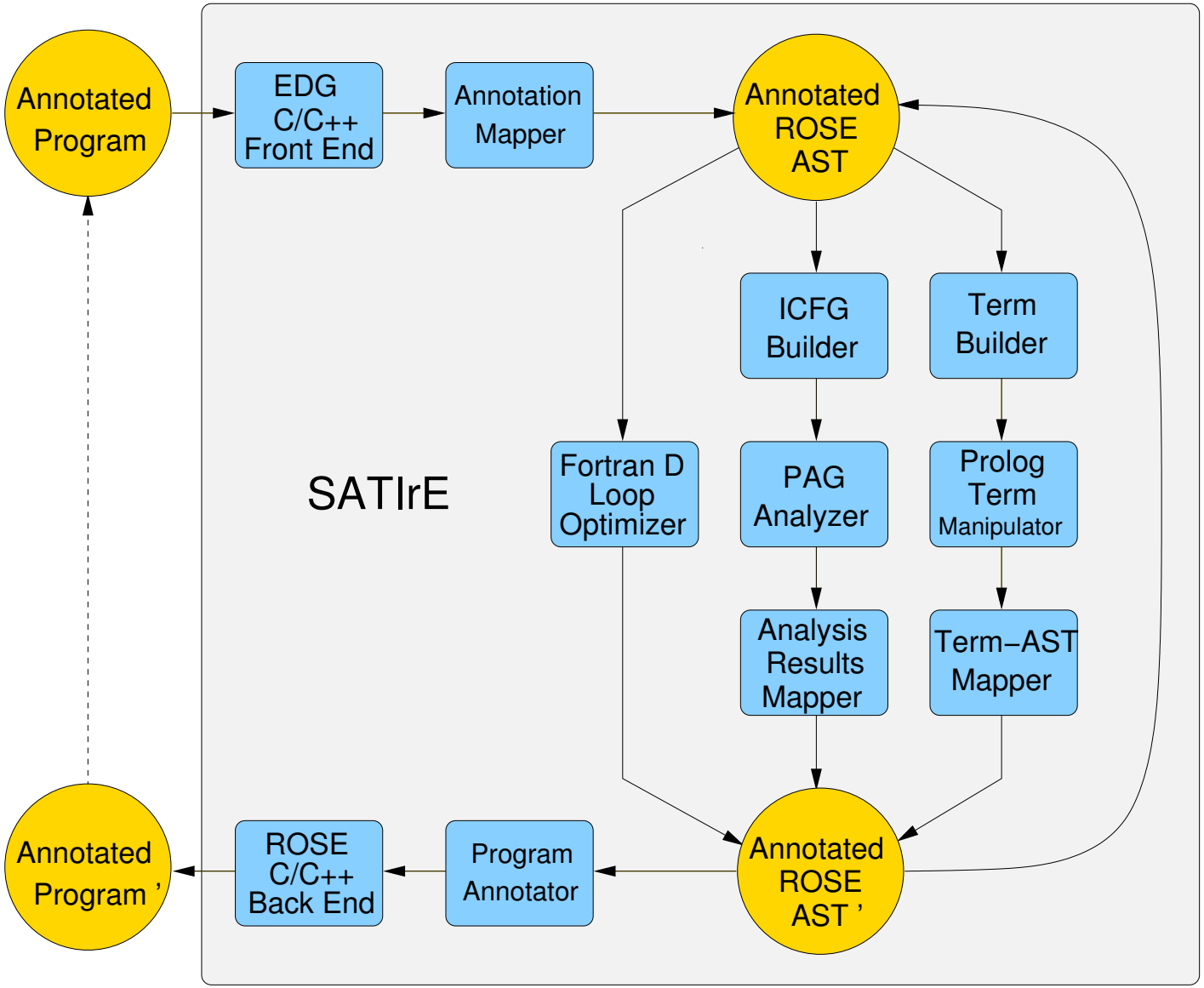
Static Analysis and Tool Integration Engine

SATIrE:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

SATIrE: Concrete Architecture (Oct'07)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

SATIrE Components (1)

- 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

SATIrE Components (2)

- Integration 1 (Loop Optimizer)
  - Loop Optimizer: ported from the Fortran D compiler and integrated in LLNL-ROSE
- Integration 2 (PAG)
  - 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

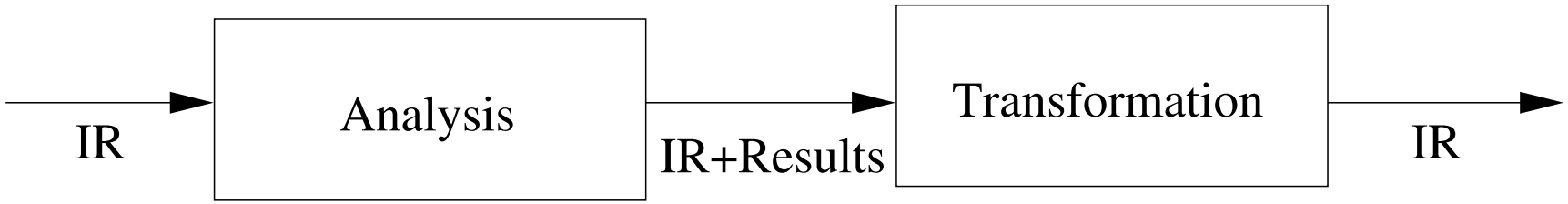
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

SATIrE Components (3)

- Integration 3 (Termite)
  - 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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimization – Schematic View



- Analysis
  - determine properties of program
  - safe, pessimistic assumptions
- Transformation
  - based on analysis results

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The Essence of Program Analysis

...program analysis 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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Typical Optimization Aspects

- Avoid redundant computations
  - reuse available results
  - move loop invariant computations outside loops
- Avoid superfluous computations
  - results known not to be needed
  - results known already at compile time

...to be demonstrated in some examples next.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Example: Lowering, IR, Address Computation

```
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];
  }
}
```

```
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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimization: Common Subexpression Elim.

- 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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Analysis: Loop Invariant 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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimization: Loop Invariant Code Motion

- Analysis: loop invariant 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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Analysis: Induction Variable 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 i whose only definitions within a loop are of the form $i = i + c$ or $i = i - c$ and c is a loop invariant.

Derived Induction Variables

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimization: Strength Reduction (2)

- Analysis: induction variable detection
- Transformation: move multiplication 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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Analysis: Copy Analysis

...determines for each program point, which 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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimization: Copy Propagation

- Analysis: Copy Analysis and def-use chains (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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Live Variables, Dead Variables

- 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.
- If a variable is not live, it is dead.

A live (dead) variable analysis

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Analysis: Dead Variable Analysis

```
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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimization: Dead Code Elimination

- Analysis: dead variable 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;
}
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimizations at a Glance

Analyses	Transformations
Available expr. analysis	Common subexpr. elim.
Loop invariant detection	Invariant code motion
Induction variable detection	Strength reduction
Copy analysis	Copy propagation
Live variables analysis	Dead code elimination

Further optimizations, i.e., analyses and transformations?

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Questions about Heap Contents (1)

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: 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: Does a procedure or a program leave behind unreachable heap cells when it returns?
  - Yes (in some state). Issue a warning.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Questions about Heap Contents (2)

- Aliasing: 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: 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Questions about Heap Contents (3)

- Reachability: 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: 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: 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimizations f. Object-Oriented Languages (1)

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
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Optimizations f. Object-Oriented Languages (2)

Related Optimizations

- Dispatch Table Compression
- Devirtualization
- Inlining
- Escape Analysis for allocating objects on the run-time stack (instead of the heap)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Further Reading for Chapter 1 (1)

- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 1, Compiler Challenges for High-Performance Architectures)
- Melvin E. Conway. *Proposal for an UNCOL*. Communications of the ACM 1(3):5, 1958.
- 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Further Reading for Chapter 1 (2)

- Donald E. Knuth. *An Empirical Study of Fortran Programs*. Software – Practice and Experience 1:105-13, 1971.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 1, Introduction to Advanced Topics)
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Chapter 1, Introduction)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter 2

Data Flow Analysis in a Nutshell

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Chapter 2.1

Program Analysis

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Typical Questions

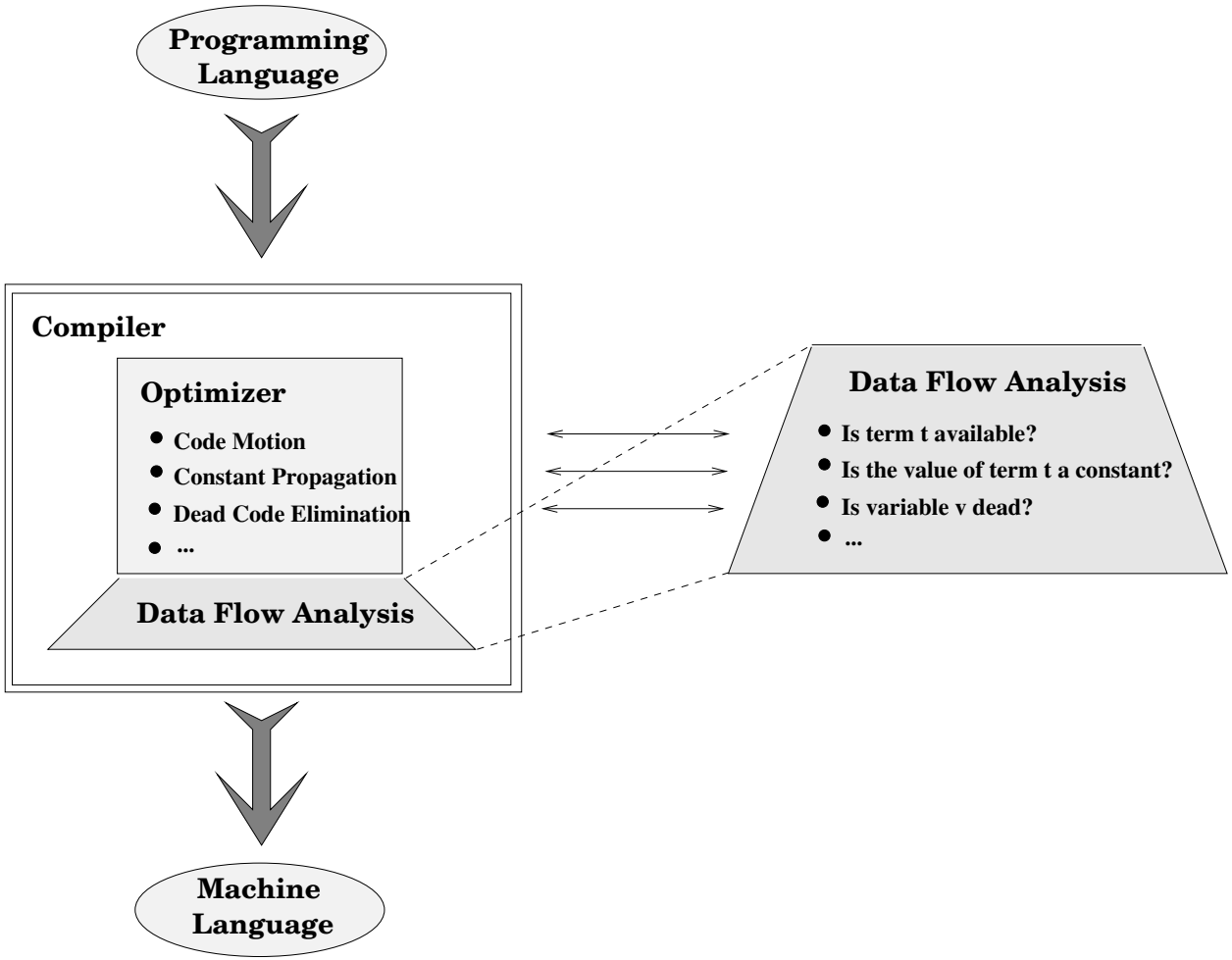
...of program analysis, especially data flow analysis:

- What is the value of a variable at a program point?
  $\rightsquigarrow$ Constant progagation and folding
- Is the value of an expression available at a program point?
  $\rightsquigarrow$ (Partial) redundancy elimination
- Is a variable dead at a program point?
  $\rightsquigarrow$ Elimination of (partially) dead code

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Common (and also our) Application Scenario

...(program) analysis for (program) optimization:



Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Essential Issues

comprise...

fundamental ones

- What does optimality mean?
  ...in analysis and optimization?

as (apparently) minor ones:

- What is an appropriate and suitable program representation?

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Outlook

In more detail we will distinguish:

- intraprocedural
- interprocedural
- parallel
- ...

data flow analysis (DFA).

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Outlook (cont'd)

Ingredients of (intraprocedural) data flow analysis:

- (Local) abstract semantics
  1. A data flow analysis lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$
  2. A data flow analysis functional $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$
  3. A Start information (start assertion) $c_s \in \mathcal{C}$
- Globalization strategies
  1. "Meet over all Paths" Approach (*MOP*)
  2. Maximum Fixed Point Approach (*MaxFP*)
- Generic Fixed Point Algorithm

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Theory of Intraprocedural DFA

Main Results:

- Safety (Soundness) Theorem
- Coincidence (Completeness Theorem

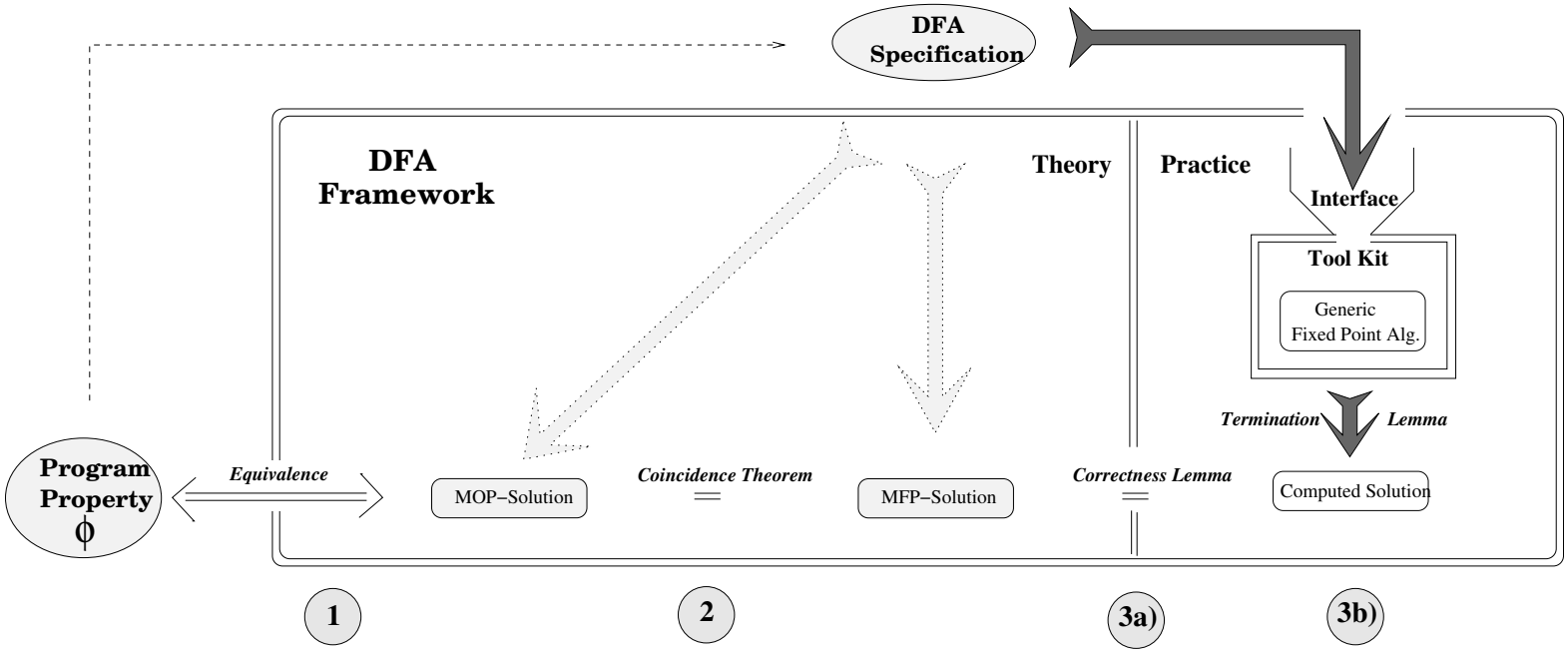
Plus:

- Effectivity (Termination) Theorem

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Practice of Intraprocedural DFA

The Intraprocedural DFA Framework / DFA Toolkit View:

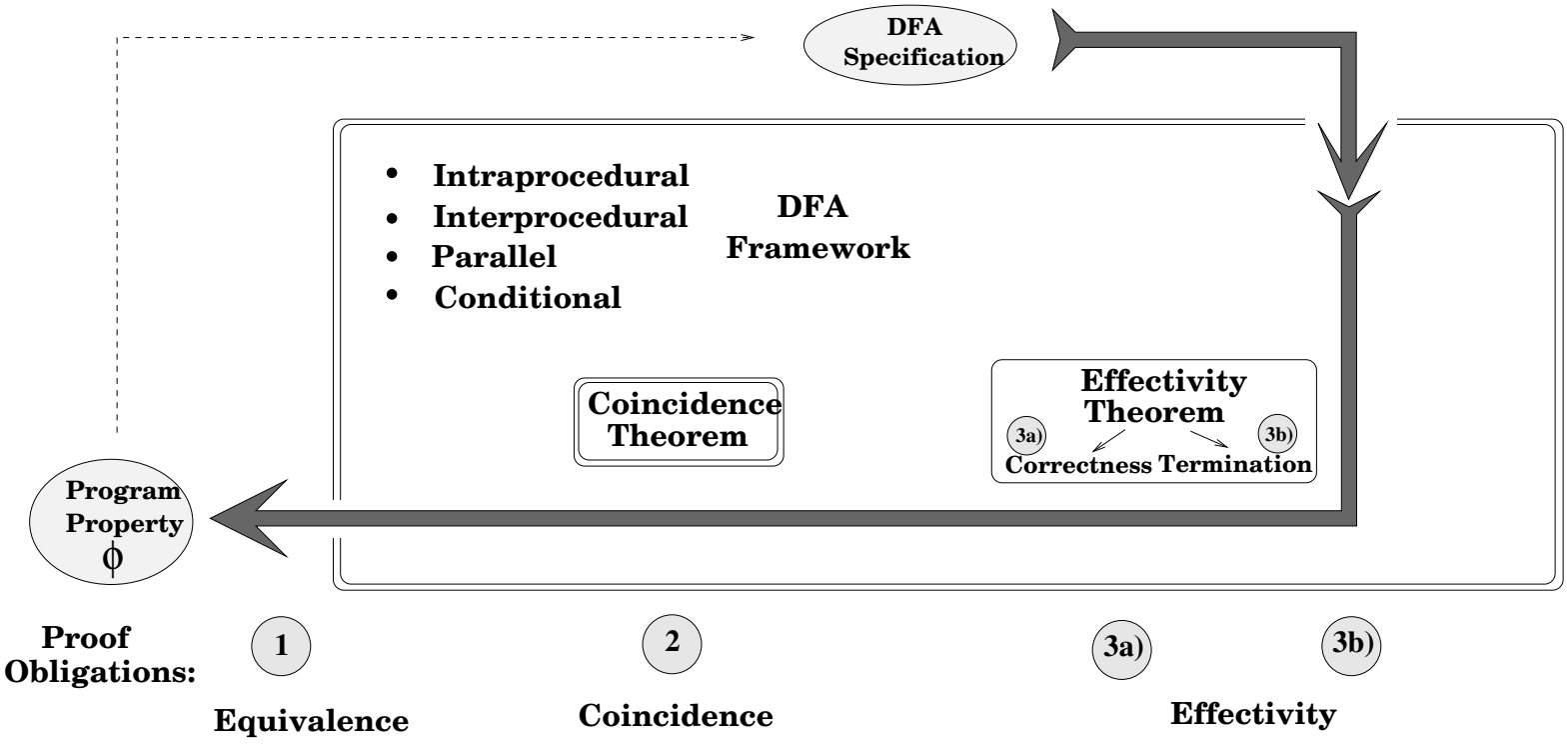


Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Practice of DFA

The constraint "intraprocedural" can be dropped.

The DFA Framework / DFA-Toolkit View holds generally:



Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Ultimate Goal

Optimal Program Optimization

...a white "Schimmel" (two twins) in computer science?

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

There is no free Lunch!

In the diction of optimizing compilation:

...w/out analysis no optimization!

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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 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)

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Chapter 2.2

Forward Analyses

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

WHILE Language

Syntactic categories

a	$\in$ AExp	arithmetic expressions
b	$\in$ BExp	boolean expressions
S	$\in$ Stmt	statements

x, y	$\in$ Var	variables
n	$\in$ Num	numerals
ℓ	$\in$ Lab	labels

op_a	$\in \mathrm{Op}_a$	arithmetic operators
op_b	$\in \mathrm{Op}_b$	boolean operators
op_r	$\in \mathrm{Op}_r$	relational operators

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Abstract Syntax

$$
\begin{array}{lcl}
a & ::= & x \mid n \mid a_1 \; op_a \; a_2 \\
b & ::= & \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \; op_b \; b_2 \mid a_1 \; op_r \; a_2 \\
S & ::= & [\text{x:=a}]^\ell \mid [\text{skip}]^\ell \\
 & & \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \\
 & & \mid \text{while}[b]^\ell \text{ do } S \text{ od} \\
 & & \mid S_1; S_2
\end{array}
$$

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 paranthesis to disambiguate syntax.

We will often refer to labelled fragments as elementary blocks.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Computing the Information (1)

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Computing the Information (2)

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

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Program of Interest (1)

We shall use the following notation:

- $S_\star$ to represent the program being analyzed (the "top level" statement)
- $\mathrm{Lab}_\star$ to represent the labels ($\mathrm{labels}(S_\star)$) appearing in $S_\star$
- $\mathrm{Var}_\star$ to represent the variables ($\mathrm{FV}(S_\star)$) appearing in $S_\star$
- $\mathrm{Blocks}_\star$ to represent the elementary blocks ($\mathrm{blocks}(S_\star)$) occuring in $S_\star$
- $\mathrm{AExp}_\star$ to represent the set of *non-trivial* arithmetic subexpressions in $S_\star$; an expression is trivial if it is a single variable or constant
- $\mathrm{AExp}(a)$, $\mathrm{AExp}(b)$ to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Program of Interest (2)

Free Variables FV(a)

The free variables of an arithmetic expression, $a \in \mathsf{AExp}$, are defined to be variables occuring in it.

Compositional definition of subset FV(a) of Var

$$
\begin{aligned}
\mathsf{FV}(n) &= \emptyset \\
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(a_1 + a_2) &= \mathsf{FV}(a_1) \cup \mathsf{FV}(a_2) \\
\mathsf{FV}(a_1 \star a_2) &= \mathsf{FV}(a_1) \cup \mathsf{FV}(a_2) \\
\mathsf{FV}(a_1 - a_2) &= \mathsf{FV}(a_1) \cup \mathsf{FV}(a_2)
\end{aligned}
$$

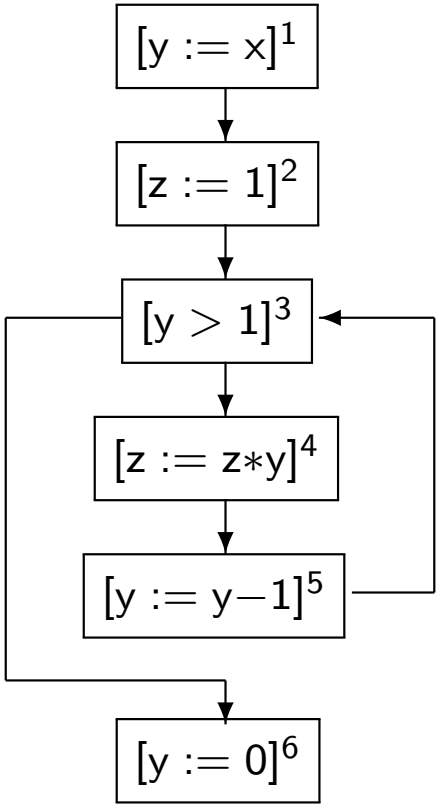
Similarly for boolean expressions, $b \in \mathsf{BExp}$, and statements, $S \in \mathsf{Stmt}$, such that $\mathsf{Var}_\star = \mathsf{FV}(S_\star)$.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

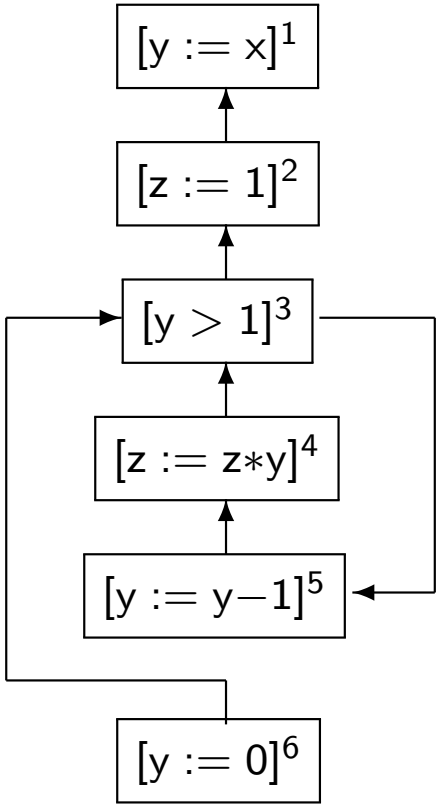
Flow Graphs – Example (1)

Example:

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



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



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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Flow Graphs – Example (2)

Example:

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

$$
\begin{aligned}
\text{labels}(S_\star) &= \{1, 2, 3, 4, 5, 6\} \\
\text{init}(S_\star) &= 1 \\
\text{final}(S_\star) &= \{6\} \\
\text{flow}(S_\star) &= \{(1,2), (2,3), (3,4), (4,5), (5,3), (3,6)\} \\
\text{flow}^R(S_\star) &= \{(6,3), (3,5), (5,4), (4,3), (3,2), (2,1)\} \\
\text{blocks}(S_\star) &= \{[y := x]^1, [z := 1]^2, [y > 1]^3, \\
&\quad [z := z * y]^4, [y := y - 1]^5, [y := 0]^6\}
\end{aligned}
$$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Simplifying Assumptions

The program of interest $S_\star$ is often assumed to satisfy:

- $S_\star$ has isolated entries if there are no edges leading into $\mathsf{init}(S_\star)$:
$$\forall \ell : (\ell, \mathsf{init}(S_\star)) \notin \mathsf{flow}(S_\star)$$
- $S_\star$ has isolated exits if there are no edges leading out of labels in $\mathsf{final}(S_\star)$:
$$\forall \ell \in \mathsf{final}(S_\star), \forall \ell' : (\ell, \ell') \notin \mathsf{flow}(S_\star)$$
- $S_\star$ is label consistent if
$$\forall B_1^{\ell_1}, B_2^{\ell_2} \in \mathsf{blocks}(S_\star) : \ell_1 = \ell_2 \rightarrow B_1 = B_2$$

This holds if $S_\star$ is uniquely labelled.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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.

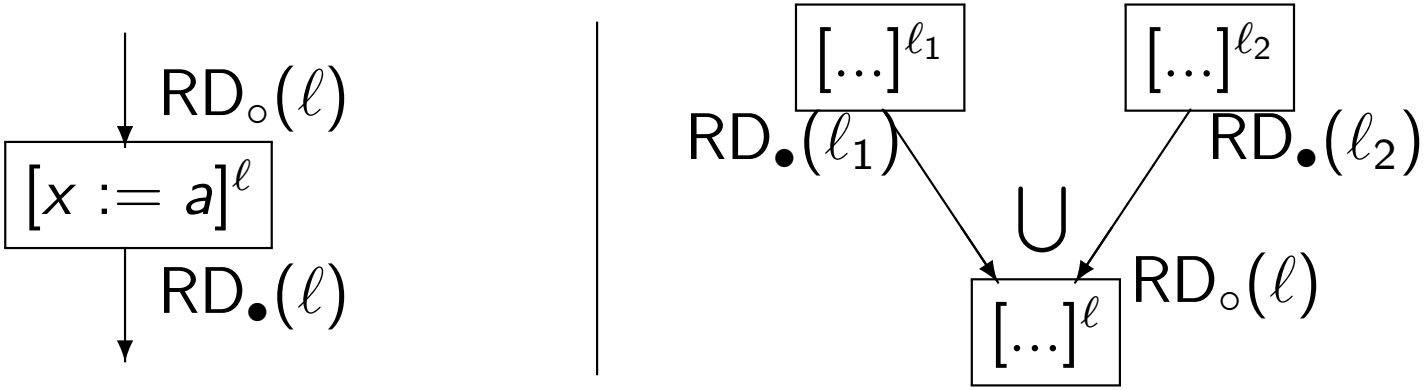
Example:

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

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Basic Idea



Analysis information: $\mathsf{RD}_\circ(\ell), \mathsf{RD}_\bullet(\ell) : \mathsf{Lab}_\star \to \mathcal{P}\mathsf{Var}_\star \times \mathsf{Lab}_\star^?$

- $\mathsf{RD}_\circ(\ell)$: the definitions that reach entry of block ℓ.
- $\mathsf{RD}_\bullet(\ell)$: the definitions that reach exit of block ℓ.

Analysis properties:

- Direction: forward
- May analysis with combination operator $\bigcup$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Analysis of Elementary Blocks

$$
\begin{array}{rcl}
\mathsf{kill}_{\mathsf{RD}}([x := a]^{\ell}) & = & \{(x, ?)\} \ \cup \ \{(x, \ell') \mid B^{\ell'} \text{is assignment to } x\} \\
\mathsf{kill}_{\mathsf{RD}}([\mathsf{skip}]^{\ell}) & = & \emptyset \\
\mathsf{kill}_{\mathsf{RD}}([b]^{\ell}) & = & \emptyset \\
\mathsf{gen}_{\mathsf{RD}}([x := a]^{\ell}) & = & \{(x, \ell)\} \\
\mathsf{gen}_{\mathsf{RD}}([\mathsf{skip}]^{\ell}) & = & \emptyset \\
\mathsf{gen}_{\mathsf{RD}}([b]^{\ell}) & = & \emptyset
\end{array}
$$

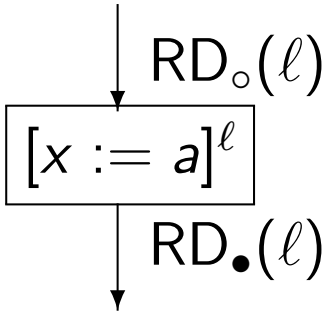
Example:

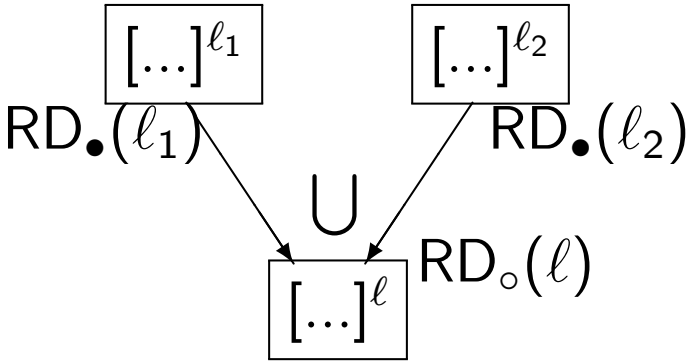
$[\mathrm{x} := \mathrm{y}]^1; [\mathrm{x} := \mathrm{x} + 3]^2;$

- $\mathsf{kill}_{\mathsf{RD}}([\mathrm{x} := \mathrm{y}]^1) = \{(x, ?)\} \cup \{(x, 1), (x, 2)\}$
- $\mathsf{gen}_{\mathsf{RD}}([\mathrm{x} := \mathrm{y}]^1) = \{(x, 1)\}$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Analysis of the Program





$$
\begin{array}{rcll}
\mathsf{RD}_\circ(\ell) & = & \left\{ \begin{array}{ll} \{(x, ?) \mid x \in FV(S_\star)\} & : \quad \text{if } \ell = \mathsf{init}(S_\star) \\ \bigcup\{\mathsf{RD}_\bullet(\ell') \mid (\ell', \ell) \in \mathsf{flow}(S_\star)\} & : \quad \text{otherwise} \end{array} \right. & \\
\mathsf{RD}_\bullet(\ell) & = & (\mathsf{RD}_\circ(\ell) \backslash \mathsf{kill}_{\mathsf{RD}}(B^\ell)) \ \cup \ \mathsf{gen}_{\mathsf{RD}}(B^\ell) & \text{where } B^\ell \in \mathsf{blocks}(S_\star)
\end{array}
$$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Example

Example:

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

Equations: Let

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

$$
\begin{array}{lcl@{\qquad}lcl}
\mathrm{RD}_\circ(1) & = & \{(x,?),(y,?),(z,?)\} & \mathrm{RD}_\bullet(1) & = & \mathrm{RD}_\circ(1) \setminus S_1 \cup \{(y,1)\} \\
\mathrm{RD}_\circ(2) & = & \mathrm{RD}_\bullet(1) & \mathrm{RD}_\bullet(2) & = & \mathrm{RD}_\circ(2) \setminus S_2 \cup \{(z,2)\} \\
\mathrm{RD}_\circ(3) & = & \mathrm{RD}_\bullet(2) \cup \mathrm{RD}_\bullet(5) & \mathrm{RD}_\bullet(3) & = & \mathrm{RD}_\circ(3) \\
\mathrm{RD}_\circ(4) & = & \mathrm{RD}_\bullet(3) & \mathrm{RD}_\bullet(4) & = & \mathrm{RD}_\circ(4) \setminus S_2 \cup \{(z,4)\} \\
\mathrm{RD}_\circ(5) & = & \mathrm{RD}_\bullet(4) & \mathrm{RD}_\bullet(5) & = & \mathrm{RD}_\circ(5) \setminus S_1 \cup \{(y,5)\} \\
\mathrm{RD}_\circ(6) & = & \mathrm{RD}_\bullet(3) & \mathrm{RD}_\bullet(6) & = & \mathrm{RD}_\circ(6) \setminus S_1 \cup \{(y,6)\}
\end{array}
$$

ℓ	$\mathrm{RD}_\circ(\ell)$	$\mathrm{RD}_\bullet(\ell)$
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)}

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Solving RD Equations

Input

- a set of reaching definitions equations

Output

- the least solution to the equations: $\mathrm{RD}_\circ$

Data structures

- The current analysis result for block entries: $\mathrm{RD}_\circ$
- 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'.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Solving RD Equations – Algorithm

```
W:=nil;
foreach (ℓ,ℓ') ∈ flow(S*) do W := cons((ℓ,ℓ'),W); od;
foreach ℓ ∈ labels(S*) do
   if ℓ ∈ init(S*) then
      RD∘(ℓ) := {(x,?) | x ∈ FV(S*)}
   else
      RD∘(ℓ) := ∅
   fi
od
while W ≠ nil do
   (ℓ,ℓ') := head(W);
   W := tail(W);
   if (RD∘(ℓ)\kill_RD(B^ℓ)) ∪ gen_RD(B^ℓ) ⊈ RD∘(ℓ') then
      RD∘(ℓ') := RD∘(ℓ') ∪ (RD∘(ℓ)\kill_RD(B^ℓ)) ∪ gen_RD(B^ℓ);
      foreach ℓ'' with (ℓ',ℓ'') in flow(S*) do
         W := cons((ℓ',ℓ''),W);
      od
   fi
od
```

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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

  $[x := 0]^1; [x := 5]^2; [y := x]^3; [z := x]^4$

- Definition-Use chains or *du* chains

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

  $[x := 0]^1; [x := 5]^2; [y := x]^3; [z := x]^4$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

UD/DU Chains – Defined via RDs

$$\mathsf{UD},\mathsf{DU} : \mathsf{Var}_\star \times \mathsf{Lab}_\star \to \mathcal{P}(\mathsf{Lab}_\star)$$

are defined by

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

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

and

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Available Expressions Analysis

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

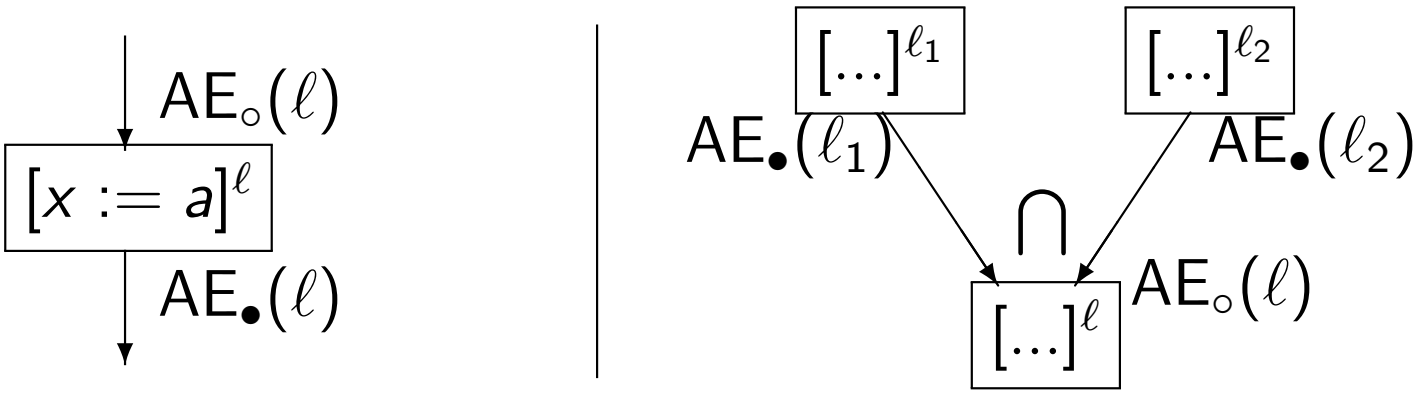
Example:

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

- No expression is available at the start of the program.
- An expression is considered available if no path kills it.
- The expression a+b is available every time execution reaches the test in the loop at 3.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Basic Idea



Analysis information: $AE_\circ(\ell), AE_\bullet(\ell) : Lab_\star \to \mathcal{P}AExp_\star$

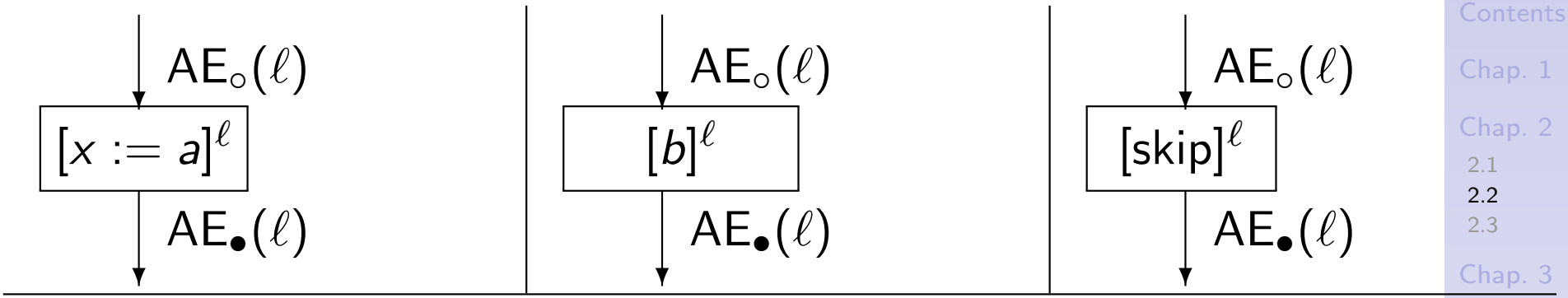
- $AE_\circ(\ell)$: the expressions that have been comp. at entry of block ℓ.
- $AE_\bullet(\ell)$: the expressions that have been comp. at exit of block ℓ.

Analysis properties:

- Direction: forward
- Must analysis with combination operator $\bigcap$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Analysis of Elementary Blocks



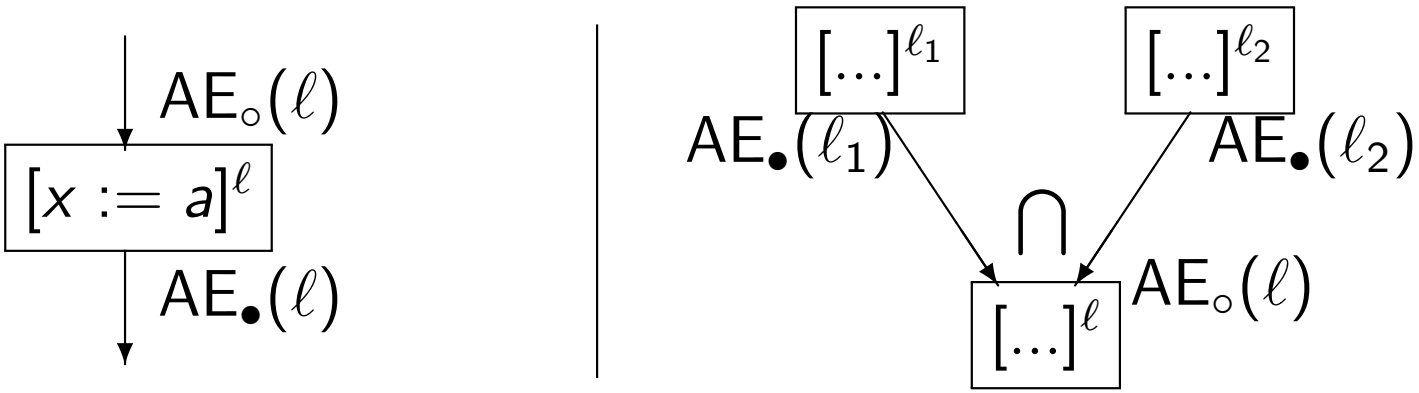
$$
\begin{aligned}
\mathrm{kill}_{\mathrm{AE}}([x := a]^\ell) &= \{a' \in \mathrm{AExp}_\star \mid x \in FV(a')\} \\
\mathrm{kill}_{\mathrm{AE}}([\mathrm{skip}]^\ell) &= \emptyset \\
\mathrm{kill}_{\mathrm{AE}}([b]^\ell) &= \emptyset \\
\mathrm{gen}_{\mathrm{AE}}([x := a]^\ell) &= \{a' \in \mathrm{AExp}(a) \mid x \notin FV(a')\} \\
\mathrm{gen}_{\mathrm{AE}}([\mathrm{skip}]^\ell) &= \emptyset \\
\mathrm{gen}_{\mathrm{AE}}([b]^\ell) &= \mathrm{AExp}(b)
\end{aligned}
$$

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

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Analysis of the Program



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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Example

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:

$$
\begin{array}{lcl@{\qquad}lcl}
AE_\circ(1) & = & \emptyset & AE_\bullet(1) & = & AE_\circ(1) \setminus \{a*x\} \cup \{a+b\} \\
AE_\circ(2) & = & AE_\bullet(1) & AE_\bullet(2) & = & AE_\circ(2) \setminus \emptyset \cup \{a*x\} \\
AE_\circ(3) & = & AE_\bullet(2) \cap AE_\bullet(5) & AE_\bullet(3) & = & AE_\circ(3) \setminus \emptyset \cup \{a+b\} \\
AE_\circ(4) & = & AE_\bullet(3) & AE_\bullet(4) & = & AE_\circ(4) \setminus \{a+b, a*x, a+1\} \cup \emptyset \\
AE_\circ(5) & = & AE_\bullet(4) & AE_\bullet(5) & = & AE_\circ(5) \setminus \{a*x\} \cup \{a+b\}
\end{array}
$$

ℓ	$AE_\circ(\ell)$	$AE_\bullet(\ell)$
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: predefined AE Analysis in PAG/WWW includes boolean expressions

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Solving AE Equations

Input

- a set of available expressions equations

Output

- the largest solution to the equations: $\mathsf{AE}_\circ$

Data structures

- The current analysis result for block entries: $\mathsf{AE}_\circ$
- 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'.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Solving AE Equations – Algorithm

```
W:=nil;
foreach (ℓ,ℓ') ∈ flow(S⋆) do W := cons((ℓ,ℓ'),W); od;
foreach ℓ ∈ labels(S⋆) do
   if ℓ ∈ init(S⋆) then
      AE∘(ℓ) := ∅
   else
      AE∘(ℓ) := AExp⋆
   fi
od
while W ≠ nil do
   (ℓ,ℓ') := head(W);
   W := tail(W);
   if (AE∘(ℓ)\kill_AE(B^ℓ)) ∪ gen_AE(B^ℓ) ⊉ AE∘(ℓ') then
      AE∘(ℓ') := AE∘(ℓ') ∩ (AE∘(ℓ)\kill_AE(B^ℓ)) ∪ gen_AE(B^ℓ);
      foreach ℓ'' with (ℓ',ℓ'') in flow(S⋆) do
         W := cons((ℓ',ℓ''),W);
      od
   fi
od
```

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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 $[\ldots a \ldots]^\ell$ in $S_\star^N$ with $a \in \mathrm{AE}_\circ(\ell)$ do

- determine the set $\{[y_1 := a]^{\ell_1}, \ldots, [y_k := a]^{\ell_k}\}$ of elementary blocks in $S_\star^N$ "defining" a that reaches $[\ldots a \ldots]^\ell$
- 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 $[\ldots a \ldots]^\ell$ with $[\ldots u \ldots]^\ell$

$[x := a]^{\ell'}$ reaches $[\ldots a \ldots]^\ell$ if there is a path in $\mathrm{flow}(S_\star^N)$ 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.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Computing the "reaches" Information

$[x := a]^{\ell'}$ reaches $[\ldots a \ldots]^{\ell}$ 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 $[\ldots a \ldots]^{\ell}$ can be computed as $\text{reaches}_\circ(a, \ell)$ where

$$\text{reaches}_\circ(a, \ell) = \begin{cases} \emptyset & : \text{ if } \ell = \text{init}(S_\star) \\ \bigcup \text{reaches}_\bullet(a, \ell') & : \text{ otherwise} \end{cases}$$

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Example – CSE

Example:

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

ℓ	$\mathsf{AE}_\circ(\ell)$
1	$\emptyset$
2	$\{\mathsf{a}{+}\mathsf{b}\}$
3	$\{\mathsf{a}{+}\mathsf{b}\}$
4	$\{\mathsf{a}{+}\mathsf{b}\}$
5	$\emptyset$

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

Result of CSE optimization wrt reaches(a+b,3):

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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

ℓ	$\mathsf{C}_\circ(\ell)$	$\mathsf{C}_\bullet(\ell)$
1	$\emptyset$	$\{(\mathsf{a},\mathsf{b})\}$
2	$\{(\mathsf{a},\mathsf{b})\}$	$\{(\mathsf{a},\mathsf{b})\}$
3	$\{(\mathsf{a},\mathsf{b})\}$	$\{(\mathsf{y},\mathsf{a}),(\mathsf{a},\mathsf{b})\}$
4	$\{(\mathsf{a},\mathsf{b})\}$	$\emptyset$
5	$\emptyset$	$\{(\mathsf{y},\mathsf{a})\}$
6	$\{(\mathsf{y},\mathsf{a})\}$	$\{(\mathsf{y},\mathsf{a})\}$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Copy Propagation (CP) (1)

...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.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Copy Propagation (CP) (2)

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 CP

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

The Optimization – CP

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

- determine the set $\{[...x...]^{\ell_1}, ..., [...x...]^{\ell_i}\}$,$1 \leq i \leq k$, of elementary blocks in $S_\star$ that uses $[x := y]^{\ell_j}$ – this can be computed from DU(x,ℓ_j)
- for each $[...x...]^{\ell_i}$ in this set determine whether $\{(x', y') \in \mathrm{C}_\circ(\ell_i) \mid x' = x\} = \{(x, y)\}$; 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 $[...x...]^{\ell_i}$ with $[...y...]^{\ell_i}$ for $1 \leq i \leq k$.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Examples – CP

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 CP

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

Example 3

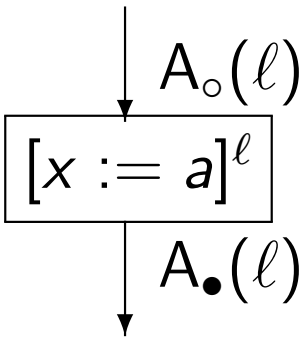
$[a := 10]^1; [b := a]^2; \text{while } [a > 1]^3 \text{ do } [a := a - 1]^4; [b := a]^5; \text{ od } \ [y := y*b]^6;$

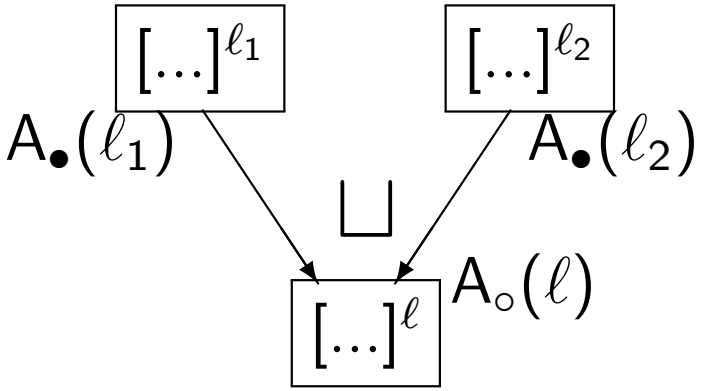
becomes after CP

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Summary: Forward Analyses





$$
\begin{array}{rcll}
A_\circ(\ell) & = & \begin{cases} \iota_A & : \text{ if } \ell = \mathrm{init}(S_\star) \\ \bigsqcup_A \{A_\bullet(\ell') \mid (\ell', \ell) \in \mathrm{flow}(S_\star)\} & : \text{ otherwise} \end{cases} & \\
A_\bullet(\ell) & = & (A_\circ(\ell) \backslash \mathrm{kill}_A(B^\ell)) \ \cup \ \mathrm{gen}_A(B^\ell) & \text{where } B^\ell \in \mathrm{blocks}(S_\star)
\end{array}
$$

where

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Further Reading for Chapter 2.2

Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Chapter 2, Data Flow Analysis)

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Chapter 2.3

Backward Analyses

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Live Variable Analysis

Definition 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.

The Aim of the Live Variables Analysis is to

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

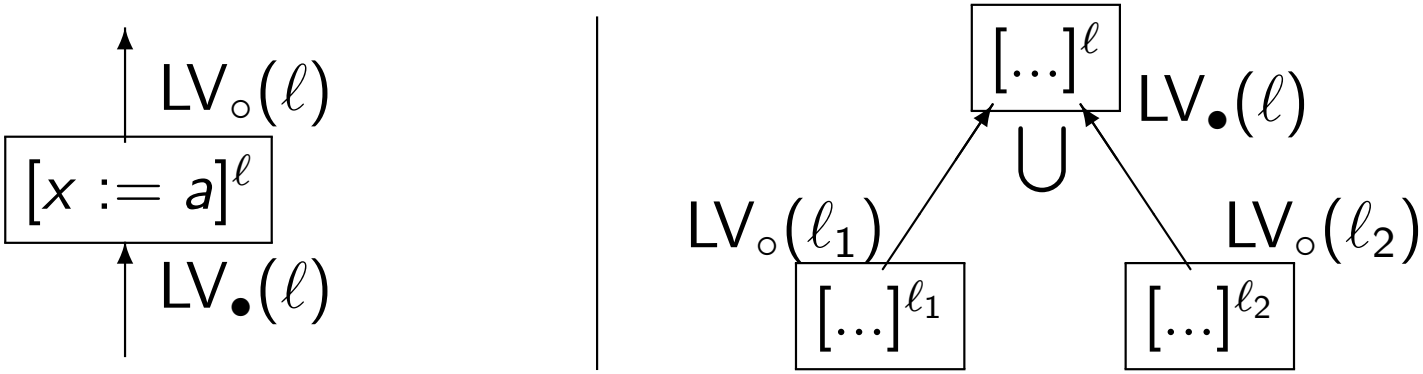
Example

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

- 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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Basic Idea



Analysis information: $\mathsf{LV}_\circ(\ell)$,$\mathsf{LV}_\bullet(\ell)$: $\mathsf{Lab}_\star \to \mathcal{P}\mathsf{Var}_\star$

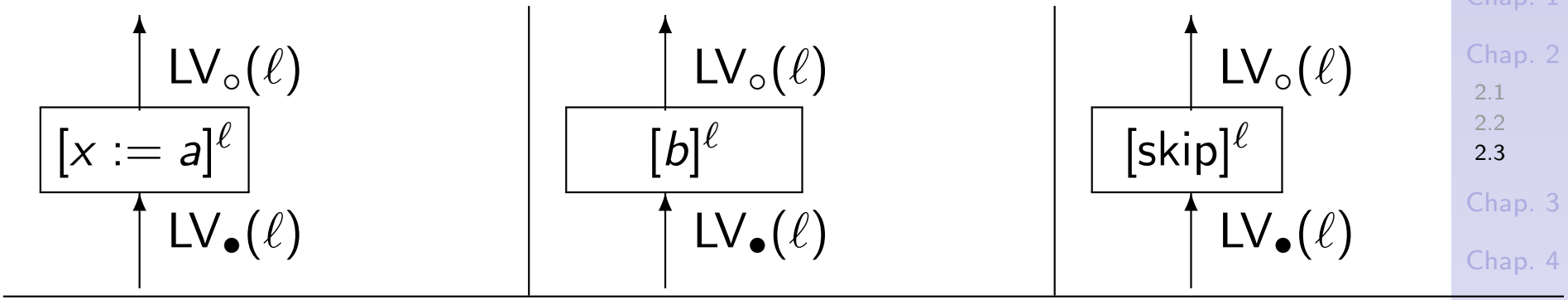
- $\mathsf{LV}_\circ(\ell)$: the variables that are live at entry of block ℓ.
- $\mathsf{LV}_\bullet(\ell)$: the variables that are live at exit of block ℓ.

Analysis properties:

- Direction: backward
- May analysis with combination operator $\bigcup$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

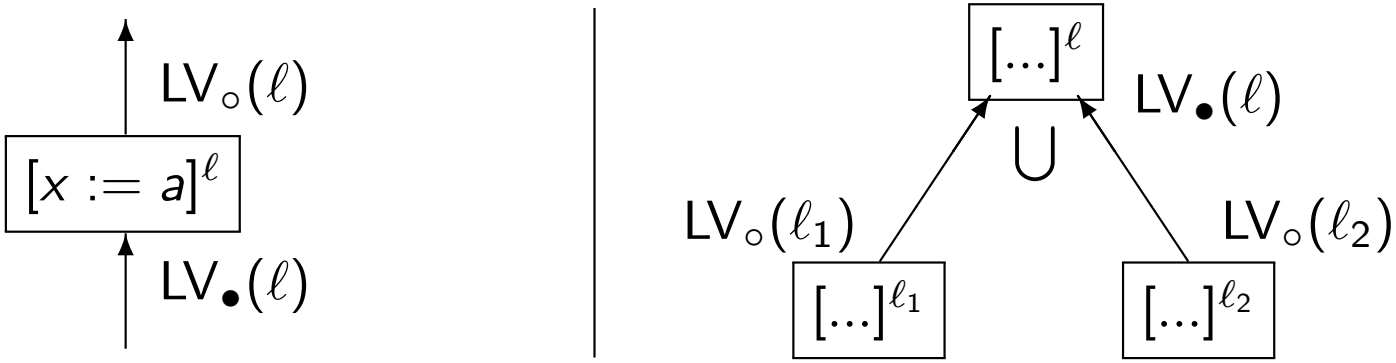
Analysis of Elementary Blocks



$$
\begin{array}{rcl}
\mathsf{kill}_{\mathsf{LV}}([x := a]^\ell) & = & \{x\} \\
\mathsf{kill}_{\mathsf{LV}}([\mathsf{skip}]^\ell) & = & \emptyset \\
\mathsf{kill}_{\mathsf{LV}}([b]^\ell) & = & \emptyset \\
\mathsf{gen}_{\mathsf{LV}}([x := a]^\ell) & = & FV(a) \\
\mathsf{gen}_{\mathsf{LV}}([\mathsf{skip}]^\ell) & = & \emptyset \\
\mathsf{gen}_{\mathsf{LV}}([b]^\ell) & = & FV(b)
\end{array}
$$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Analysis of the Program



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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Example

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

ℓ	$\text{kill}_{LV}(\ell)$	$\text{gen}_{LV}(\ell)$
0	$\{y\}$	$\emptyset$
1	$\{u\}$	$\{a,b\}$
2	$\{y\}$	$\{a,u\}$
3	$\emptyset$	$\{y,u\}$
4	$\{a\}$	$\{a\}$
5	$\{u\}$	$\{a,b\}$
6	$\{x\}$	$\{u\}$
7	$\emptyset$	$\emptyset$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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 $x \notin \mathsf{LV}_\bullet(\ell)$ (i.e. dead) eliminate $[x := a]^\ell$ from the program.

Example:

Before DCE:

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

After DCE:

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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?

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Faint Variables

...generalize the notion of dead variables.

Consider the following program consisting of three statements:

$[\mathsf{x} := 1]^1; [\mathsf{x} := 2]^2; [\mathsf{y} := \mathsf{x}]^3;$

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.

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Very Busy Expressions Analysis

Definition 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 are redefined.

The Aim of the Very Busy Expression Analysis is to

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

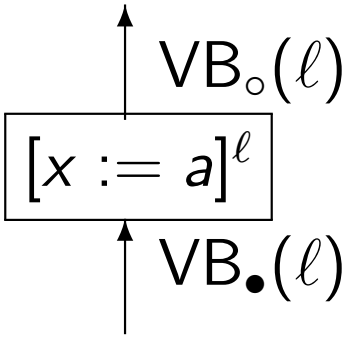
Example

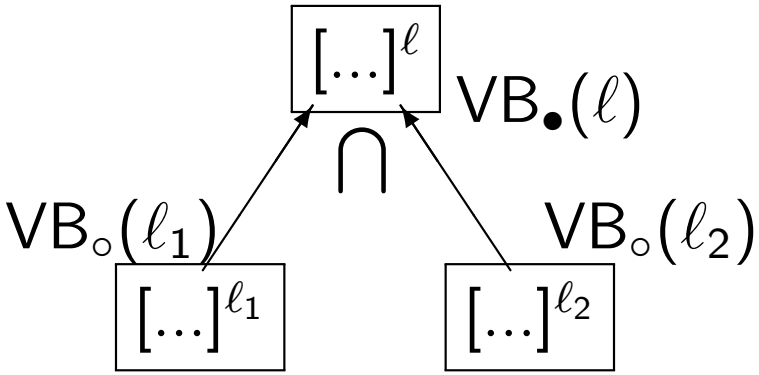
if $[a > b]^1$ then $([x := b-a]^2; [y := a-b]^3)$ else $([y := b-a]^4; [x := a-b]^5)$

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Basic Idea





Analysis information: $\mathsf{VB}_\circ(\ell)$,$\mathsf{VB}_\bullet(\ell)$: $\mathsf{Lab}_\star \to \mathcal{P}\mathsf{AExp}_\star$

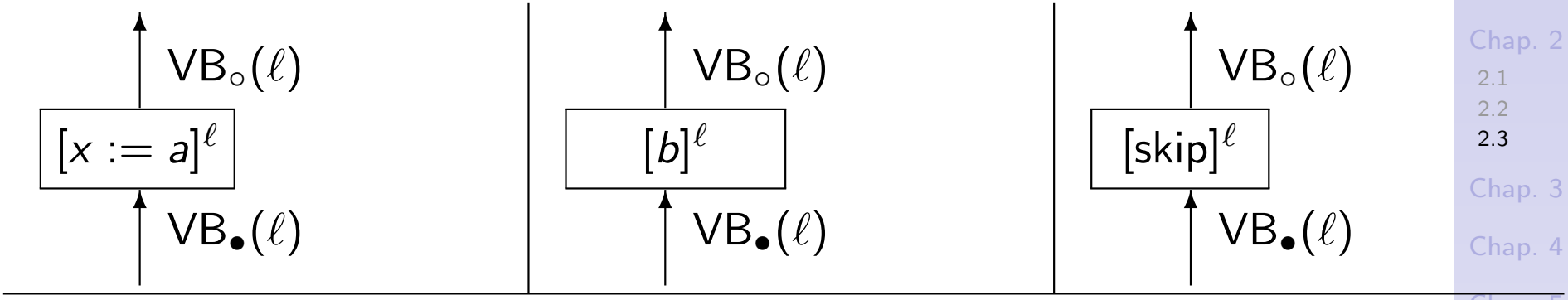
- $\mathsf{VB}_\circ(\ell)$: the expressions that are very busy at entry of block ℓ.
- $\mathsf{VB}_\bullet(\ell)$: the expressions that are very busy at exit of block ℓ.

Analysis properties:

- Direction: backward
- Must analysis with combination operator $\bigcap$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

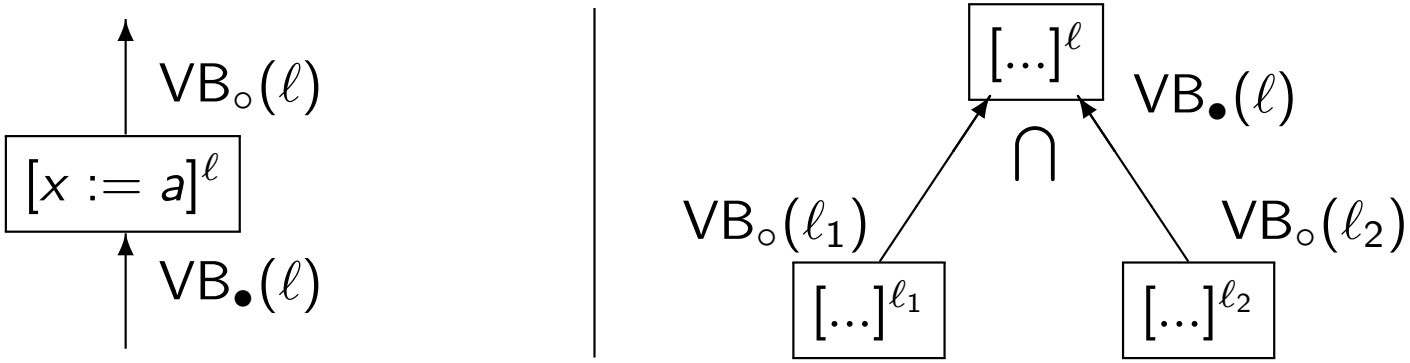
Analysis of Elementary Blocks



$$
\begin{array}{rcl}
\mathsf{kill}_{\mathsf{VB}}([x := a]^\ell) & = & \{a' \in \mathsf{AExp}_\star \mid x \in FV(a')\} \\
\mathsf{kill}_{\mathsf{VB}}([\mathsf{skip}]^\ell) & = & \emptyset \\
\mathsf{kill}_{\mathsf{VB}}([b]^\ell) & = & \emptyset \\
\mathsf{gen}_{\mathsf{VB}}([x := a]^\ell) & = & \mathsf{AExp}(a) \\
\mathsf{gen}_{\mathsf{VB}}([\mathsf{skip}]^\ell) & = & \emptyset \\
\mathsf{gen}_{\mathsf{VB}}([b]^\ell) & = & \mathsf{AExp}(b)
\end{array}
$$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Analysis of the Program



$$
\begin{array}{lcll}
\mathsf{VB}_\circ(\ell) & = & (\mathsf{VB}_\bullet(\ell) \backslash \mathsf{kill}_{\mathsf{VB}}(B^\ell)) \ \cup \ \mathsf{gen}_{\mathsf{VB}}(B^\ell) & \text{where } B^\ell \in \mathsf{blocks}(S_\star) \\
\mathsf{VB}_\bullet(\ell) & = & \begin{cases} \emptyset & : \ \text{if } \ell = \mathsf{final}(S_\star) \\ \bigcap\{\mathsf{VB}_\circ(\ell') | (\ell', \ell) \in \mathsf{flow}^R(S_\star)\} & : \ \text{otherwise} \end{cases} &
\end{array}
$$

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Example

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

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

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Code Hoisting (CH)

...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 CH:

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

After CH:

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

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Further Reading for Chapter 2.3

Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Chapter 2, Data Flow Analysis)

Contents
Chap. 1
Chap. 2
2.1
2.2
2.3
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Chapter 3

Taxonomy of DFA-Analyses

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Taxonomy of Classical DFA-Analyses

Analysis	may	must
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Analysis	may	must
Combination Op.	$\cup$	$\cap$
Solution of equ.	smallest	largest

Analysis	Extremal labels set	Abstract flow graph
Forward	$\{\mathsf{init}(S_\star)\}$	$\mathsf{flow}(S_\star)$
Backward	$\mathsf{final}(S_\star)$	$\mathsf{flow}^R(S_\star)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Bit Vectors and Bit Vector Analyses

The classical 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'

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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).
- 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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Some Non-Bit Vector Framework Examples (2)

- May be unitialized 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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Further Reading for Chapter 3 (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)
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 8.3, Taxonomy of Data-Flow Problems and Solution Methods)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 3 (2)

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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Part II

Intraprocedural Data Flow Analysis

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter 4

Flow Graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Programs as Flow Graphs

For program analysis, especially data flow analysis, it is usual to

- represent programs in terms of (non-deterministic) flow graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Flow Graphs

A (non-deterministic) flow graph is a tuple $G = (N, E, s, e)$ 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 successors

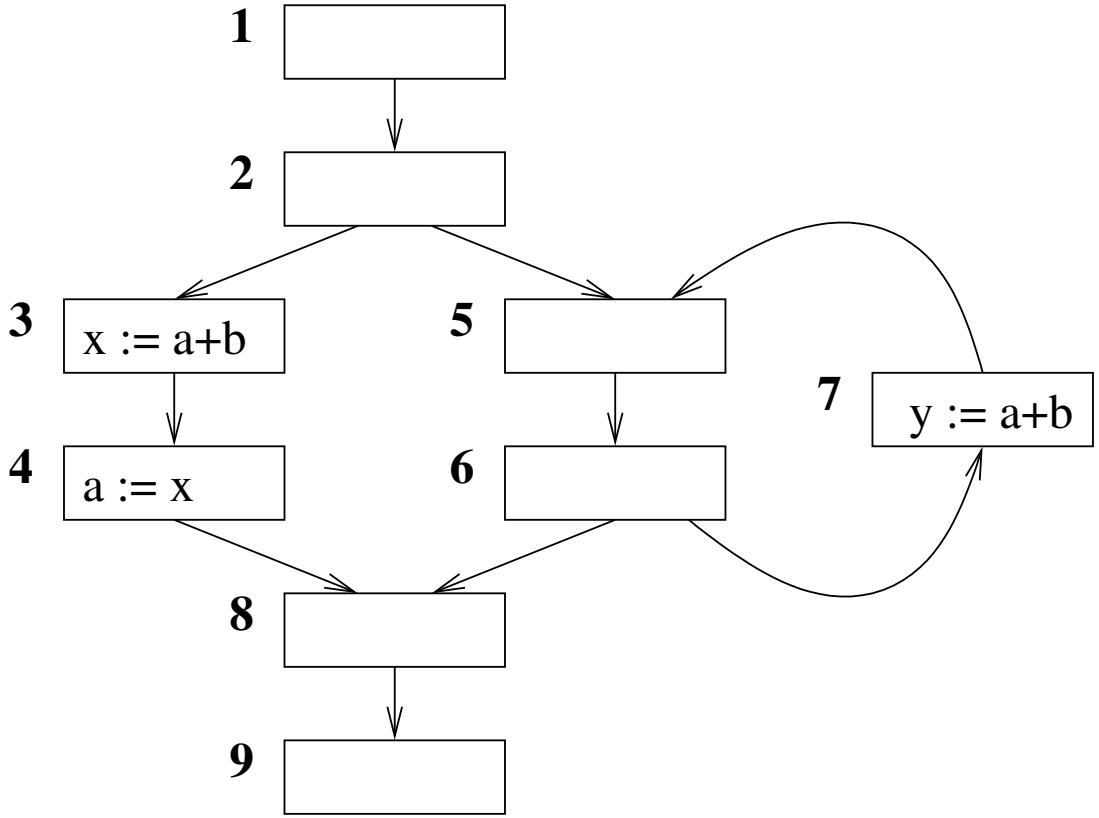
Nodes represent program points, edges represent the branching structure. Program statements (assignments, tests) can be represented by

- nodes: node-labelled flow graph
- edges: edge-labelled flow graph

where nodes and edges are labelled by single instructions or basic blocks, respectively.

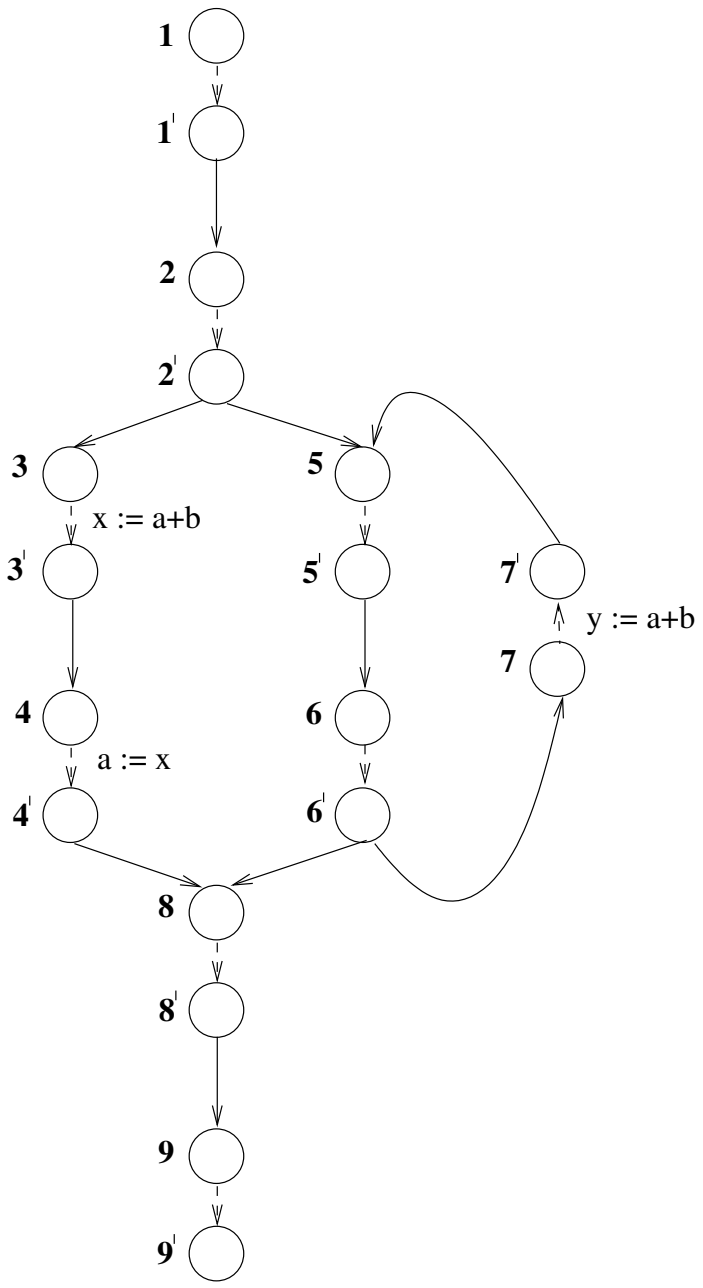
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

A Node-Labelled Flow Graph



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

An Edge-Labelled Flow Graph

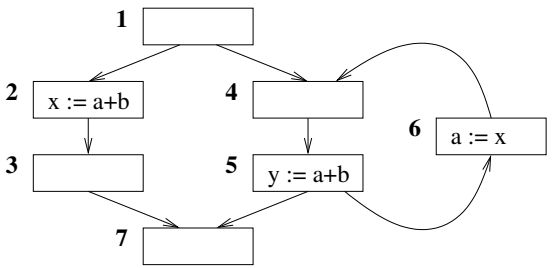


Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

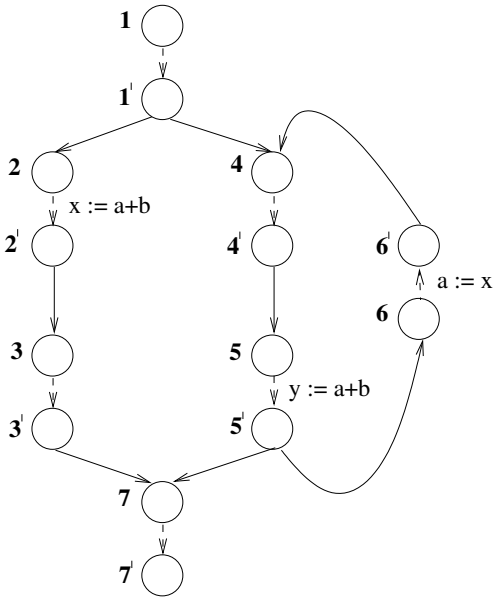
Single Instruction Flow Graphs

Node-labelled vs. edge-labelled:

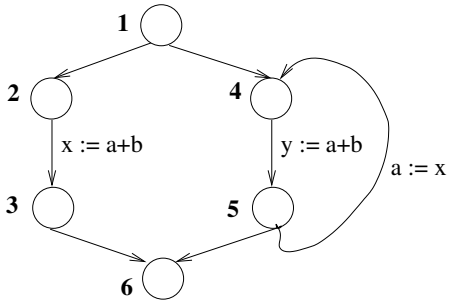
a)



b)



i) Schematisch

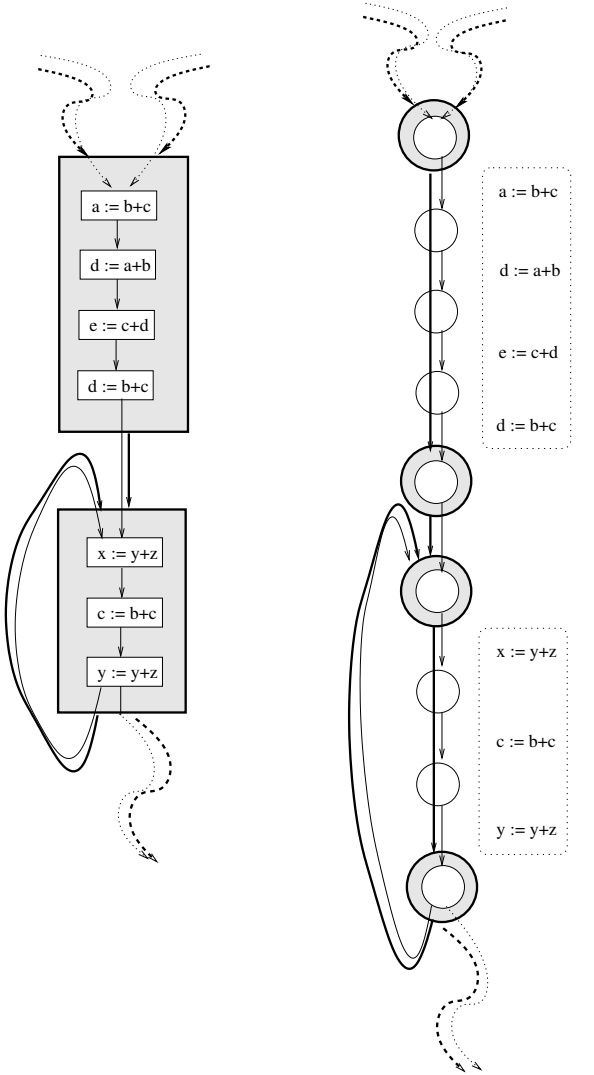


ii) "Optimiert"

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Basic Block Flow Graphs

Node-labelled vs. edge-labelled:



Node–labeled (BB–) Graph Edge–labeled (BB–) Grap

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Summing up

We distinguish:

- Node-labelled flow graphs
  - Single instruction graphs (SI graphs)
  - Basic block graphs (BB graphs)
- Edge-labelled flow graphs
  - Single instruction graphs (SI graphs)
  - Basic block graphs (BB graphs)

Later on we will preferably deal w/ edge-labelled SI graphs.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Notations

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

- $\mathbf{P}_G[m, n]$ 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 uniquely determined by the context, then we drop the index and simply write $\mathbf{P}$ instead of $\mathbf{P}_G$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 4

- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Appendix B.3.1, Graphical Intermediate Representations)
- 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.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 7, Control-Flow Analysis)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter 5

The Intraprocedural DFA Framework

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

DFA Specification, DFA Problem

Definition (5.1, DFA Specification)

A DFA specification is given by

- a (local) abstract semantics consisting of
  1. a DFA lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$
  2. a DFA functional $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$
- a start information/assertion: $c_s \in \mathcal{C}$

Definition (5.2, DFA Problem)

A DFA specification defines a DFA problem.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Practically Relevant

...are DFA problems that are

- monotonic
- distributive/additive

and satisfy the

- ascending/descending chain condition

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Properties

...of DFA functionals, DFA problems:

Definition (5.3)

A DFA functional $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$ is monotonic/distributive/additive iff for all $e \in E$ the local semantic function $[\![\ e\ ]\!]$ is monotonic/distributive/additive.

Definition (5.4)

A DFA problem is monotonic/distributive/additive iff the DFA functional $[\![\ ]\!]$ of the underlying DFA specification $(\hat{\mathcal{C}}, [\![\ ]\!], c_s)$ is monotonic/distributive/additive.

These properties induce a taxonomy of DFA problems.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Monotonicity, Distributivity, Additivity (1)

Definition (5.5, Properties of DFA Functionals)

Let $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$ be a complete (DFA) lattice and $f : \mathcal{C} \to \mathcal{C}$ a function on $\mathcal{C}$. Then f is

1. monotonic iff $\forall\, c, c' \in \mathcal{C}.\ c \sqsubseteq c' \Rightarrow f(c) \sqsubseteq f(c')$
   (Preservation of the order of elements)
2. distributive iff $\forall\, C' \subseteq \mathcal{C}.\ \ f(\bigsqcap C') = \bigsqcap \{f(c) \mid c \in C'\}$
   (Preservation of greatest lower bounds)
3. additive gdw $\forall\, C' \subseteq \mathcal{C}.\ \ f(\bigsqcup C') = \bigsqcup \{f(c) \mid c \in C'\}$
   (Preservation of least upper bounds)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Monotonicity vs. Distributivity and Additivity

We have:

Lemma (5.6)

Let $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$ *be a complete (DFA) lattice and* $f : \mathcal{C} \to \mathcal{C}$ *a function on* $\mathcal{C}$*. Then:*

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Chain Condition

Definition (5.7 Chain Condition)

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

1. descending chain condition, if every descending chain gets stationary, i.e. for each chain $c_1 \sqsupseteq c_2 \sqsupseteq \ldots \sqsupseteq c_n \sqsupseteq \ldots$ there is an index $m \geq 1$ such that $c_m = c_{m+j}$ holds for all $j \in \mathbb{N}$
2. ascending chain condition, if every ascending chain gets stationary, i.e. for each chain $c_1 \sqsubseteq c_2 \sqsubseteq \ldots \sqsubseteq c_n \sqsubseteq \ldots$ there is an index $m \geq 1$ such that $c_m = c_{m+j}$ holds for all $j \in \mathbb{N}$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Next Step

...globalizing a local abstract semantics from statements to flow graphs.

For that we introduce two (globalization) strategies:

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Chapter 5.1

The *MOP* Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The *MOP* Approach

Essential for the *MOP* approach:

Definition (5.1.1, Path extension of $[\![\ ]\!]$)

The extension of a local abstract semantics $[\![\ ]\!]$ onto paths $p = \langle e_1, e_2 \ldots, e_q \rangle$ is defined by

$$[\![\, p \,]\!] =_{df} \begin{cases} Id_{\mathcal{C}} & \text{falls } q < 1 \\ [\![\, \langle e_2, \ldots, e_q \rangle \,]\!] \circ [\![\, e_1 \,]\!] & \text{otherwise} \end{cases}$$

where $Id_{\mathcal{C}}$ denotes the identity on $\mathcal{C}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The *MOP* Solution

Definition (5.1.2, *MOP* Solution)

Let $(\hat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}})$ be the specification of a DFA problem. Then, for all nodes $n \in N$ the *MOP* solution is defined by:

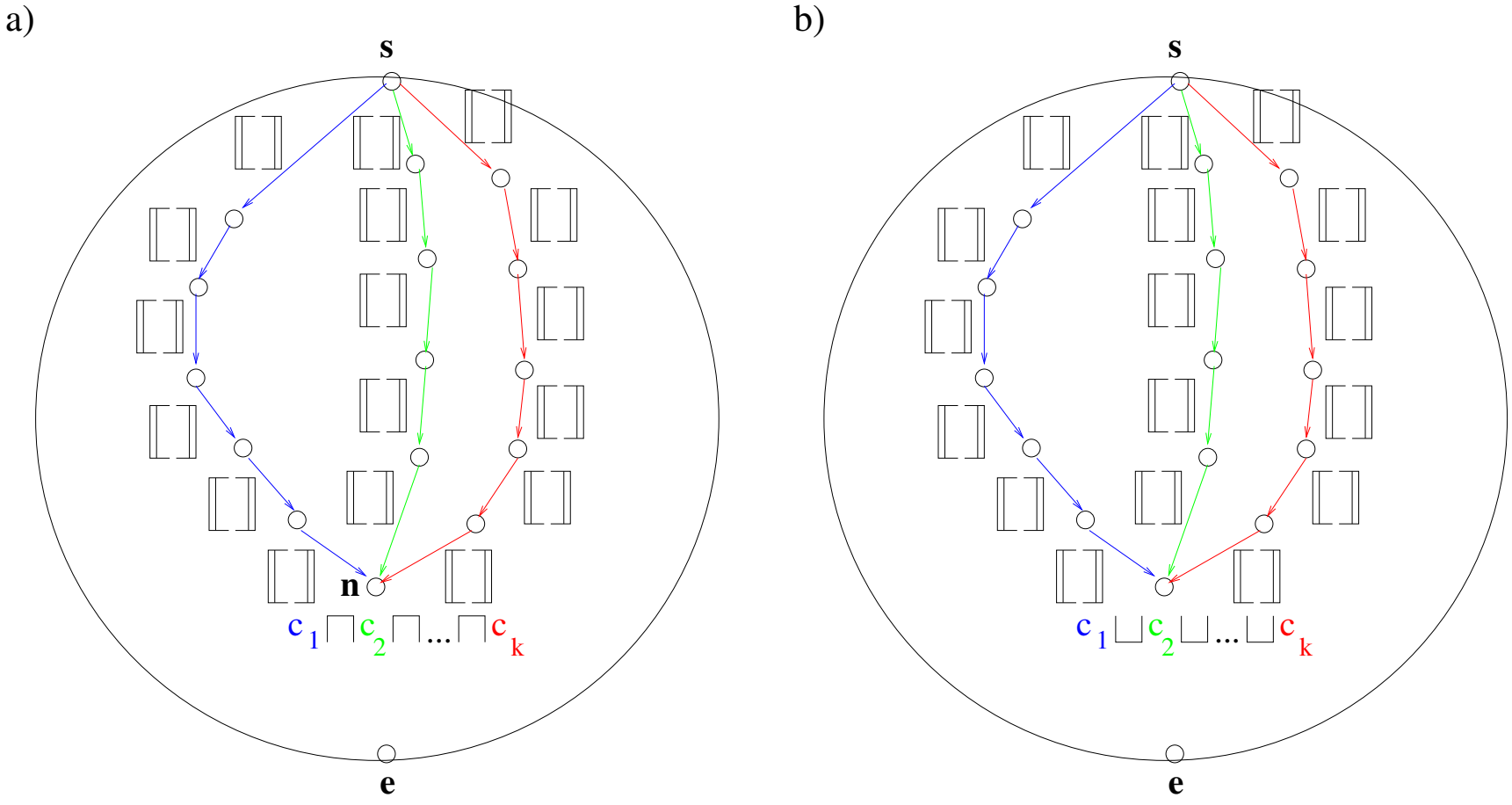
$$MOP_{(\hat{\mathcal{C}},[\![\ ]\!],c_{\mathbf{s}})}(n) = \bigsqcap \{\, [\![\, p \,]\!](c_{\mathbf{s}}) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\}$$

Essential:

- The *MOP* solution is considered the specifying solution of the DFA problem given by $(\hat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}})$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Illustration



a) Traditional DFA view:
'Meeting' infos: *MOP*
(Universally quantified)

b) Traditional AI view:
'Joining' infos: *JOP*
(Existentially quantified)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Unfortunately

...the following negative result holds:

Theorem (5.1.1, Undecidabality)

There is no algorithm A satisfying:

- *The input of A are*
  - *algorithms for the computation of the meet, the equality test, and the application of functions on the lattice elements of a monotonic DFA framework*
  - *a DFA problem p specified by an instance of a DFA specification* $(\mathcal{C}, [\![\ \ ]\!], c_{\mathsf{s}})$
- *The output of A is the MOP solution of p.*

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Towards the *MaxFP* Approach

Because of the preceding negative result we introduce a second globalization approach of a local abstract semantics, the *MaxFP* approach.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Chapter 5.2

The *MaxFP* Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The *MaxFP* Approach

Essential for the *MaxFP* approach:

Definition (5.2.1, *MaxFP* Equation System)

The *MaxFP* equation system is given by:

$$inf(n) = \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigsqcap \{ [\![(m,n)]\!](inf(m)) \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The *MaxFP* Solution

Definition (5.2.2, *MaxFP* Solution)

Let $(\hat{\mathcal{C}}, [\![\ ]\!], c_s)$ be the specification of a DFA problem. Then, for all nodes $n \in N$ the *MaxFP* solution is defined by:

$$MaxFP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}(n) =_{df} inf^*_{c_s}(n)$$

where $inf^*_{c_s}$ denotes the greatest solution of the *MaxFP* Equation System 5.2.1 wrth $(\hat{\mathcal{C}}, [\![\ ]\!], c_s)$.

Essential:

- The *MaxFP* solution is effectively computable under the conditions of the Termination Theorem 5.2.4). It is thus considered the computable solution of a DFA problem.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The Generic Fixed Point Algorithm 5.2.3 (1)

Input: (1) A flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, (2) a DFA problem given by a (local) abstract semantics consisting of a DFA lattice $\mathcal{C}$, a DFA functional $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$, and (3) a start information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: The *MaxFP* solution, if the preconditions of the Termination Theorem 5.2.4 hold. Depending on the properties of the DFA functional we have:
(i) $[\![\ ]\!]$ is distributive: The variable *inf* stores for each node the strongest post-condition wrt the start information $c_{\mathbf{s}}$.
(ii) $[\![\ ]\!]$ is monotonic: The variable *inf* stores for each node a safe (i.e. lower) approximation of the strongest post-condition wrt the start information $c_{\mathbf{s}}$.

Remark: The variable *workset* controls the iterative process. Its elements are nodes of G, whose annotation has recently been updated.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The Generic Fixed Point Algorithm 5.2.3 (2)

```
( Prologue: Initializing inf and workset )
FORALL n ∈ N\{s} DO inf[n] := ⊤ OD;
inf[s] := c_s;
workset := N;
( Main loop: The iterative fixed point computation )
WHILE workset ≠ ∅ DO
    CHOOSE m ∈ workset;
        workset := workset\{ m };
        ( Update the annotations of all successors of node m )
        FORALL n ∈ succ(m) DO
            meet := ⟦ (m, n) ⟧(inf[m]) ⊓ inf[n];
            IF inf[n] ⊐ meet
                THEN
                    inf[n] := meet;
                    workset := workset ∪ { n }
            FI
        OD ESOOHC OD.
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Effectivity

Theorem (5.2.4, Termination)

The Generic Fixed Point Algorithm 5.2.3 terminates with the $MaxFP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}$ *solution, if*

a) *the DFA functional* $[\![\ ]\!]$ *is monotonic*

b) *the DFA lattice* $\hat{\mathcal{C}}$ *satisfies the descending chain condition.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Note

The Generic Fixed Point Algorithm 5.2.3 is formulated for

- universally quantified ("distributive") forward problems

The other three variants of DFA problems of

- existentially quantified ("additive") problems
- backward problems

can be captured and solved with the Generic Fixed Point Algorithm 5.2.3 by "turning round"

- the lattice (by replacing of $\sqsubset$ by $\sqsupset$)
- the flow graph (by reversing all edges)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Chapter 5.3

Coincidence and Safety Theorem

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Main Results: Soundness and Completeness

The relationship of *MOP* and *MaxFP* solution

- Soundness
- Completeness

In Detail:

- Soundness:
  Does always hold $MaxFP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)} \sqsubseteq MOP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}$?
- Completeness:
  Does always hold $MaxFP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)} \sqsupseteq MOP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}$?

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Soundness

Theorem (5.3.1, Safety)

The MaxFP solution is a safe (conservative), i.e. lower approximation of the MOP solution, i.e.,

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ MaxFP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}(n) \sqsubseteq MOP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}(n)$$

if the DFA functional $[\![\ ]\!]$ *is monotonic.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Completeness (and simultaneously Soundness)

Theorem (5.3.2, Coincidence)

The MaxFP solution coincides with the MOP solution, i.e.,

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ MaxFP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}(n) = MOP_{(\hat{\mathcal{C}},[\![\ ]\!],c_s)}(n)$$

if the DFA functional $[\![\ ]\!]$ *is distributive.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Note

In the context of DFA

- Safety
- Coincidence

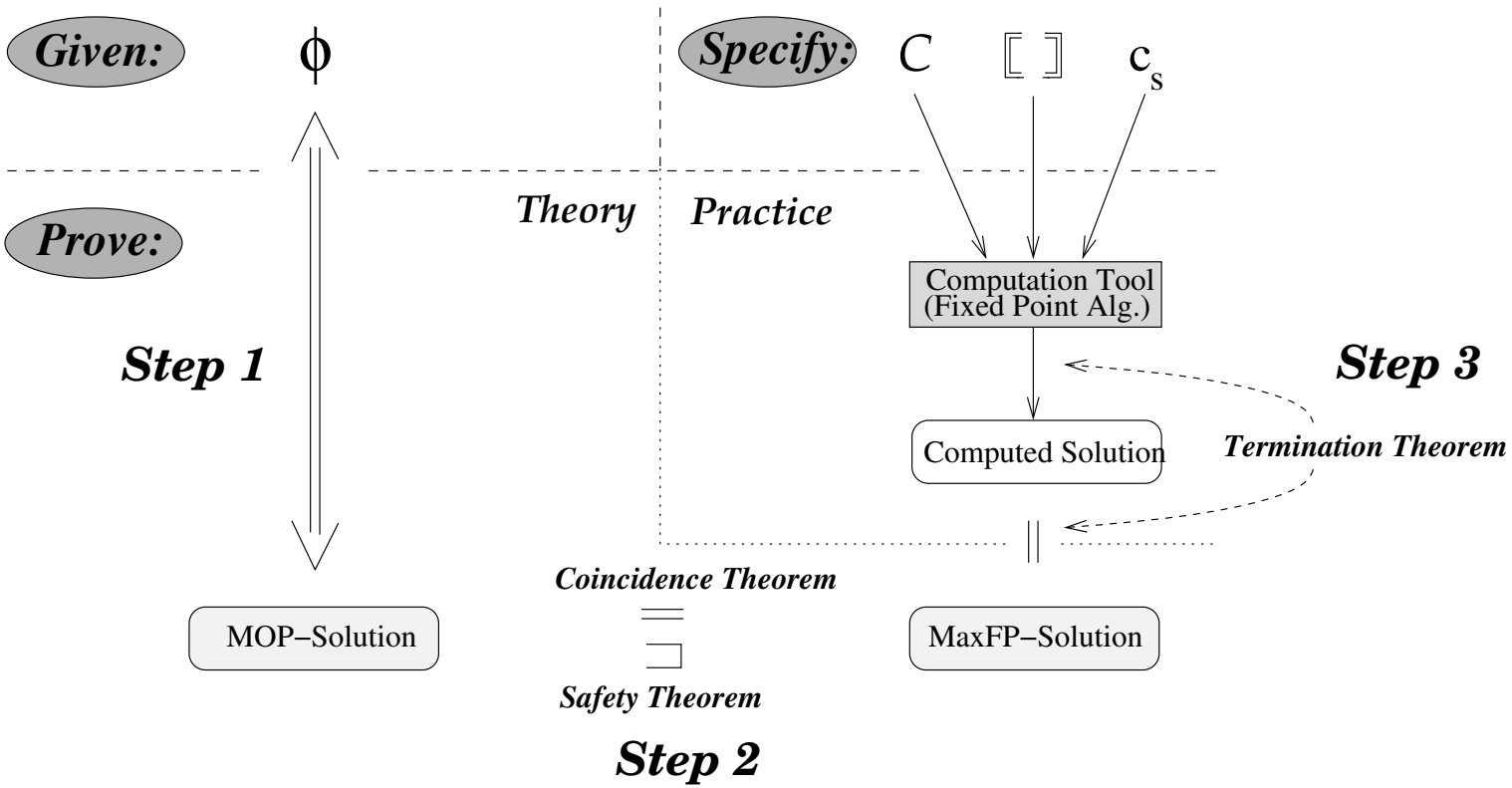
are traditionally used instead of

- Soundness
- Completeness

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Intraprocedural DFA at a Glance (1)

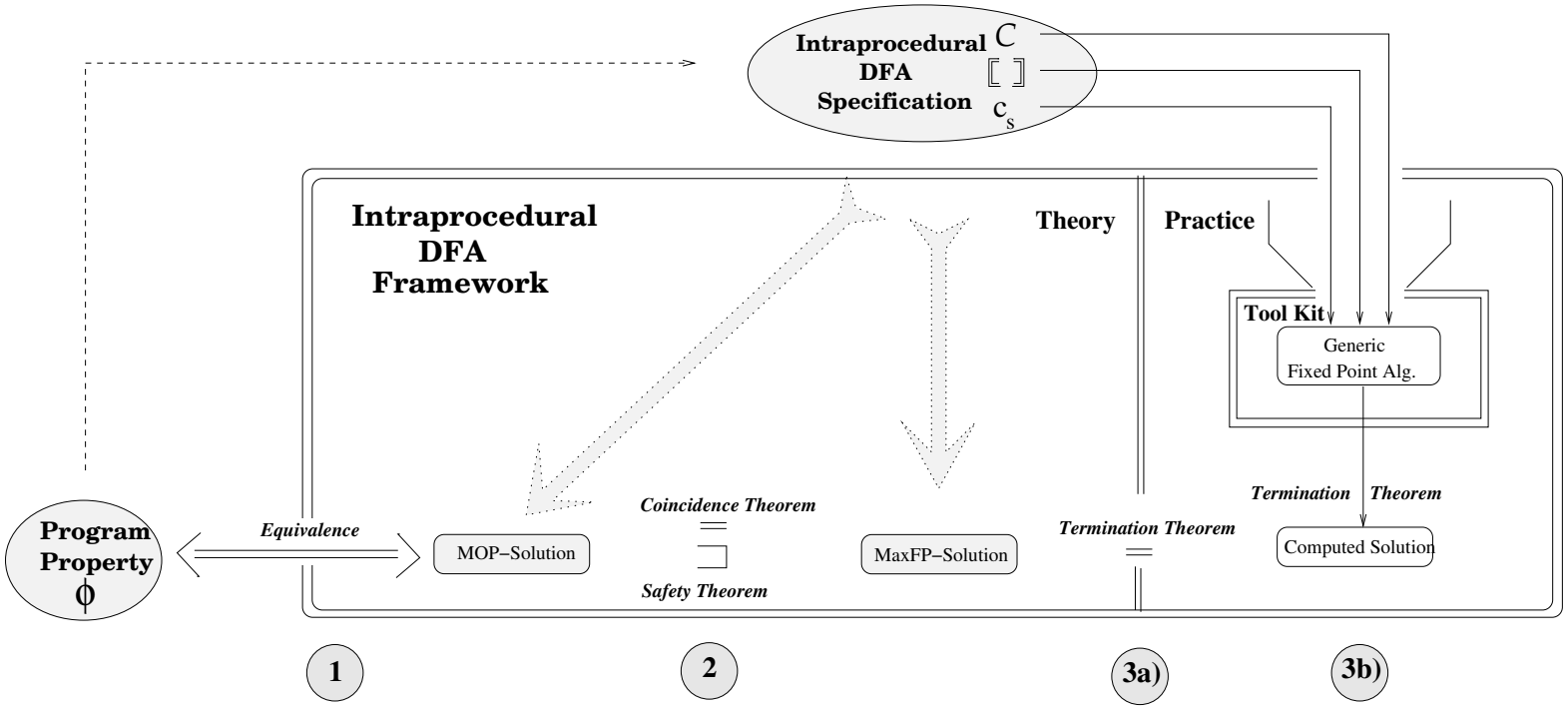
The schematic view:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Intraprocedural DFA at a Glance (2)

Focused on the framework/toolkit view:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Chapter 5.4

Examples: Available Expressions, Simple Constants

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Two Prototypical DFA Problems

- Available Expressions
  $\rightsquigarrow$ a canonical example of a distributive DFA problem
- Simple Constants
  $\rightsquigarrow$ a canonical example of a monotonic DFA problem

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Chapter 5.4.1

Available Expressions

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Available Expressions

...a typical distributive DFA problem.

- Local abstract semantics for available expressions:
  1. DFA lattice:
     $$(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathbb{B}, \wedge, \vee, \leq, \textbf{false}, \textbf{true})$$
  2. DFA functional: $[\![\ ]\!]_{av} : E \to (\mathbb{B} \to \mathbb{B})$ defined by

$$\forall\, e \in E.\ [\![\, e \,]\!]_{av} =_{df} \begin{cases} Cst_{\textbf{true}} & \text{if } Comp_e \wedge Transp_e \\ Id_{\mathbb{B}} & \text{if } \neg Comp_e \wedge Transp_e \\ Cst_{\textbf{false}} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Notations

- $\hat{\mathbb{B}} =_{df} (\mathbb{B}, \wedge, \vee, \leq, \textbf{false}, \textbf{true})$: The lattice of Boolean values w/ **false** $\leq$ **true** and the logical $\wedge$ and $\vee$ as meet operation and join operation $\sqcap$ and $\sqcup$, respectively.
- $Cst_{\textbf{true}}$ and $Cst_{\textbf{false}}$: The constant functions "true" and "false" on $\hat{\mathbb{B}}$, respectively.
- $Id_{\mathbb{B}}$: The identity function on $\hat{\mathbb{B}}$.

...and for a fixed candidate expression t:

- $Comp_e$: t is computed by the instruction attached to edge e (i.e., t is a subexpression of the right-hand side expression)
- $Transp_e$: no operand of t is assigned a new value by the instruction attached to edge e (i.e. no operand of t occurs on the left-hand side: e is transparent for t)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Main Results

Lemma (5.4.1.1)

$[\![\ ]\!]_{av}$ *is distributive.*

Corollary (5.4.1.2)

The MOP solution and the MaxFP solution coincide for available expressions.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

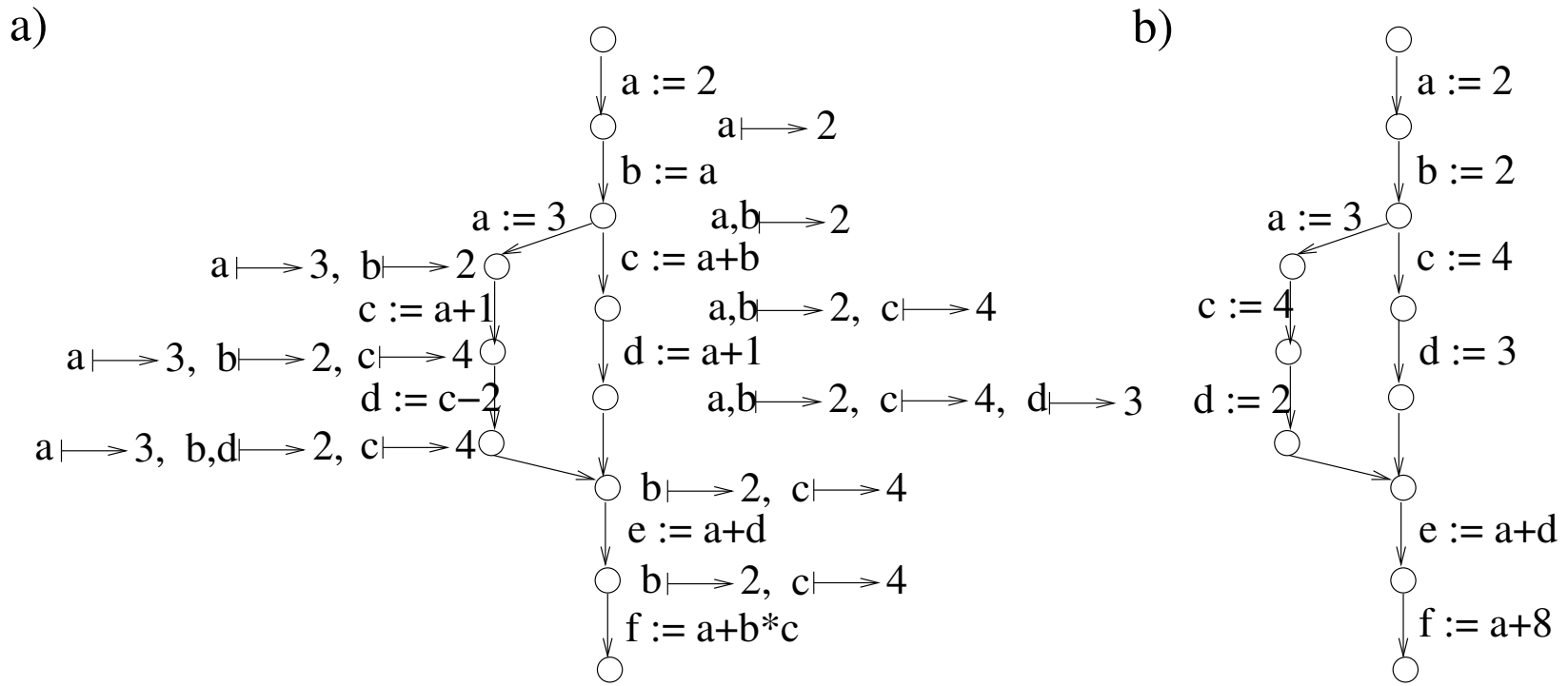
Chapter 5.4.2

Simple Constants

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Simple Constants

...a typical monotonic (but non distributive) DFA problem.



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Abstract Semantics for Simple Constants

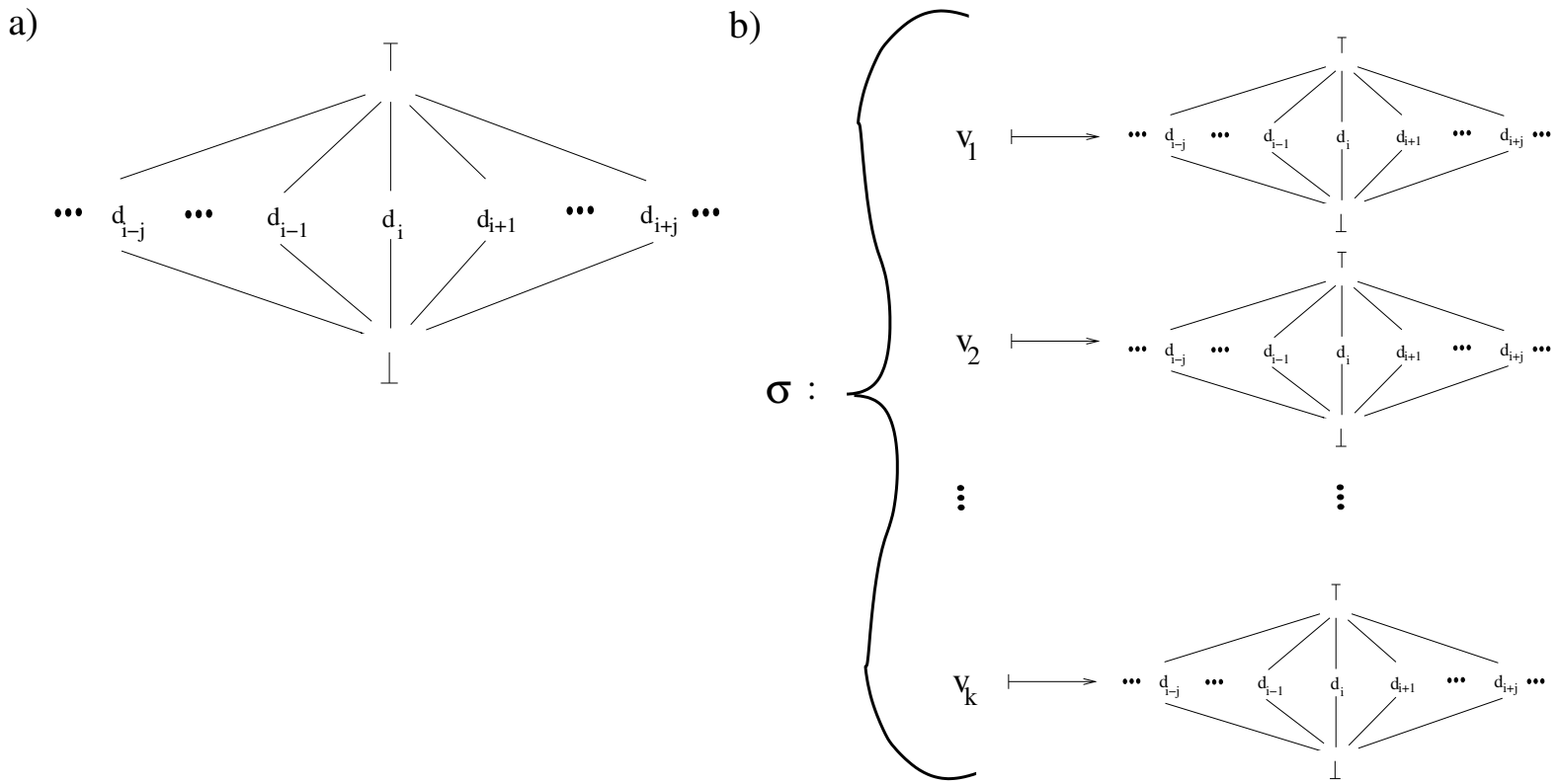
- Local abstract semantics for simple constants:
  1. DFA lattice: $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\Sigma, \sqcap, \sqcup, \sqsubseteq, \sigma_\bot, \sigma_\top)$
  2. DFA functional: $[\![\ ]\!]_{sc} : E \to (\Sigma \to \Sigma)$ defined by

$$\forall\, e \in E.\ [\![\, e \,]\!]_{sc} =_{df} \theta_e$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

DFA Lattice for Simple Constants

The "canonical" lattice for constant propagation and folding:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The Semantics of Terms

The semantics of terms $t \in \mathbf{T}$ is given by the inductively defined evaluation function

$$\mathcal{E} : \mathbf{T} \to (\Sigma \to \mathbf{D})$$

$$\forall t \in \mathbf{T}\, \forall \sigma \in \Sigma.\, \mathcal{E}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t = x \in \mathbf{V} \\ I_0(c) & \text{if } t = c \in \mathbf{C} \\ I_0(op)(\mathcal{E}(t_1)(\sigma), \ldots, \mathcal{E}(t_r)(\sigma)) & \\ & \text{if } t = op(t_1, \ldots, t_r) \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Some Yet to be defined Notions

...to complete the definition of the semantics of terms:

- Term syntax
- Interpretation
- State

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The Syntax of Terms (1)

Let

- **V** be a set of variables
- **Op** be a set of n-ary operators, $n \geq 0$, and $\mathbf{C} \subseteq \mathbf{Op}$ be the set of 0-ary operators, the so-called constants in **Op**.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The Syntax of Terms (2)

We legen fest:

1. Each variable $v \in \mathbf{V}$ and each constant $c \in \mathbf{C}$ is a term.
2. If $op \in \mathbf{Op}$ is an n-ary operator, $n \geq 1$, and $t_1, \ldots, t_n$ are terms, then $op(t_1, \ldots, t_n)$ is a term, too.
3. There are no other terms in addition to those that can be constructed by the above two rules.

The set of all terms is denoted by $\mathbf{T}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Interpretation

Let $\mathbf{D}'$ be a suitable data domain (e.g. the set of integers), let $\perp$ and $\top$ be two distinguished elements w/ $\perp, \top \notin \mathbf{D}'$, and let $\mathbf{D} =_{df} \mathbf{D}' \cup \{\perp, \top\}$.

An interpretation on $\mathbf{T}$ and $\mathbf{D}$ is a tuple $I \equiv (\mathbf{D}, I_0)$, where

- I_0 is a function, which associates w/ each 0-ary operator $c \in \mathbf{Op}$ a datum $I_0(c) \in \mathbf{D}'$ and w/ each n-ary operator $op \in \mathbf{Op}$, $n \geq 1$, a total function $I_0(op) : \mathbf{D}^n \to \mathbf{D}$, which is assumed to be strict (i.e. $I_0(op)(d_1, \ldots, d_n) = \perp$, if there is a $j \in \{1, \ldots, n\}$ w/ $d_j = \perp$)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Set of States

$$\Sigma =_{df} \{ \sigma \mid \sigma : \mathbf{V} \to \mathbf{D} \}$$

...denotes the set of states, i.e. the set of mappings σ from the set of variables $\mathbf{V}$ to a suitable data domain $\mathbf{D}$ (that is not specified in more detail here).

In particular

- $\sigma_\perp$: ...denotes the totally undefined state of Σ that is defined as follows: $\forall\, v \in \mathbf{V}.\ \sigma_\perp(v) = \perp$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

The State Transformation Function

The state transformation function

$$\theta_\iota : \Sigma \to \Sigma, \quad \iota \equiv x := t$$

is defined by:

$$\forall\, \sigma \in \Sigma\ \forall\, y \in \mathbf{V}.\ \theta_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{falls } y = x \\ \sigma(y) & \text{sonst} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Main Results

Lemma (5.4.2.1)

$[\![\ ]\!]_{sc}$ *is monotonic.*

Note: Distributivity does not hold! (Excercise)

Corollary (5.4.2.2)

The MOP solution and the MaxFP solution do in general not coincide. The MaxFP solution, however, is always a safe approximation of the MOP solution for simple constants.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Further Reading for Chapter 5 (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 Analyis)
- F. E. Allen, J. A. Cocke. *A Program Data Flow Analysis Procedure*. Communications of the ACM 19(3):137-147, 1976.
- 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 1, Overview of Compilation; Chapter 8, Introduction to Code Optimization; Chapter 9, Data Flow Analysis)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Further Reading for Chapter 5 (2)

- C. Fecht, Helmut Seidl. *An Even Faster Solver for General Systems of Equations*. In Proceedings SAS'96, LNCS 1145, 189-204, 1996.
- C. Fecht, Helmut Seidl. *Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems*. In Proceedings ESOP'98, LNCS 1381, 90-104, 1998.
- C. Fecht, Helmut Seidl. *A Faster Solver for General Systems of Equations*. Science of Computer Programming 35(2):137-161, 1999.
- Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Further Reading for Chapter 5 (3)

- Susan Horwitz, A. Demers, T. Teitelbaum. *An Efficient General Iterative Algorithm for Dataflow Analysis*. Acta Informatica 24:679-694, 1987.
- John B. Kam, Jeffrey D. Ullman. *Global Data Flow Analysis and Iterative Algorithms*. Journal of the ACM 23:158-171, 1976.
- John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. Acta Informatica 7:305-317, 1977.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Further Reading for Chapter 5 (4)

- Marion Klein, Jens Knoop, Dirk Koschützki, Bernhard Steffen. *DFA&OPT-METAFrame: A Toolkit for Program Analysis and Optimization*. In Proceedings 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.
- 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Further Reading for Chapter 5 (5)

- Thomas J. Marlowe, Barbara G. Ryder. *Properties of Data Flow Frameworks*. Acta Informatica 28(2):121-163, 1990.
- Florian Martin. *PAG - An Efficient Program Analyzer Generator*. Journal of Software Tools for Technology Transfer 2(1):46-67, 1998.
- Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 1, Introduction to Advanced Topics; Chapter 4, Intermediate Representations; Chapter 7, Control-Flow Analysis; Chapter 8, Data Flow Analysis; Chapter 11, Introduction to Optimization; Chapter 12, Early Optimizations)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Further Reading for Chapter 5 (6)

- Flemming Nielson. *Semantics-directed Program Analysis: A Tool-maker's Perspective*. In Proceedings SAS'96, Springer-V., LNCS 1145, 2-21, 1996.
- 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Further Reading for Chapter 5 (7)

- 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)
- Barry K. Rosen. *High-level Data Flow Analysis*. Communications of the ACM 20(10):141-156, 1977.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.4
5.4.1
5.4.2
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14

Chapter 6

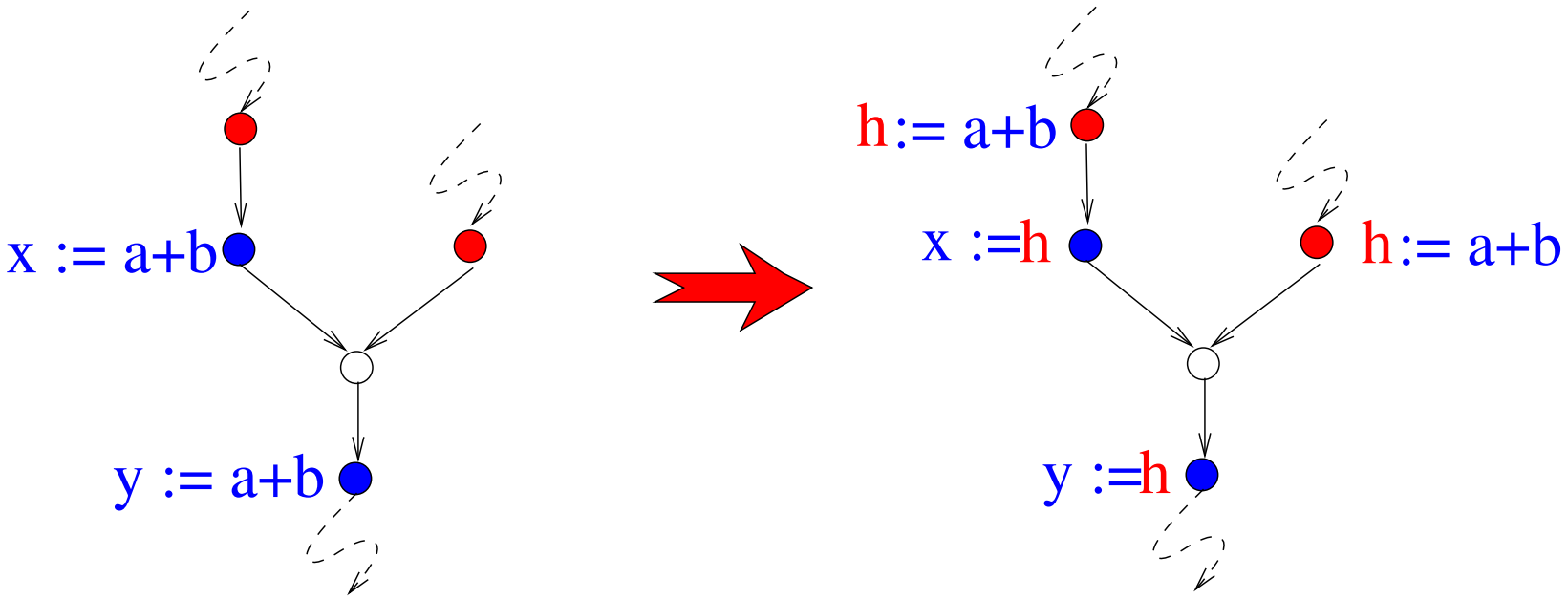
Partial Redundancy Elimination

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Partial Redundancy Elimination (PRE)

What's it all about?

...avoiding multiple (re-) computations of the same value!



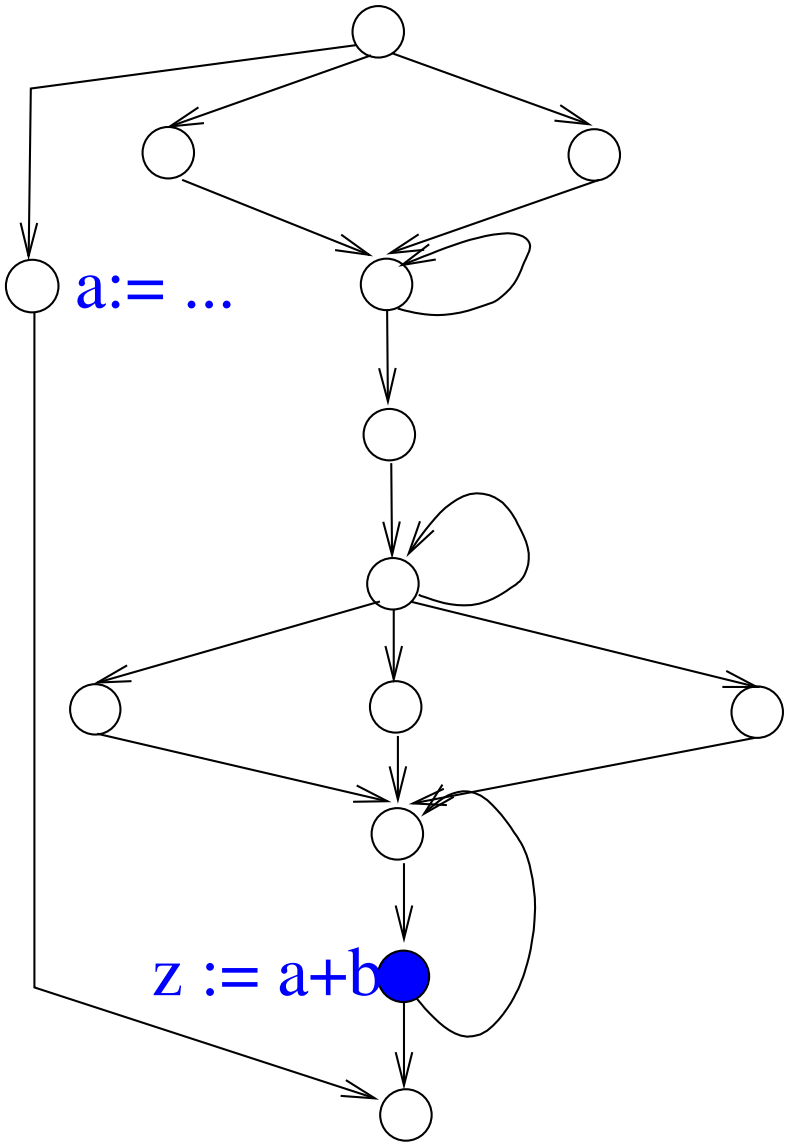
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 6.1

Motivation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

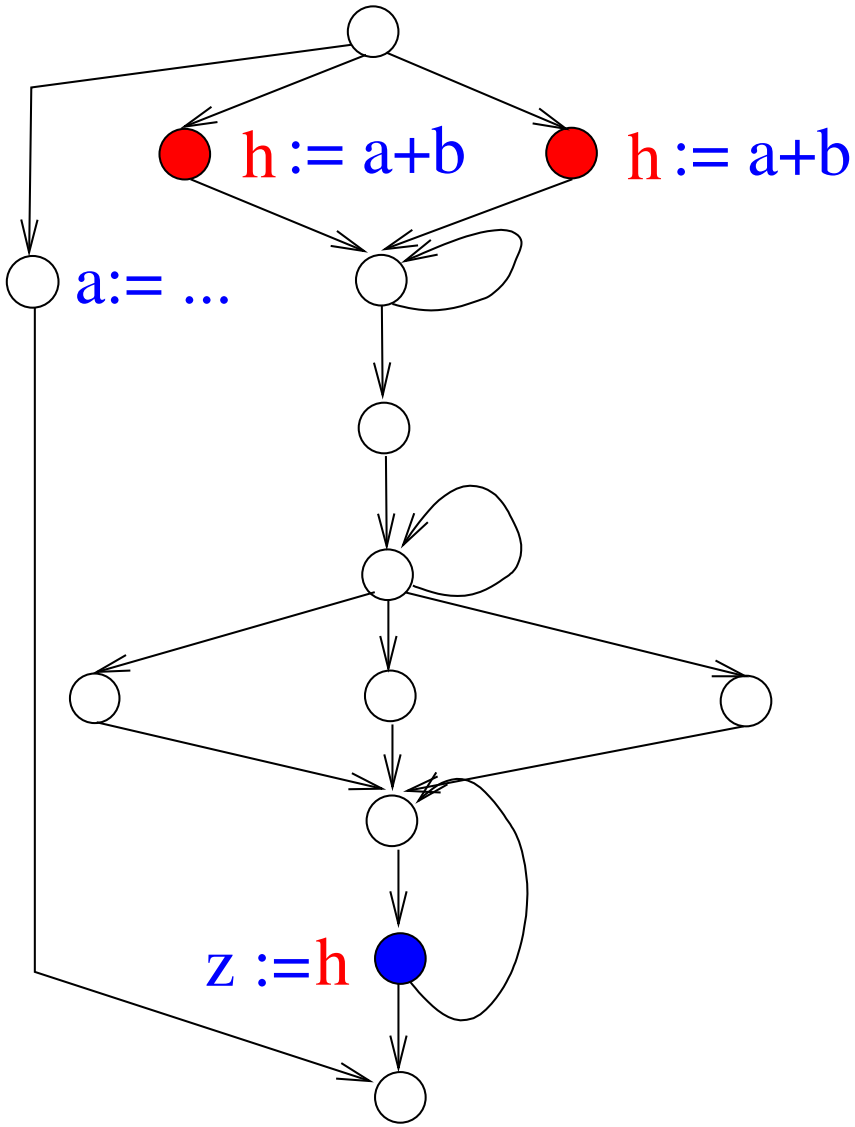
PRE – Particularly Striking for Loops



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

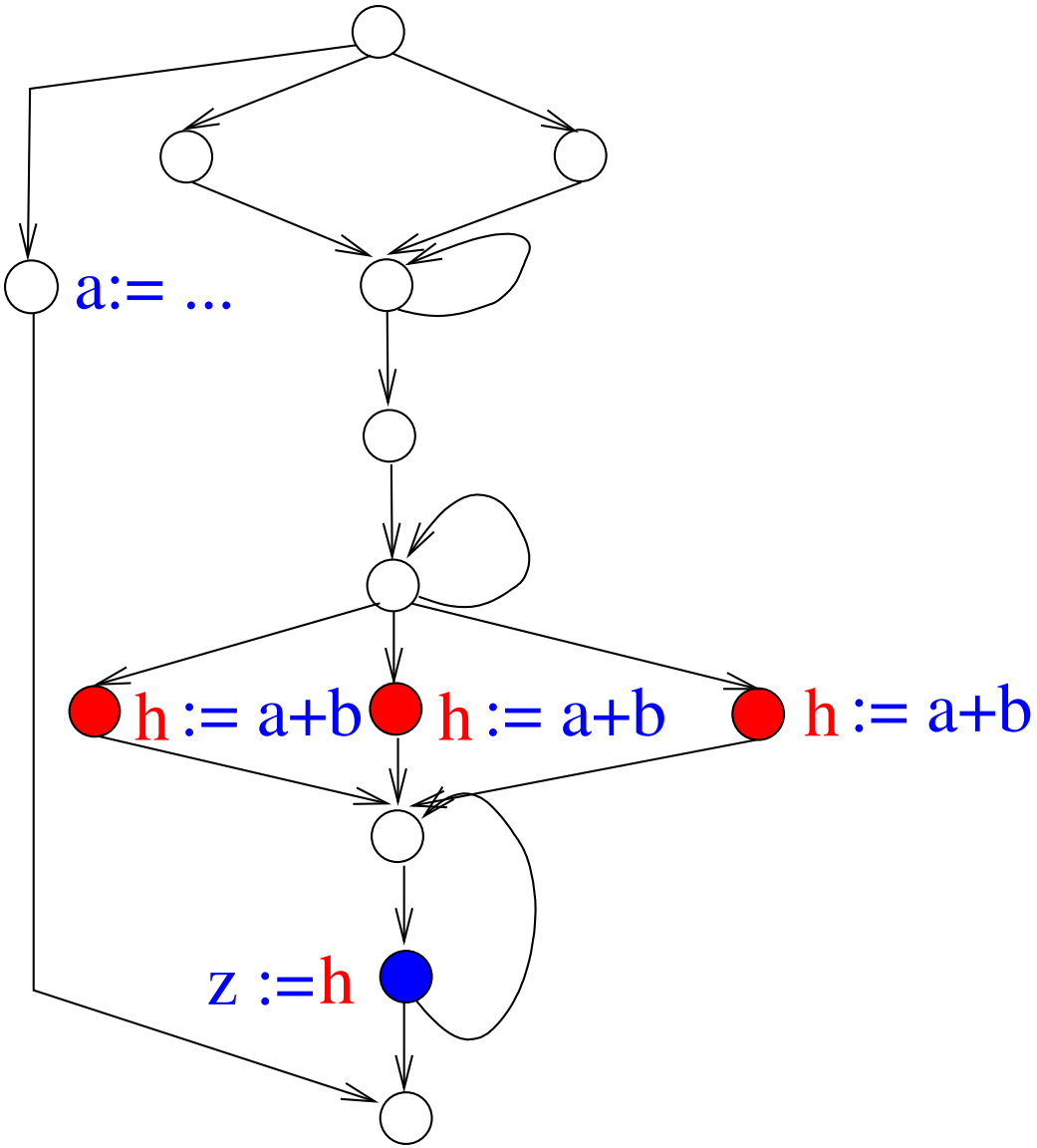
A Computationally Optimal Program

...w/out any redundancy at all!



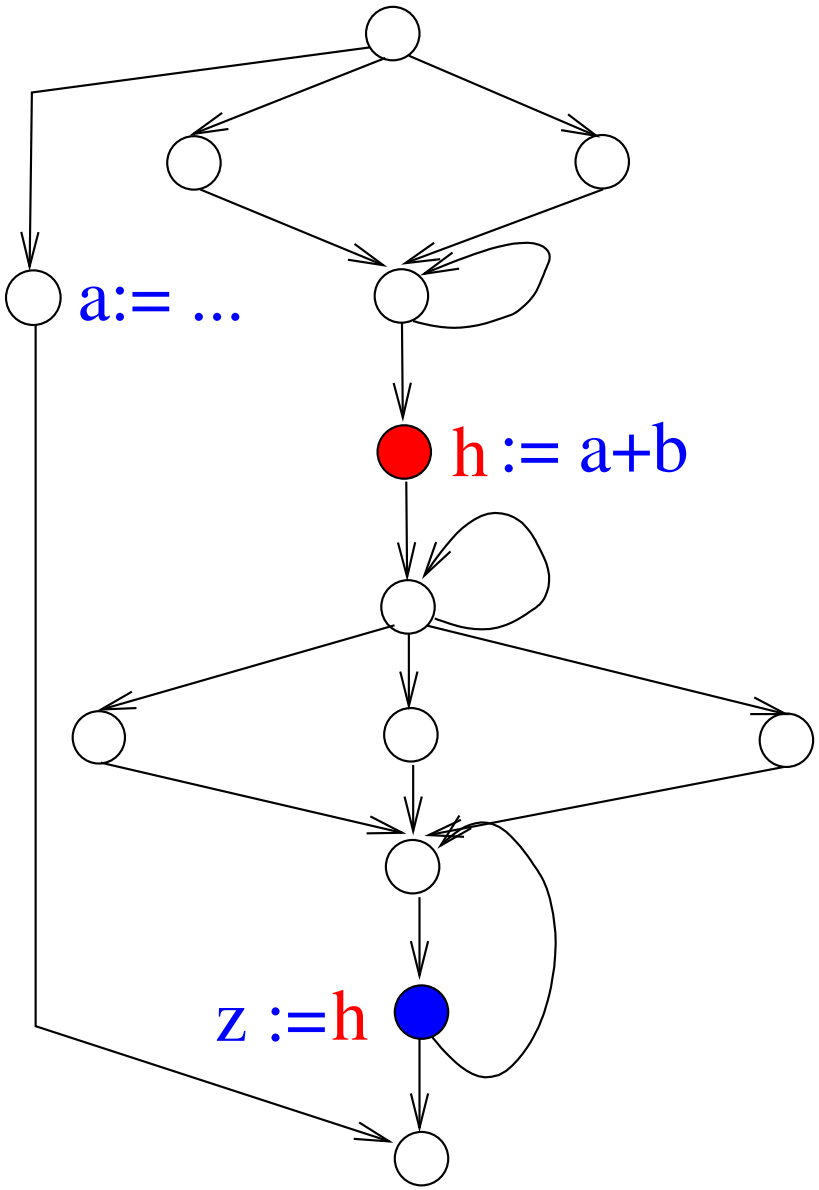
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Often there is more than one!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Which one shall PRE deliver?



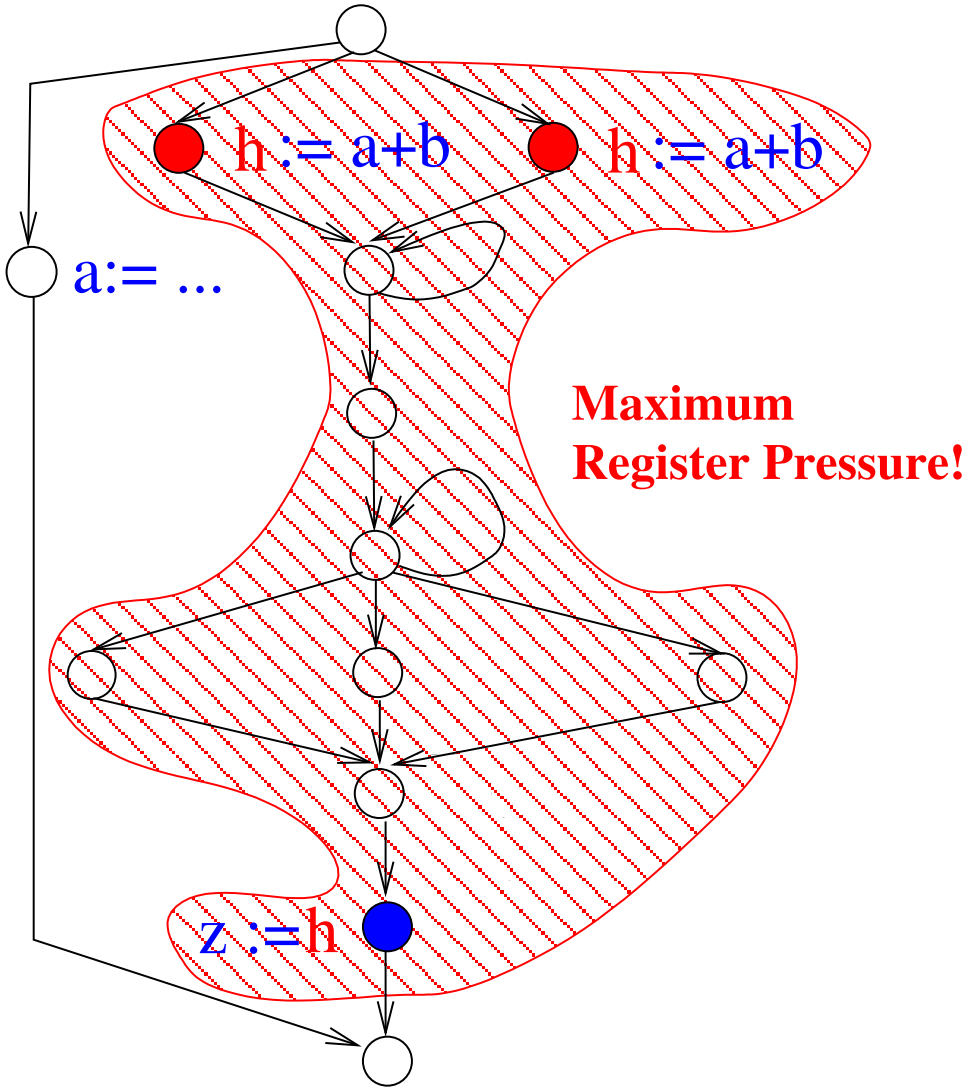
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The (Optimization) Goals make the Difference!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The first Transformation

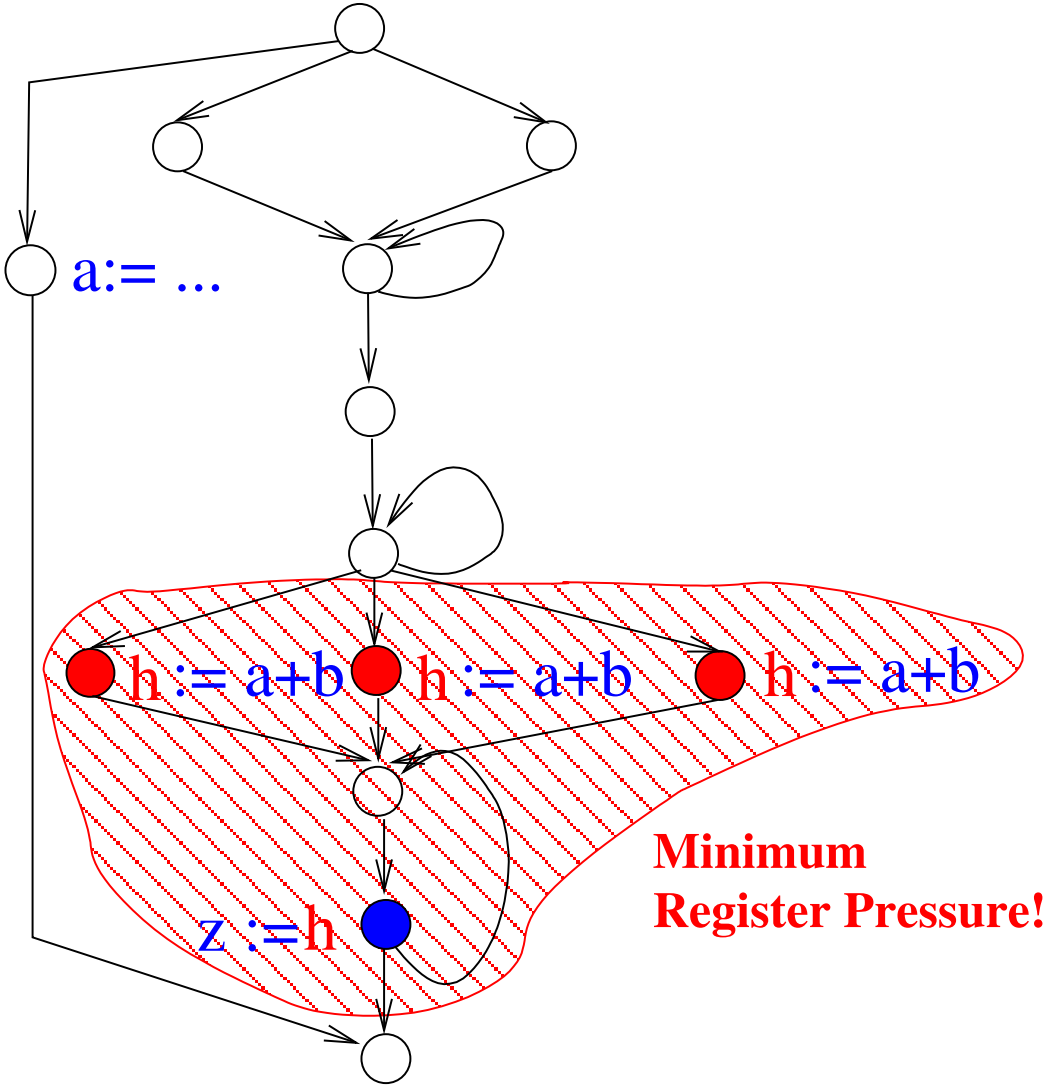
...no redundancies but maximum register pressure!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

The second Transformation

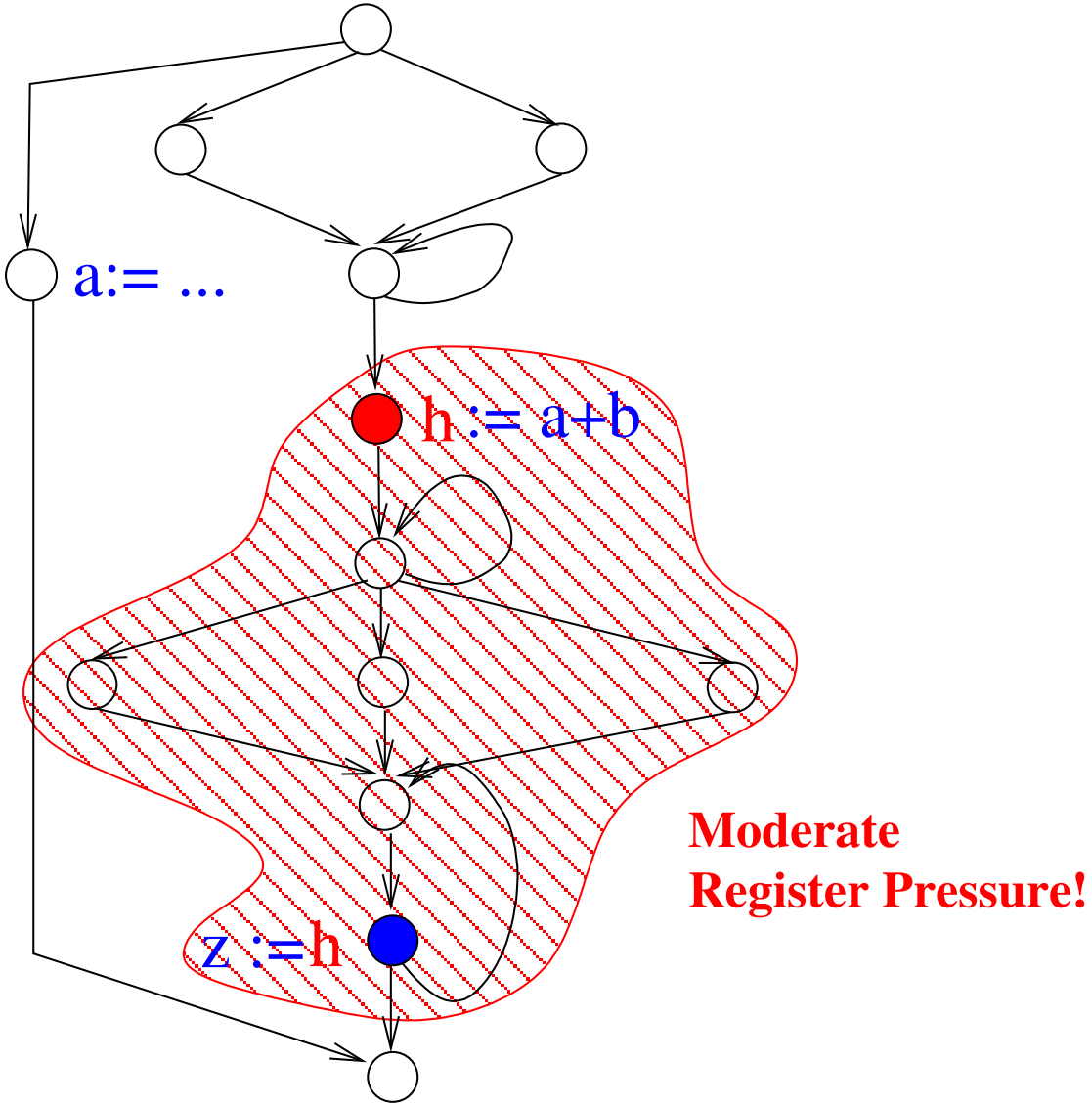
...no redundancies, too, but minimum register pressure!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The third Transformation

...no redundancies, moderate register pressure, no code replication!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

The (Optimization) Goals make the Difference!

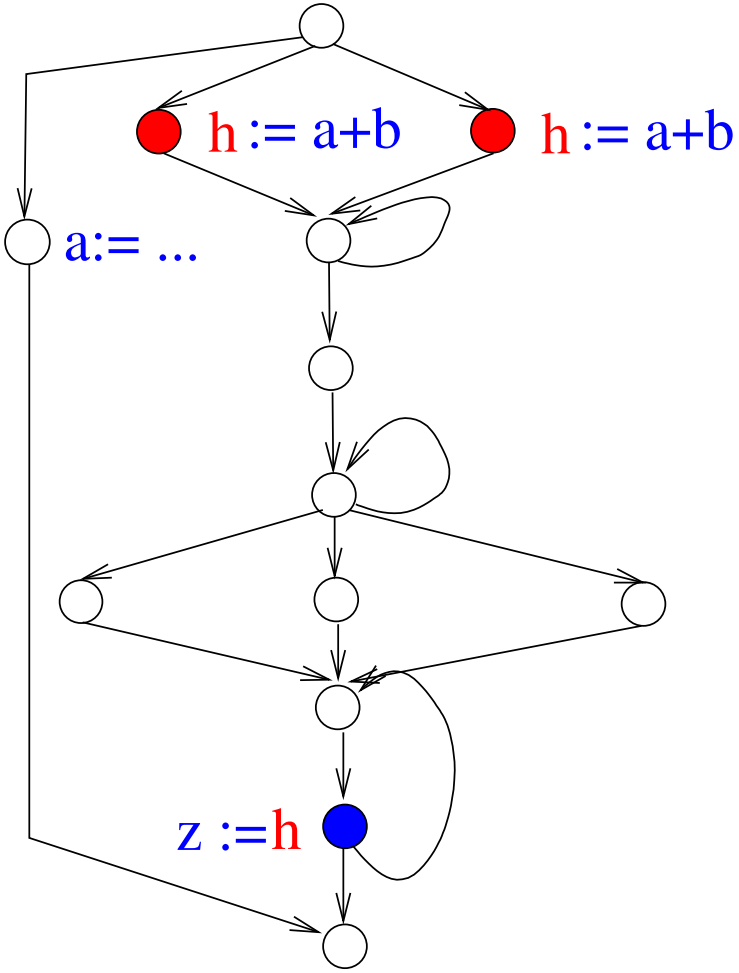
In our running example:

- Performance: Avoiding unnecessary (re-) computations
  $\rightsquigarrow$ Computational quality, computational optimality
- Register pressure: Avoiding unnecessary code motion
  $\rightsquigarrow$ Liftime quality, lifetime optimality
- Space: Avoiding unnecessary code replication
  $\rightsquigarrow$ Code size quality, code size optimality

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Result of Busy Code Motion

...placing computations as early as possible!

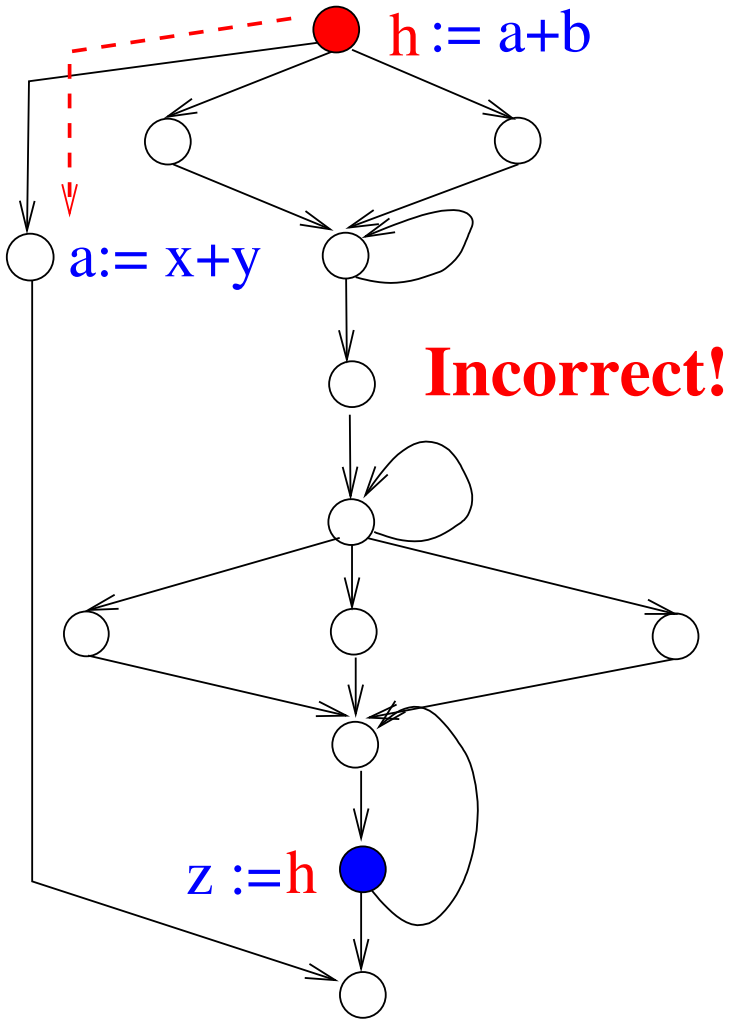


...yields computationally optimal programs.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Note: As Early as Possible

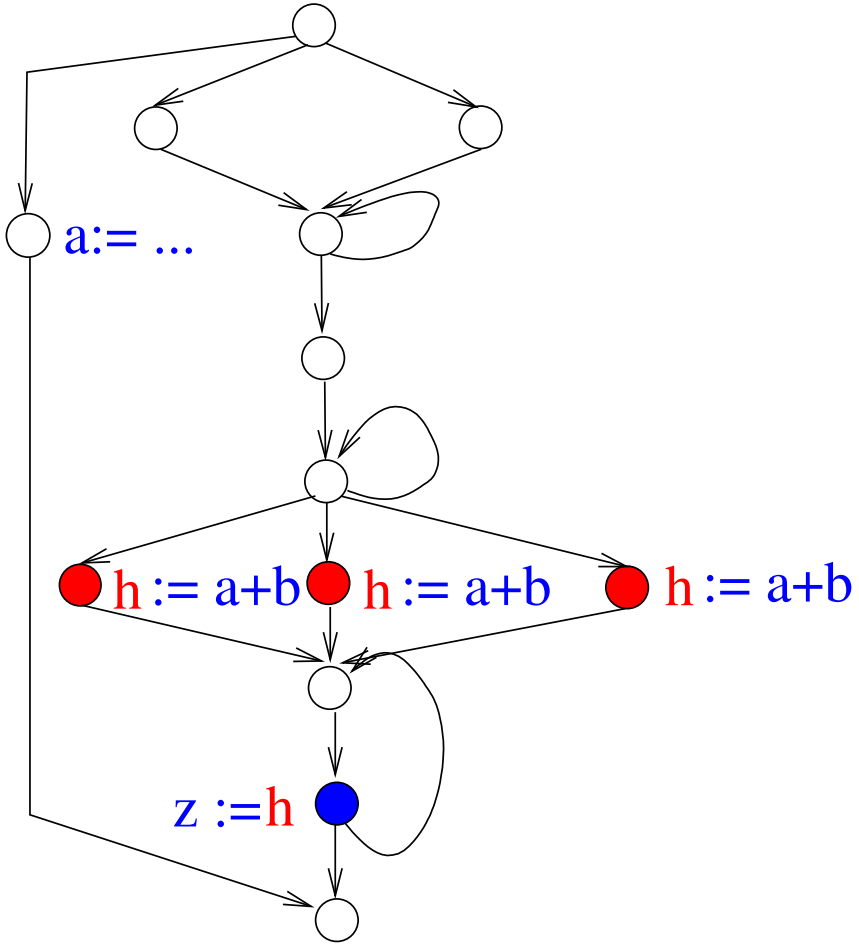
...means earliest indeed but not earlier as earliest.



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

The Result of Lazy Code Motion

...placing computations as late as possible!

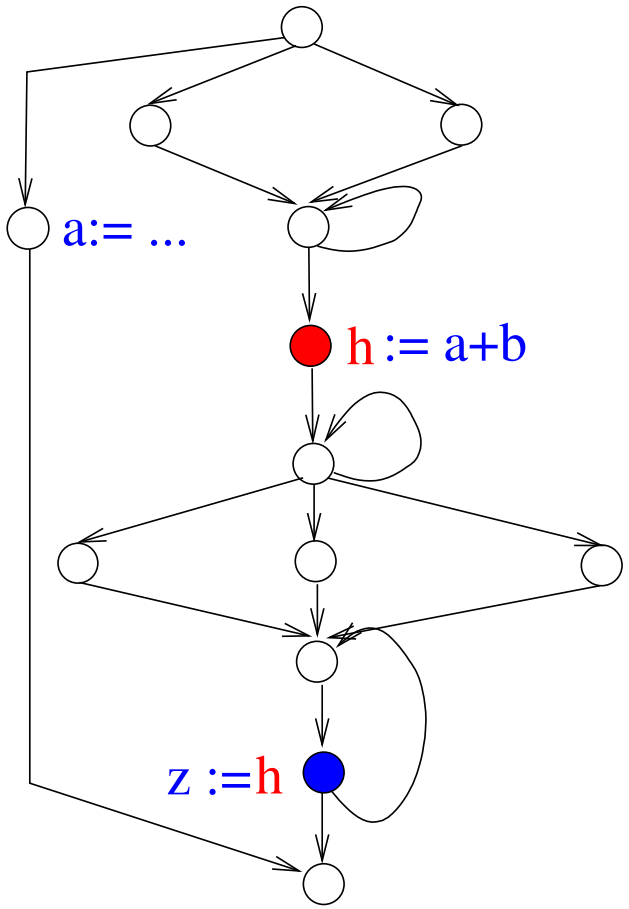


...yields computationally and lifetime optimal programs.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Result of Sparse Code Motion

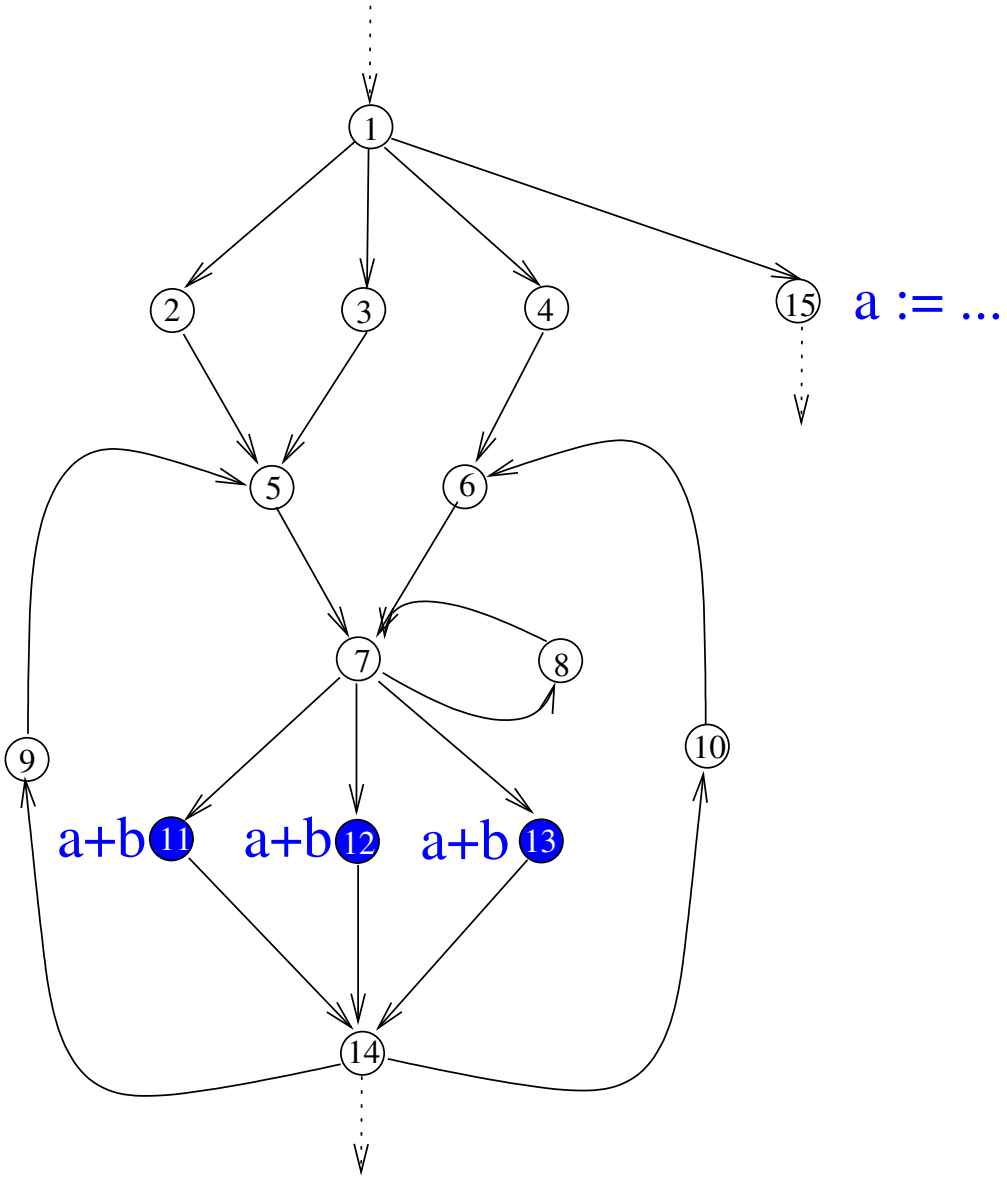
...placing computations as late as possible but as early as necessary!



...yields comp. and lifetime best code-size optimal programs.

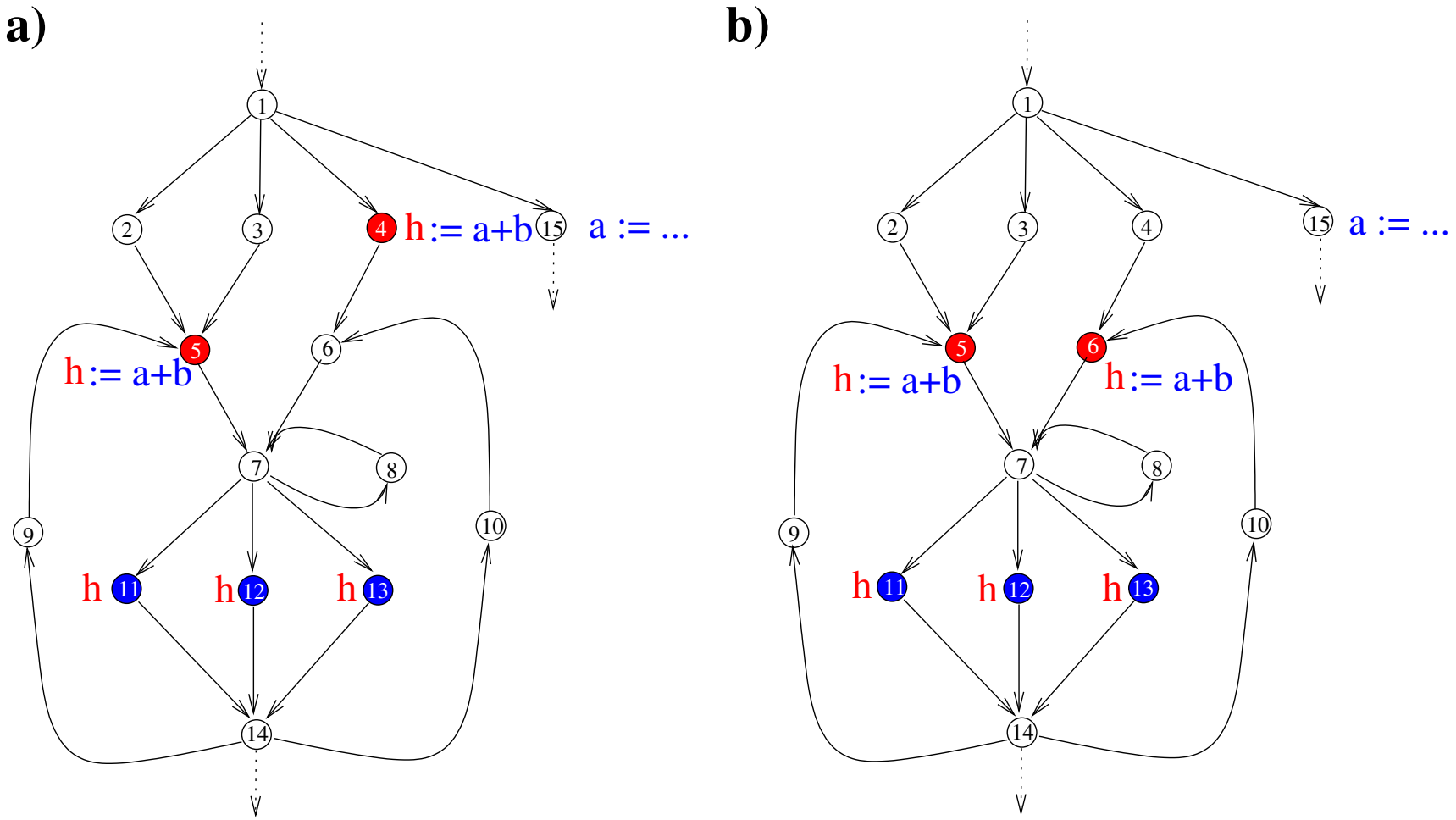
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

A More Complex Example (1)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

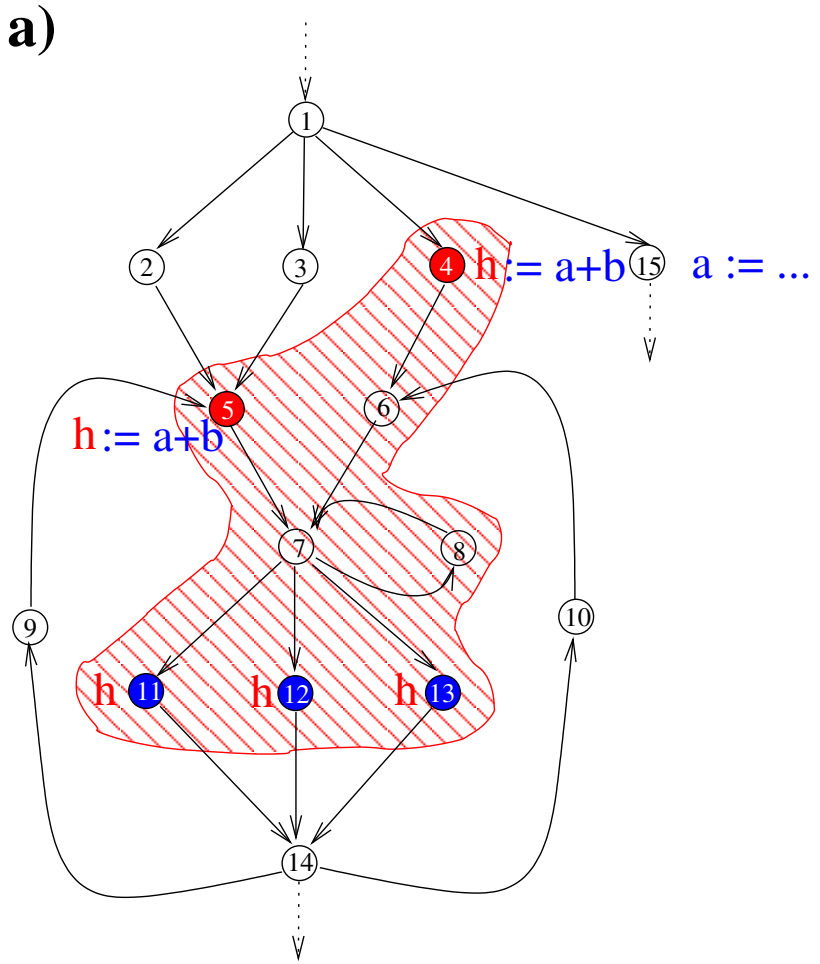
A More Complex Example (2)



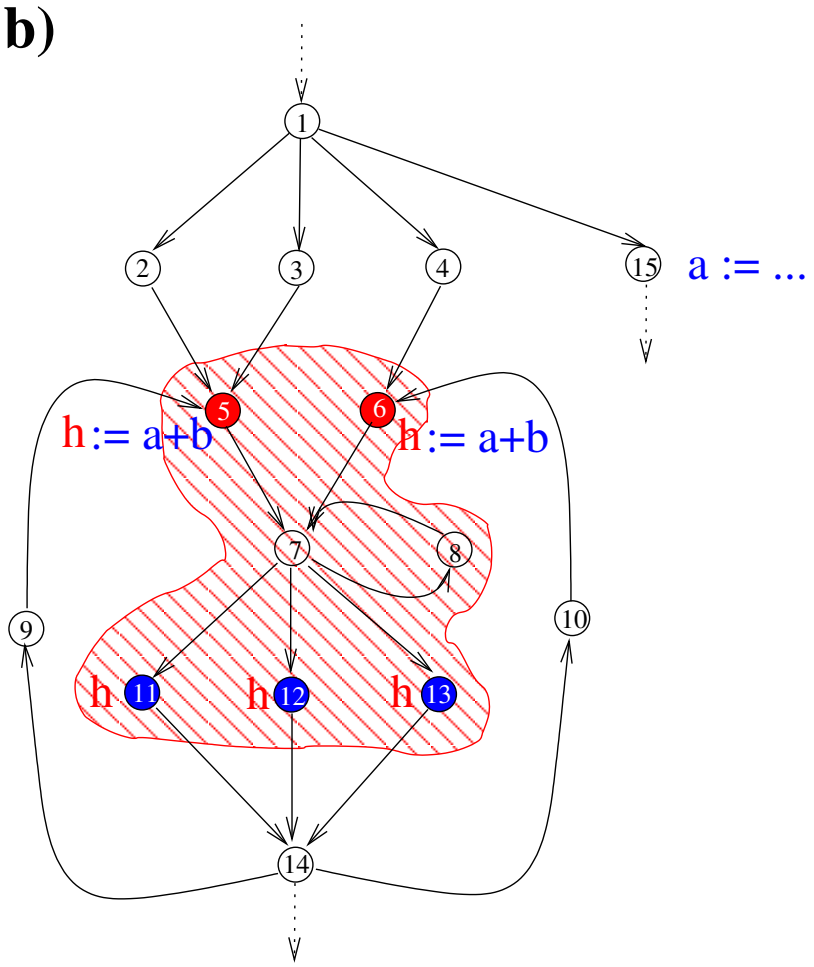
Two Code–size Optimal Programs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

A More Complex Example (3)



SQ > CQ > LQ

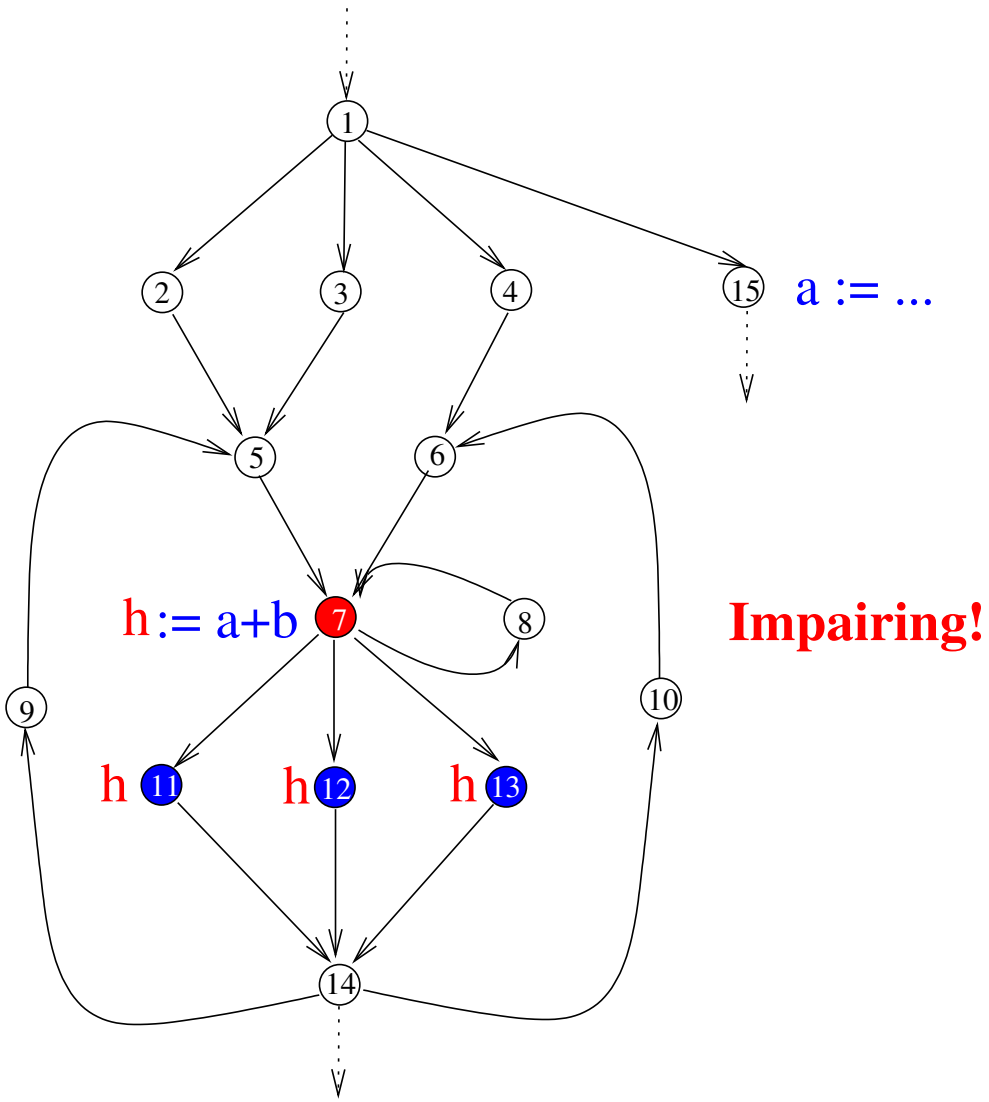


SQ > LQ > CQ

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

A More Complex Example (4)

Note: The below transformation is not desired!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Summing up

The previous examples demonstrate that in general we can not achieve

- computational, lifetime, and space optimality

at the same time.

Think, however, about the following problem (homework):

- Let P be a program containing partially redundant computations.

  Can you imagine an algorithm that always suceeds to transform P into a program P' such that P and P' have the same semantics and that P' is free of any partially redundant computation?

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 6.2

The PRE Algorithm of Morel&Renvoise

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Groundbreaking Algorithm for PRE

PRE is intrinsically tied to Etienne Morel und Claude Renvoise. The PRE algorithm they presented in 1979 can be considered the *prime father* of all code motion (CM) algorithms and was until the early 1990s the "state of the art" PRE algorithm.

Technically, the PRE algorithm of Morel and Renvoise is composed of:

- 3 uni-directional bitvector analyses (AV, ANT, PAV)
- 1 bi-directional bitvector analysis (PP)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The PRE Algorithm of Morel&Renvoise (1)

- Availability:

$$\mathbf{AVIN}(n) = \begin{cases} \mathbf{false} & \text{if } n = \mathbf{s} \\ \prod_{m \in pred(n)} \mathbf{AVOUT}(m) & \text{otherwise} \end{cases}$$

$$\mathbf{AVOUT}(n) = \mathrm{TRANSP}(n) * (\mathrm{COMP}(n) + \mathbf{AVIN}(n))$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The PRE Algorithm of Morel&Renvoise (2)

- Very Busyness (Anticipability):

$$\mathbf{ANTIN}(n) = \mathrm{COMP}(n) + \mathrm{TRANSP}(n) * \mathbf{ANTOUT}(n)$$

$$\mathbf{ANTOUT}(n) = \begin{cases} \mathbf{false} & \text{if } n = \mathbf{e} \\ \prod_{m \in succ(n)} \mathbf{ANTIN}(m) & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The PRE Algorithm of Morel&Renvoise (3)

- Partial Availability:

$$\mathbf{PAVIN}(n) = \begin{cases} \mathbf{false} & \text{if } n = \mathbf{s} \\ \sum_{m \in pred(n)} \mathbf{PAVOUT}(m) & \text{otherwise} \end{cases}$$

$$\mathbf{PAVOUT}(n) = \mathrm{TRANSP}(n) * (\mathrm{COMP}(n) + \mathbf{PAVIN}(n))$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The PRE Algorithm of Morel&Renvoise (4)

- Placement Possible:

$$\mathbf{PPIN}(n) = \begin{cases} \mathbf{false} & \text{if } n = \mathbf{s} \\ \mathbf{CONST}(n) * & \\ (\prod_{m \in pred(n)} (\mathbf{PPOUT}(m) + \mathbf{AVOUT}(m)) * & \\ (\mathrm{COMP}(n) + \mathrm{TRANSP}(n) * \mathbf{PPOUT}(n)) & \\ \qquad \text{otherwise} & \end{cases}$$

$$\mathbf{PPOUT}(n) = \begin{cases} \mathbf{false} & \text{if } n = \mathbf{e} \\ \prod_{m \in succ(n)} \mathbf{PPIN}(m) & \text{otherwise} \end{cases}$$

where

$$\mathbf{CONST}(n) =_{df} \mathbf{ANTIN}(n) * (\mathbf{PAVIN}(n) + \neg \mathrm{COMP}(n) * \mathrm{TRANSP}(n))$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The PRE Algorithm of Morel&Renvoise (5)

- Initializing temporaries where:

$$\mathbf{INSIN}(n) \quad =_{df} \quad \mathbf{false}$$

$$\mathbf{INSOUT}(n) \quad =_{df} \quad \mathbf{PPOUT}(n) * \neg\mathbf{AVOUT}(n) * (\neg\mathbf{PPIN}(n) + \neg\mathrm{TRANSP}(n))$$

- Replacing original computations where:

$$\mathbf{REPLACE}(n) \quad =_{df} \quad \mathrm{COMP}(n) * \mathbf{PPIN}(n)$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Summing up (1)

Achievements and merits of Morel&Renvoise's PRE algorithm:

- First systematic algorithm for PRE
- State-of-the-art PRE algorithm for about 15 years

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Summing up (2)

Short-comings of Morel&Renvoise's PRE algorithm:

- Conceptually
  - Fails computational optimality
    $\rightsquigarrow$ only, however, because of not splitting critical edges
  - Fails lifetime optimality
    $\rightsquigarrow$ Register pressure is heuristically dealt with
  - Fails code-size optimality
    $\rightsquigarrow$ Not considered at all (in the early days of PRE)
- Technically
  - Bi-directional
    $\rightsquigarrow$ conceptually and computationally thus more complex

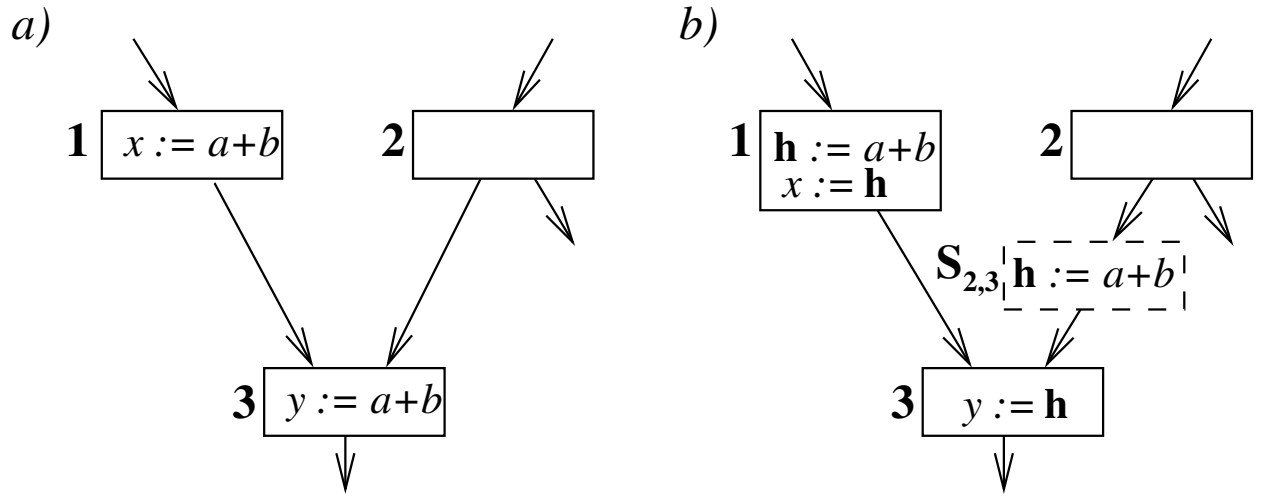
...the transformation result lies (unpredictably) between those of the BCM transformation and the LCM transformation.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Critical Edges

An edge is called critical, if it connects a branching node with a join node.

Illustration:

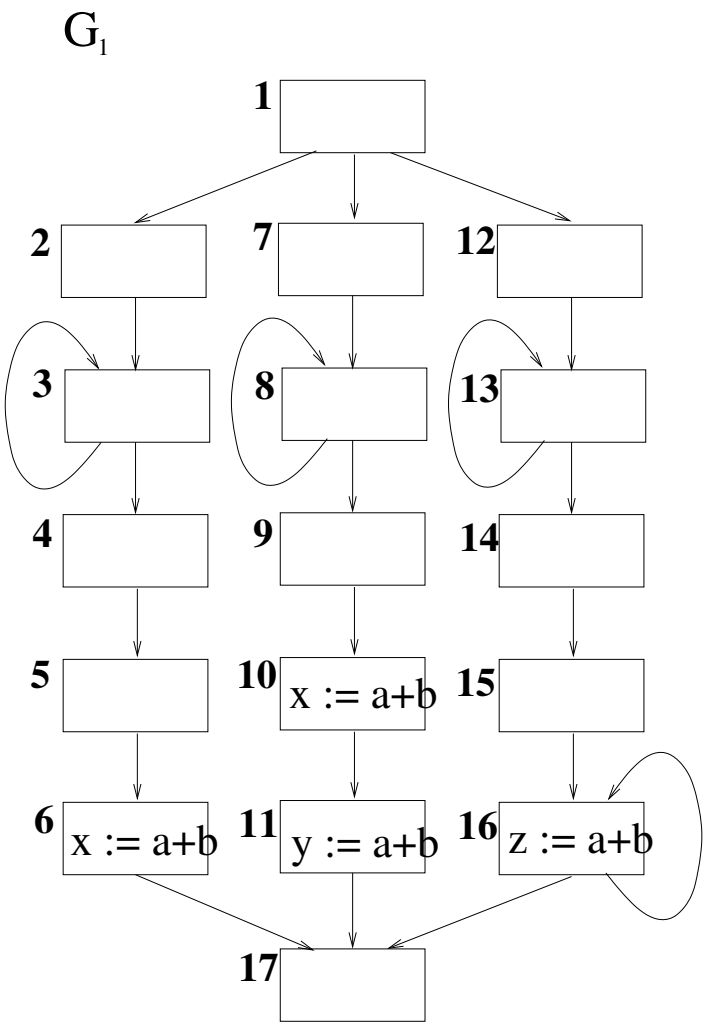


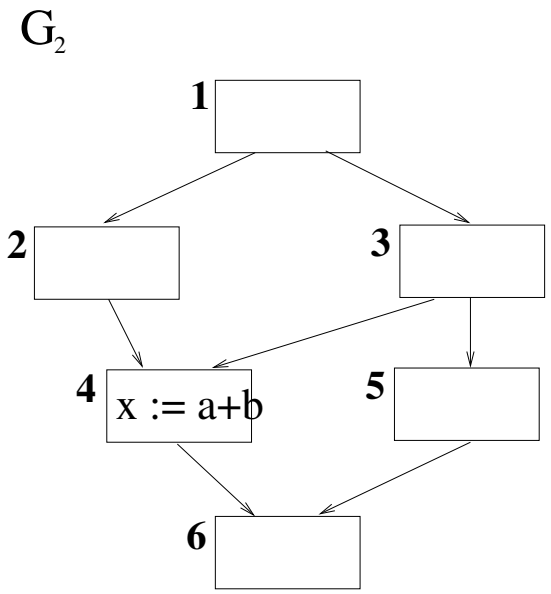
...by introducing the synthetic node $\mathbf{S}_{2,3}$, the critical edge from node **2** to node **3** is split which allows to eliminate the partially redundant computation of $a + b$ at node **3**.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Instructive

...optimizing the following two programs using the PRE algorithm of Morel&Renvoise:





Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 6 (1)

- 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)
- D. 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 6 (2)

- D. M. Dhamdhere. *E-path_pre: Partial Redundancy Elimi- nation Made Easy*. ACM SIGPLAN Notices 37(8):53-65, 2002.
- K.-H. Drechsler, M. 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.
- 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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 6 (3)

- 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.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 6 (4)

- 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.
- Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 13, Redundancy Elimination)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 6 (5)

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
6.1
6.2
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 7

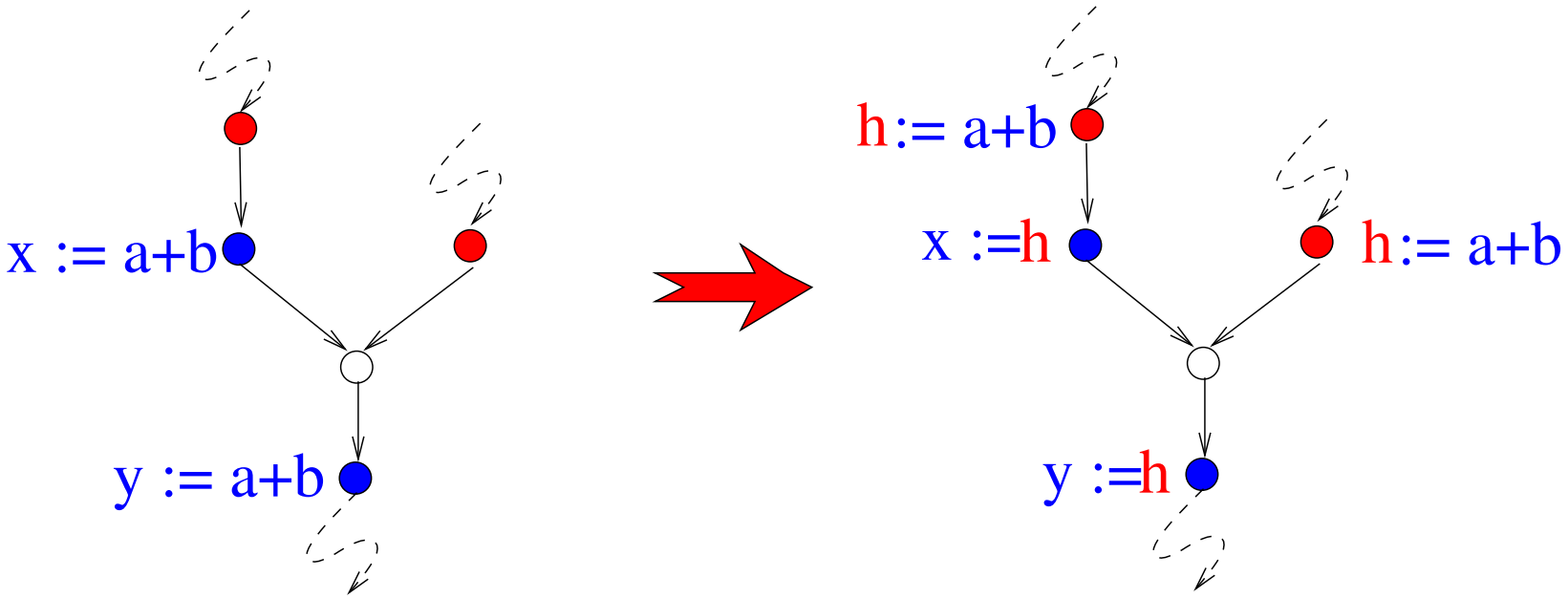
Busy Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Very Idea

...of Code Motion (CM) – often synonymously used with Partial Redundancy Elimination (PRE) – recalled:

...avoiding multiple (re-) computations of the same value!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 7.1

Preliminaries

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Notations

Given a flow gragh $G = (N, E, \mathbf{s}, \mathbf{e})$ let

- $pred(n) =_{df} \{m \mid (m, n) \in E\}$ 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 $(e_1, \ldots, e_k)$ with $dest(e_i) = source(e_{i+1})$ for all $1 \leq i < k$ denote a finite path.

  Note: Instead of edge sequences we also consider node sequences as paths, where reasonable.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Notations (Cont'd)

More specifically:

- $p = \langle e_1, \ldots, e_k \rangle$ 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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Convention

W/out losing generality we assume:

- Each node of a flow graph lies on a path from **s** to **e**

  Intuition: There are no unreachable parts within a flow graph.

...this is a typical and usual assumption for analysis and optimization!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

An additional CM specific Convention

W/out losing generality we focus in the following on flow graphs given

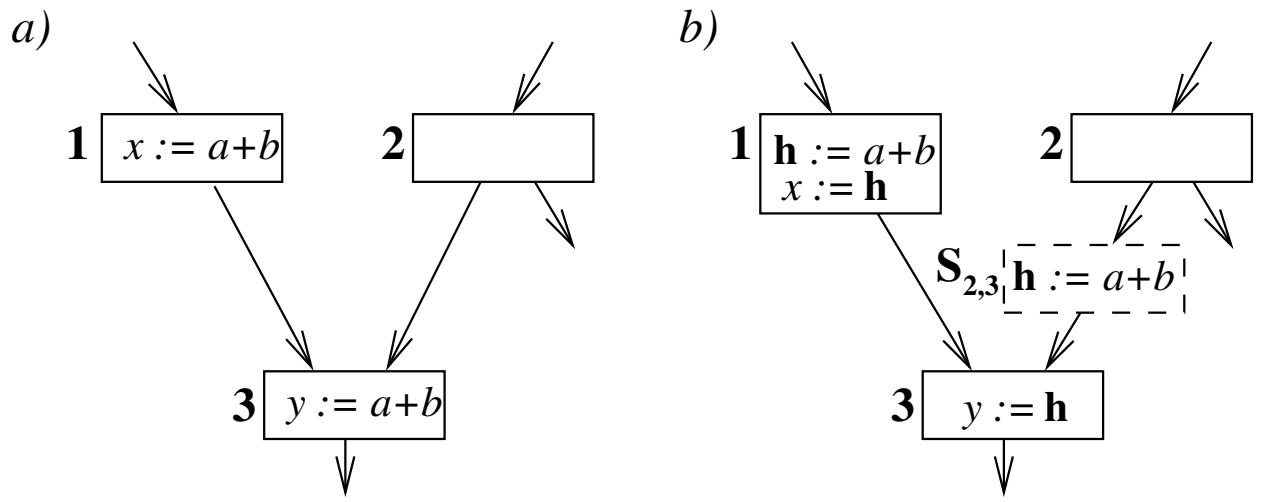
- as node labelled SI graphs
- where all edges leading to a join node are split by inserting a so-called synthetic node (i.e., not just critical edges are split)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Reminder: Critical Edges

An edge is called critical, if it connects a branching node with a join node.

Illustration: ...by introducing the synthetic node $\mathbf{S}_{2,3}$, the critical edge from node **2** to node **3** is split.



In the following we assume that also edges like the one from node **1** to node **3** are split by introducing a new node $\mathbf{S}_{1,3}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Background and Motivation

...underlying this convention:

- The CM process becomes simpler.

  $\rightsquigarrow$ computationally optimal results can be achieved by initializing temporaries always at node entries.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Note

Computationally optimal results can also be achieved, if only critical edges are split.

This, however, requires that a PRE algorithm is able to perform initializations both at node entries (N-initializations) and at node exits (X-Initializations).

This is not a problem at all. Agreeing, however, on the above assumption simplifies the presentation of the CM algorithm even more.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Work Plan

In the following we will define:

- The set of CM transformations
- The set of admissible CM transformations
- The set of computationally optimal CM transformations
- The *BCM* transformation as a specific computationally optimal CM transformation
- The *LCM* transformation as the one and only computatio- nally and lifetime optimal CM transformation (Chapter 8)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Generic Pattern of a CM Transformations

The generic 3-step (transformation) pattern for a term t:

- Introduce a fresh temporary h for t in G
- Insert at some nodes of G the assignment statement $h := t$
- Replace some of the original occurrences of t in G by h

Remark: t is often called a candidate expression.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Observation

Two predicates (defined on nodes)

- $Insert_{CM}$
- $Repl_{CM}$

suffice to specify a CM (resp. PRE) transformation completely

(the first step of declaring the temporary h is the same for each CM transformation and thus does not need to be considered explicitly).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Set of CM Transformations

...let $\mathcal{CM}_t$ denote the set of all CM transformations (for the candidate expression t).

In the following we will consider a fixed candidate expression t and thus drop the index t.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Observation

Obviously, some transformations in $\mathcal{CM}$ do not preserve the semantics and are thus not acceptable.

This leads us to the notion of admissible CM transformations.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Admissible CM Transformations

Let $CM \in \mathcal{CM}$.

CM is called admissible, if CM is safe and correct.

Intuitively:

- Safe: There is no path, on which by inserting an initialization a new value is computed.
- Correct: Whereever the temporary is used, it stores the "right" value, i.e., it stores the same value that a recomputation of t at the use site yields.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Formalising this

...requires two (local) predicates:

- $Comp_t(n)$: the candidate expression t is computed at n.
- $Transp_t(n)$: n is transparent for t, i.e., n does not modify any operand of t.

Note: In the following we will drop the index t.

Moreover, it is useful to introduce a third (local) predicate:

- $Comp_{CM}(n) =_{df} Insert_{CM}(n) \vee Comp(n) \wedge \neg Repl_{CM}(n)$: The candidate expression t is computed after the application of CM.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Extending Predicates from Nodes to Paths

Let p be a path and let p_i denote the ith node of p.

Then we define:

- $Predicate^{\forall}(p) \iff \forall 1 \leq i \leq \lambda_p.\ Predicate(p_i)$
- $Predicate^{\exists}(p) \iff \exists 1 \leq i \leq \lambda_p.\ Predicate(p_i)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Safety and Correctness

Definition (7.1.1, Safety and Correctness)

Let $n \in N$. Then:

1. $Safe(n) \iff_{df}$
   $\forall \langle n_1, \ldots, n_k \rangle \in \mathbf{P}[s, e]\ \forall i.\ (n_i = n) \Rightarrow$
   $i)\ \exists j < i.\ Comp(n_j) \wedge Transp^{\forall}(\langle n_j, \ldots, n_{i-1} \rangle)\ \vee$
   $ii)\ \exists j \geq i.\ Comp(n_j) \wedge Transp^{\forall}(\langle n_i, \ldots, n_{j-1} \rangle)$
2. Let $CM \in \mathcal{CM}$. Then:
   $Correct_{CM}(n) \iff_{df} \forall \langle n_1, \ldots, n_k \rangle \in \mathbf{P}[s, n]$
   $\exists i.\ Insert_{CM}(n_i) \wedge Transp^{\forall}(\langle n_i, \ldots, n_{k-1} \rangle)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Up-Safety and Down-Safety

Constraining the definition of safety to condition (i) resp. (ii) leads to the notions of

- up-safety (availability)
- down-safety (anticipability, very busyness)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Intuition

A computation of t at program point 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Up-Safety and Down-Safety

Definition (7.1.2, Up-Safety and Down-Safety)

1. $\forall\, n \in N.\ \mathit{U\text{-}Safe}(n) \iff_{df}$
   $\forall\, p \in \mathbf{P}[s, n]\ \exists\, i < \lambda_p.\ \mathit{Comp}\,(p_i) \wedge \mathit{Transp}^{\forall}(\, p[i, \lambda_p[\,)$
2. $\forall\, n \in N.\ \mathit{D\text{-}Safe}(n) \iff_{df}$
   $\forall\, p \in \mathbf{P}[n, e]\ \exists\, i \leq \lambda_p.\ \mathit{Comp}\,(p_i) \wedge \mathit{Transp}^{\forall}(\, p[1, i[\,)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Admissible CM-Transformations

This allows us to define:

Definition (7.1.3, 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. $Repl_{CM}(n) \Rightarrow Correct_{CM}(n)$

The set of all admissible CM-transformations is denoted by $\mathcal{CM}_{Adm}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

First Results

Lemma (7.1.4, Safety)

$$\forall\, n \in N.\ Safe(n) \iff D\text{-}Safe(n) \vee U\text{-}Safe(n)$$

Lemma (7.1.5, Correctness)

$$\forall\, CM \in \mathcal{CM}_{Adm}\ \forall\, n \in N.\ Correct_{CM}(n) \Rightarrow Safe(n)$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Computationally Better

Definition (7.1.6, Computationally Better)

A CM-transformation $CM \in \mathcal{CM}_{Adm}$ is computationally better as a CM-transformation $CM' \in \mathcal{CM}_{Adm}$ iff

$$\forall\ p \in \mathbf{P}[s, e].\ |\ \{i \mid Comp_{CM}(p_i)\}\ | \leq |\ \{i \mid Comp_{CM'}(p_i)\}\ |$$

Note: The relation "computationally better" is a quasi-order, i.e., a reflexive and transitive relation.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Computational Optimality

Definition (7.1.7, Comp. Optimal CM-Transf.)

An admissible CM-transformation $CM \in \mathcal{CM}_{Adm}$ is computa- tionally optimal iff CM is computationally better than any other admissible CM-transformation.

We denote the set of all computationally optimal CM-transformations by $\mathcal{CM}_{CmpOpt}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Properties of Relations – A Reminder

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 7.2

The *BCM*-Transformation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Conceptually

...CM can be considered a two-stage process consisting of:

1. Hoisting expressions
   ...hoisting expressions to "earlier" safe computation points
2. Eliminating totally redundant expressions
   ...elimination computations that became totally redundant by hoisting expressions

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Earliestness Principle

...induces an extreme placing (i.e., hoisting) strategy:

Placing computations as early as possible...

- Theorem (Computational Optimality)
  ...hoisting computations to their earliest safe computation points yields computationally optimal programs.

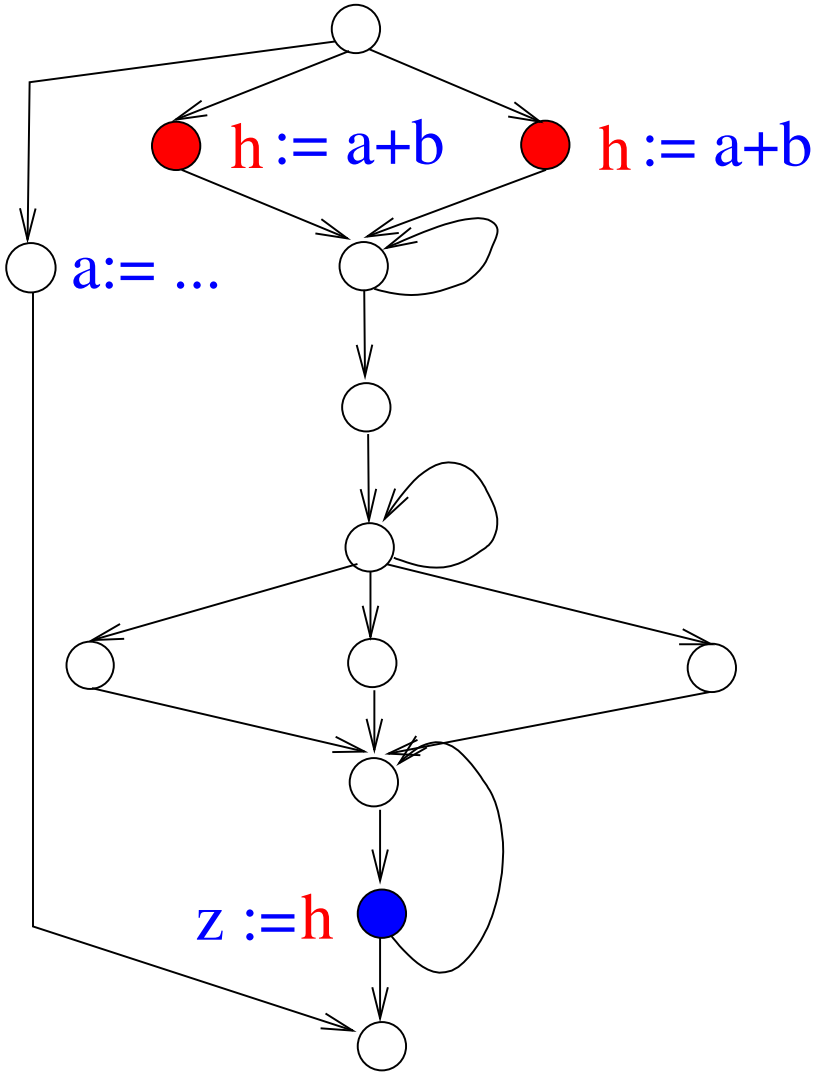
  $\rightsquigarrow$ ...known as the Busy Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Earliestness Principle

Placing computations as early as possible...

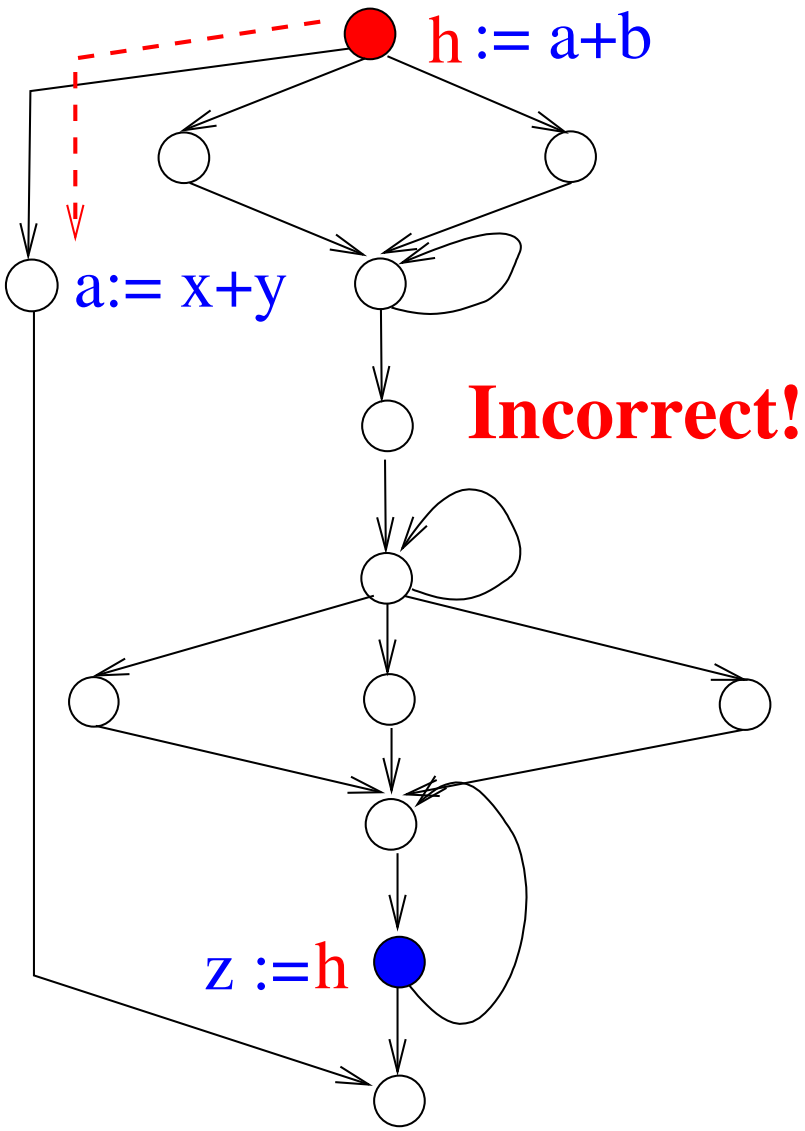
yields computationally optimal programs.



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Note

...earliest means indeed as early as possible, but not earlier!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

Busy Code Motion

Intuitively:

Place computations as early as possible in a program w/out violating safety and correctness!

Note: Following this principle computations are moved as far as possible in the opposite direction of the control flow

$\rightsquigarrow$ ...motivates the choice of the term busy.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Earliestness

Definition (7.2.1, Earliestness)

$\forall\, n \in N.\ Earliest(n) =_{df}$

$$Safe(n) \wedge \begin{cases} \textbf{true} & \text{if } n = \mathbf{s} \\ \bigvee_{m \in pred(n)} \neg Transp(m) \vee \neg Safe(m) & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The *BCM* Transformation

The *BCM* Transformation is defined by:

- $Insert_{BCM}(n) =_{df} Earliest(n)$
- $Repl_{BCM}(n) =_{df} Comp(n)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The *BCM* Theorem

Theorem (7.2.2, *BCM* Theorem)

The BCM-Transformation is computationally optimal, i.e., $BCM \in \mathcal{CM}_{CmpOpt}$.

The proof of the *BCM* Theorem 7.2.2 relies on the Earliestness Lemma 7.2.3 and the *BCM* Lemma 7.2.4.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Earliestness Lemma

Lemma (7.2.3, Earliestness Lemma)

Let $n \in N$. Then we have:

1. $Safe(n) \Rightarrow \forall\, p \in \mathbf{P}[s, n]\ \exists\, i \leq \lambda_p.$
   $Earliest(p_i) \wedge\ Transp^{\forall}(\, p[i, \lambda_p[\,)$
2. $Earliest(n) \iff$
   $D\text{-}Safe(n) \wedge \bigwedge_{m \in pred(n)} (\neg Transp(m) \vee \neg Safe(m))$
3. $Earliest(n) \iff$
   $Safe(n) \wedge\ \forall\, CM \in$
   $\mathcal{CM}_{Adm}.\ Correct_{CM}(n) \Rightarrow Insert_{CM}(n)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The *BCM* Lemma

Lemma (7.2.4, *BCM* Lemma)

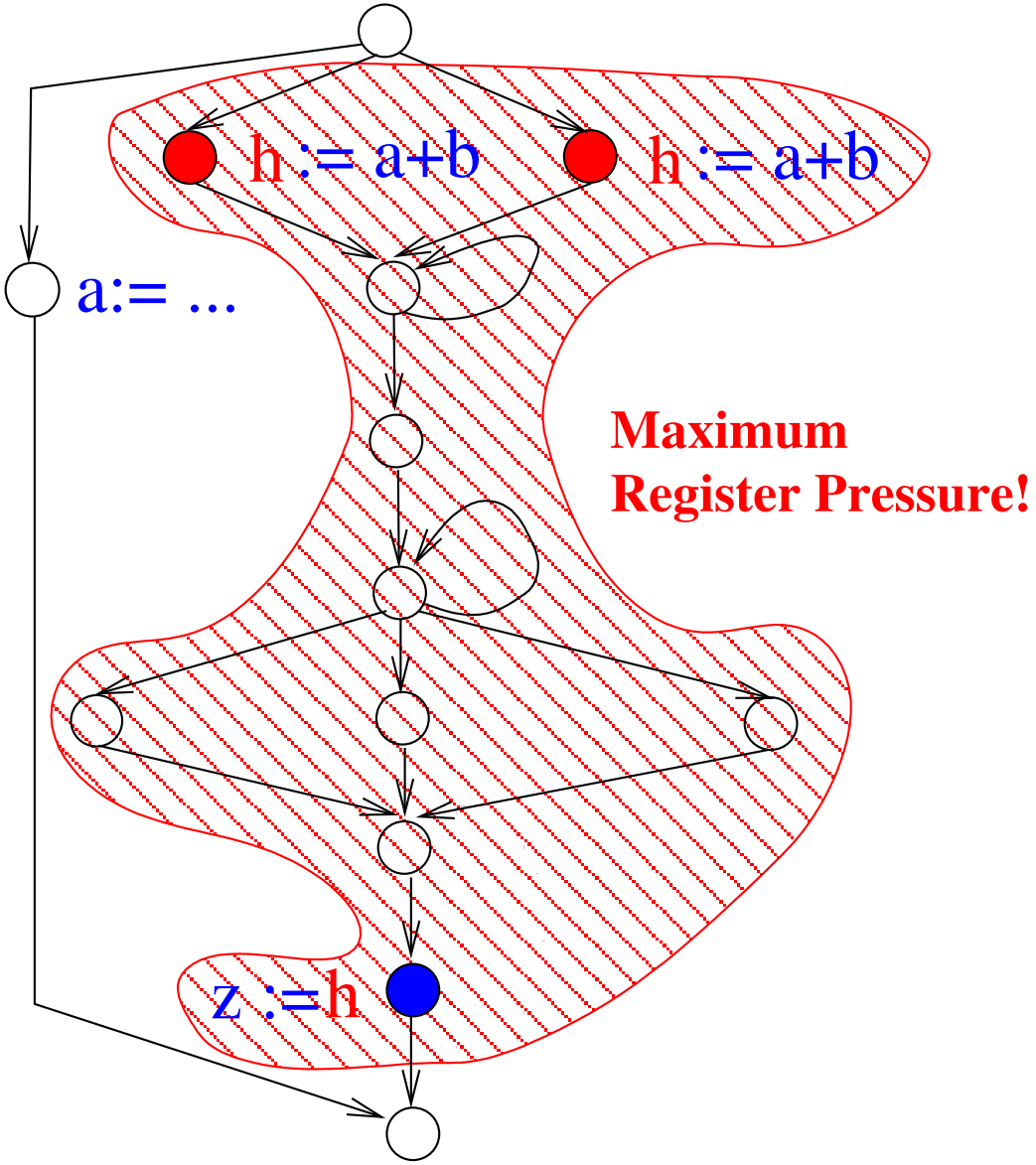
Let $p \in \mathbf{P}[s, e]$. *Then we have:*

1. $\forall\, i \leq \lambda_p.\ Insert_{BCM}(p_i) \iff$
   $\exists\, j \geq i.\ p[i,j] \in FU\text{-}LtRg(BCM)$
2. $\forall\, CM \in \mathcal{CM}_{Adm}\ \forall\, i, j \leq \lambda_p.\ p[i,j] \in LtRg(BCM) \Rightarrow$
   $Comp^{\exists}_{CM}(\, p[i,j]\,)$
3. $\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, i \leq \lambda_p.\ Comp_{CM}(p_i) \Rightarrow$
   $\exists\, j \leq i \leq l.\ p[j,l] \in FU\text{-}LtRg(BCM)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Result of the *BCM* Transformation

...computationally optimal, but maximum register pressure.



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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. *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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
7.1
7.2
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 8

Lazy Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Latestness Principle

...induces an extreme dual placing strategy:

Placing computations as late as possible...

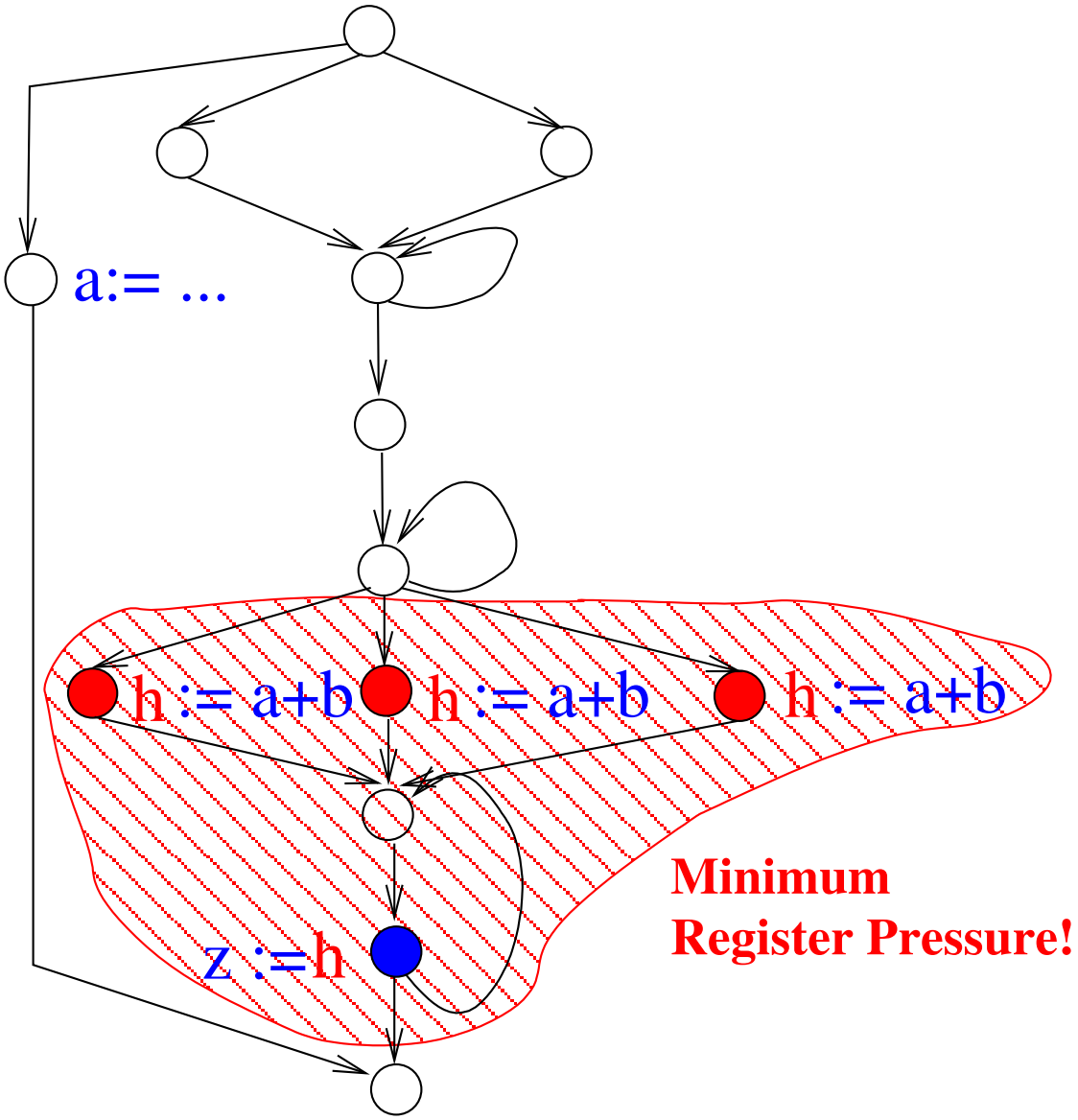
- Theorem (Lifetime Optimality)
  ...hoisting computations as little as possible, but as far as necessary (to achieve computational optimality), yields computationally optimal programs w/ minimum register pressure.

  $\rightsquigarrow$ ...known as the Lazy Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Result of the *LCM* Transformation

...computationally optimal w/ minimum register pressure!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Lazy Code Motion

Intuitively:
Place computations as late as possible in a program w/out violating safety, correctness and computational optimality!

Note: Following this principle computations are moved as little as possible in the opposite direction of the control flow

$\rightsquigarrow$ ...motivates the choice of the term lazy.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Work Plan

Next we will define:

- The set of lifetime optimal CM transformations
- The *LCM* transformation as the unique determined sole lifetime optimal CM transformation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Chapter 8.1

Preliminaries

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Central to Capture Register Pressure Formally

...is the notion of a (first-use) lifetime range.

Definition (8.1.1, Lifetime Ranges)

Let $CM \in \mathcal{CM}$.

- Lifetime range
  $LtRg(CM) =_{df}$
  $\{p \mid Insert_{CM}(p_1) \ \wedge Repl_{CM}(p_{\lambda_p}) \wedge \neg Insert^{\exists}_{CM}(\ p]1, \lambda_p]\ )\}$
- First-use lifetime range
  $FU\text{-}LtRg(CM) =_{df}$
  $\{p \in LtRg(CM) \mid \forall\, q \in LtRg(CM).\ (q \sqsubseteq p) \Rightarrow (q = p)\}$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

First Result

Lemma (8.1.2, 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* $j_2 < i_1$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Lifetime Better

Definition (8.1.3, 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Lifetime Optimality

Definition (8.1.4, Lifetime Optimal CM-Transf.)

A computationally 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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

We have

Lemma (8.1.5)

$\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, p \in LtRg(CM)\ \exists\, q \in LtRg(BCM).\ p \sqsubseteq q$

Intuitively:

- No computationally optimal CM-transformation places computations earlier as the *BCM* transformation
- The *BCM* transformation is that computationally optimal CM-transformation w/ maximum register pressure

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Uniqueness of Lifetime Optimal PRE

Obviously we have:

$$\mathcal{CM}_{LtOpt} \subseteq \mathcal{CM}_{CmpOpt} \subseteq \mathcal{CM}_{Adm} \subset \mathcal{CM}$$

Actually, we have even more:

Theorem (8.1.6, Uniqueness of Lifetime Optimal CM-Transformations)

$$|\mathcal{CM}_{LtOpt}| \leq 1$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Chapter 8.2

The *ALCM*-Transformation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Delayability

Definition (8.2.1, Delayability)

$\forall\, n \in N.\ Delayed(n) \iff_{df}$
$\quad \forall\, p \in \mathbf{P}[s, n]\ \exists\, i \leq \lambda_p.\ Earliest(p_i) \wedge \neg Comp^{\exists}(\, p[i, \lambda_p[\,)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Delayability Lemma

Lemma (8.2.2, Delayability Lemma)

1. $\forall\, n \in N.\ Delayed(n) \ \Rightarrow D\text{-}Safe(n)$
2. $\forall\, p \in \mathbf{P}[s, e]\ \forall i \leq \lambda_p.\ Delayed(p_i) \Rightarrow \exists\ j \leq i \leq l.$
   $p[j, l] \in FU\text{-}LtRg(BCM)$
3. $\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, n \in N.\ Comp_{CM}(n) \Rightarrow Delayed(n)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Latestness

Definition (8.2.3, Latestness)

$$\forall\, n \in N.\ Latest(n) =_{df} Delayed(n) \wedge (Comp(n) \vee \bigvee_{m \in succ(n)} \neg Delayed(m))$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Latestness Lemma

Lemma (8.2.4, Latestness Lemma)

1. $\forall\, p \in LtRg(BCM)\ \exists\, i \leq \lambda_p.\ Latest(p_i)$
2. $\forall\, p \in LtRg(BCM)\ \forall\, i \leq \lambda_p.\ Latest(p_i) \Rightarrow \neg Delayed^{\exists}(\, p]i, \lambda_p]\,)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The *ALCM* Transformation

The "Almost Lazy Code Motion" Transformation is defined by:

- $Insert_{ALCM}(n) =_{df} Latest(n)$
- $Repl_{ALCM}(n) =_{df} Comp(n)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Almost Lifetime Optimal

Definition (8.2.5, Almost Lifetime Optimal CM-Transformation)

A computationally 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 $\mathcal{CM}_{ALtOpt}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The *ALCM* Theorem

Theorem (8.2.6, *ALCM* Theorem)

The ALCM transformation is almost lifetime optimal, i.e., $ALCM \in \mathcal{CM}_{ALtOpt}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Chapter 8.3

Lazy Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Isolated Computations

Definition (8.3.1, *CM*-Isolation)

$\forall\, CM \in \mathcal{CM}\ \forall\, n \in N.\ Isolated_{CM}(n) \iff_{df}$
$\forall\, p \in \mathbf{P}[n, e]\ \forall\, 1 < i \leq \lambda_p.\ Repl_{CM}(p_i) \Rightarrow Insert^{\exists}_{CM}(\, p]1, i]\,)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Isolation Lemma

Lemma (8.3.2, Isolation Lemma)

1. $\forall\, CM \in \mathcal{CM}\ \forall\, n \in N.\ Isolated_{CM}(n) \iff$
$\forall\, p \in LtRg(CM).\ \langle n \rangle \sqsubseteq p \Rightarrow \lambda_p = 1$
2. $\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, n \in N.\ Latest(n) \Rightarrow$
$(Isolated_{CM}(n) \iff Isolated_{BCM}(n))$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 17

The *LCM* Transformation

The *LCM* Transformation is defined by:

- $Insert_{LCM}(n) =_{df} Latest(n) \wedge \neg Isolated_{BCM}(n)$
- $Repl_{LCM}(n) =_{df} Comp(n) \wedge \neg(Latest(n) \wedge Isolated_{BCM}(n))$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The *LCM* Theorem

Theorem (8.3.3, *LCM* Theorem)

The LCM transformation is lifetime optimal, i.e., $LCM \in \mathcal{CM}_{LtOpt}$.

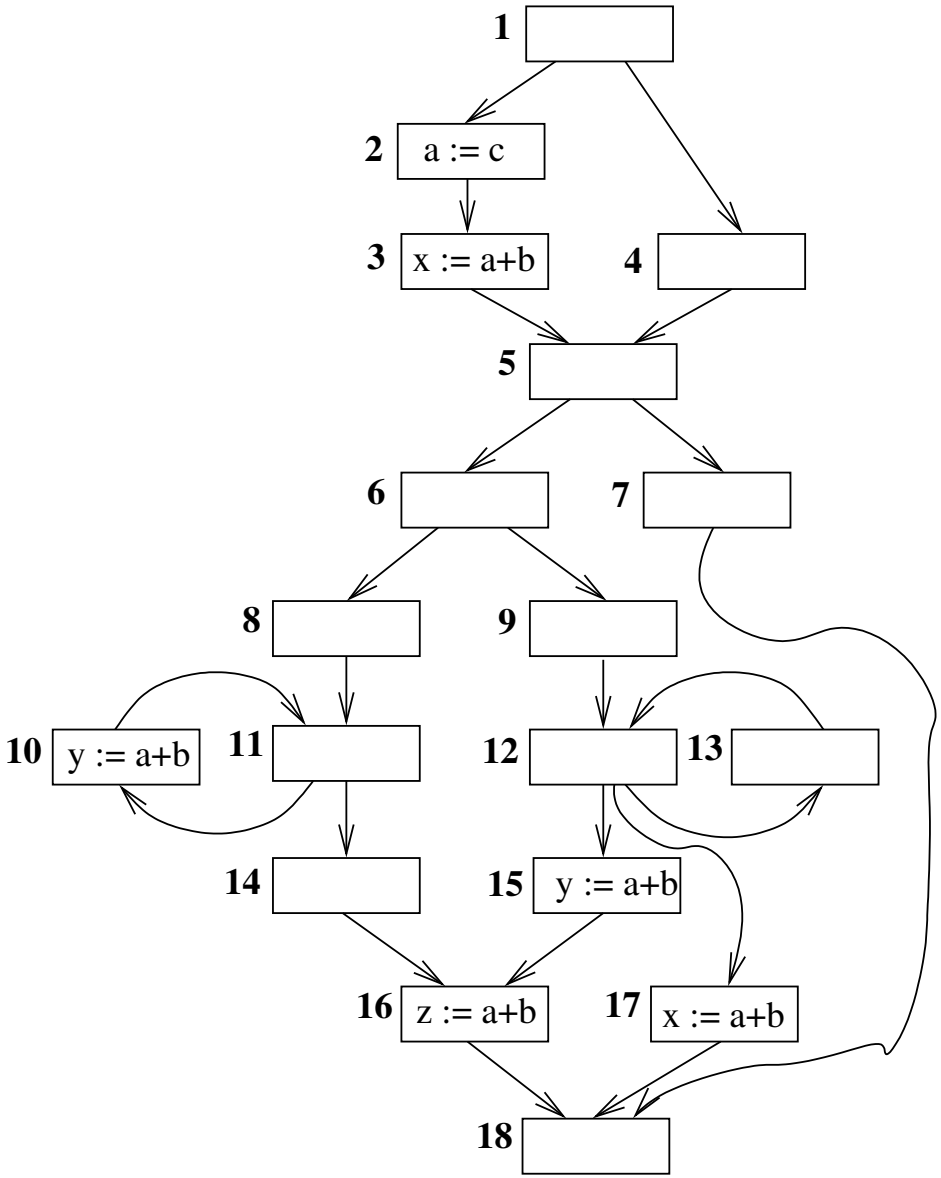
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Chapter 8.4

An Extended Example

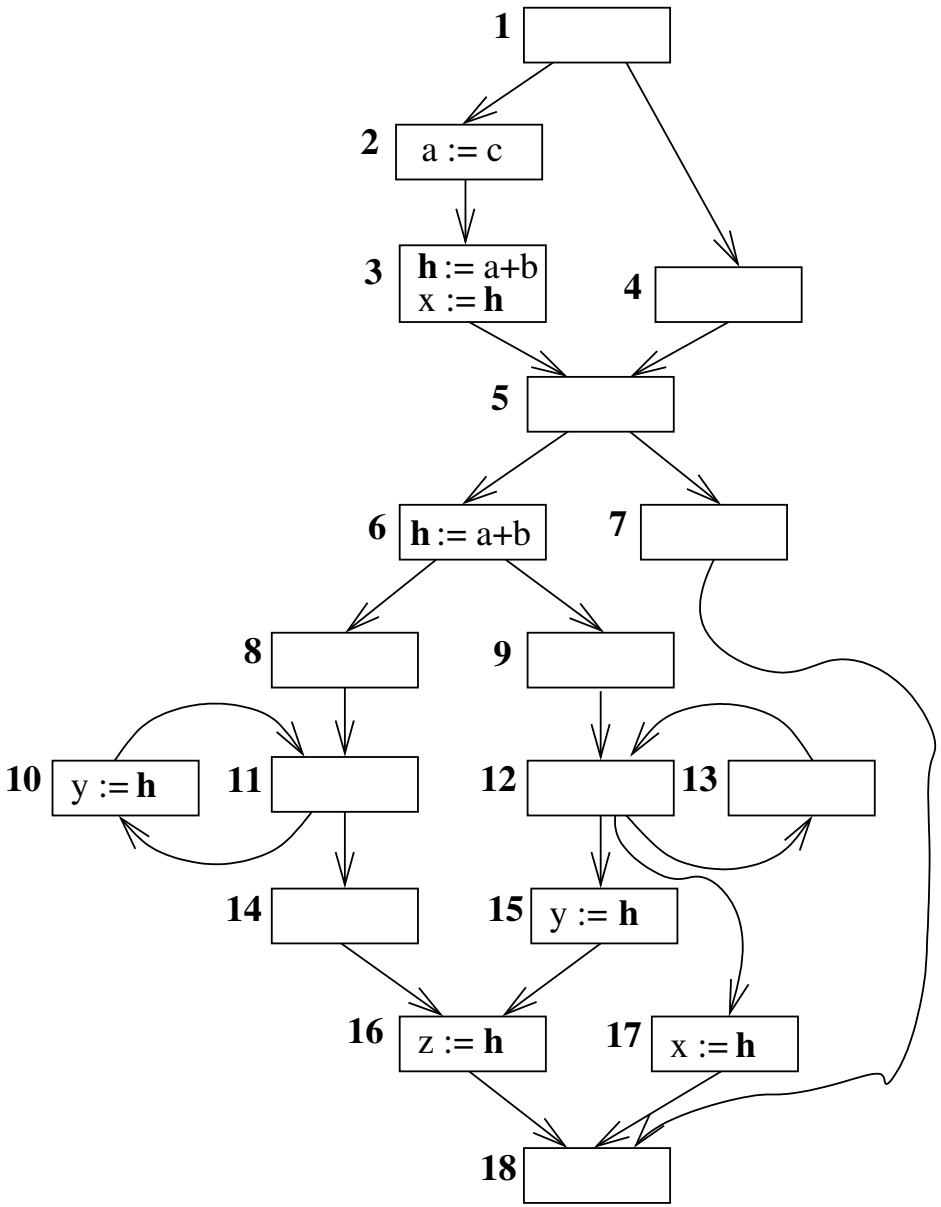
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Original Program



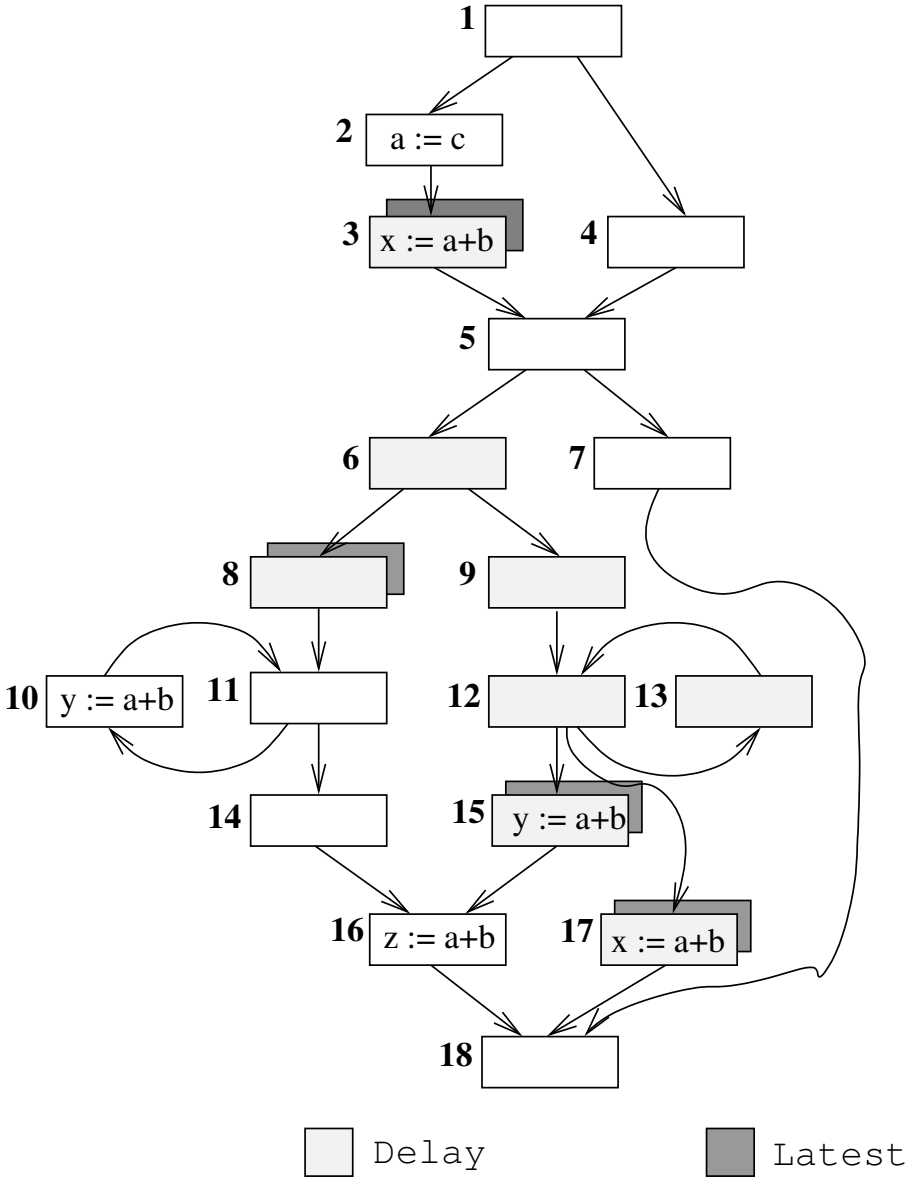
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Result of the *BCM* Transformation



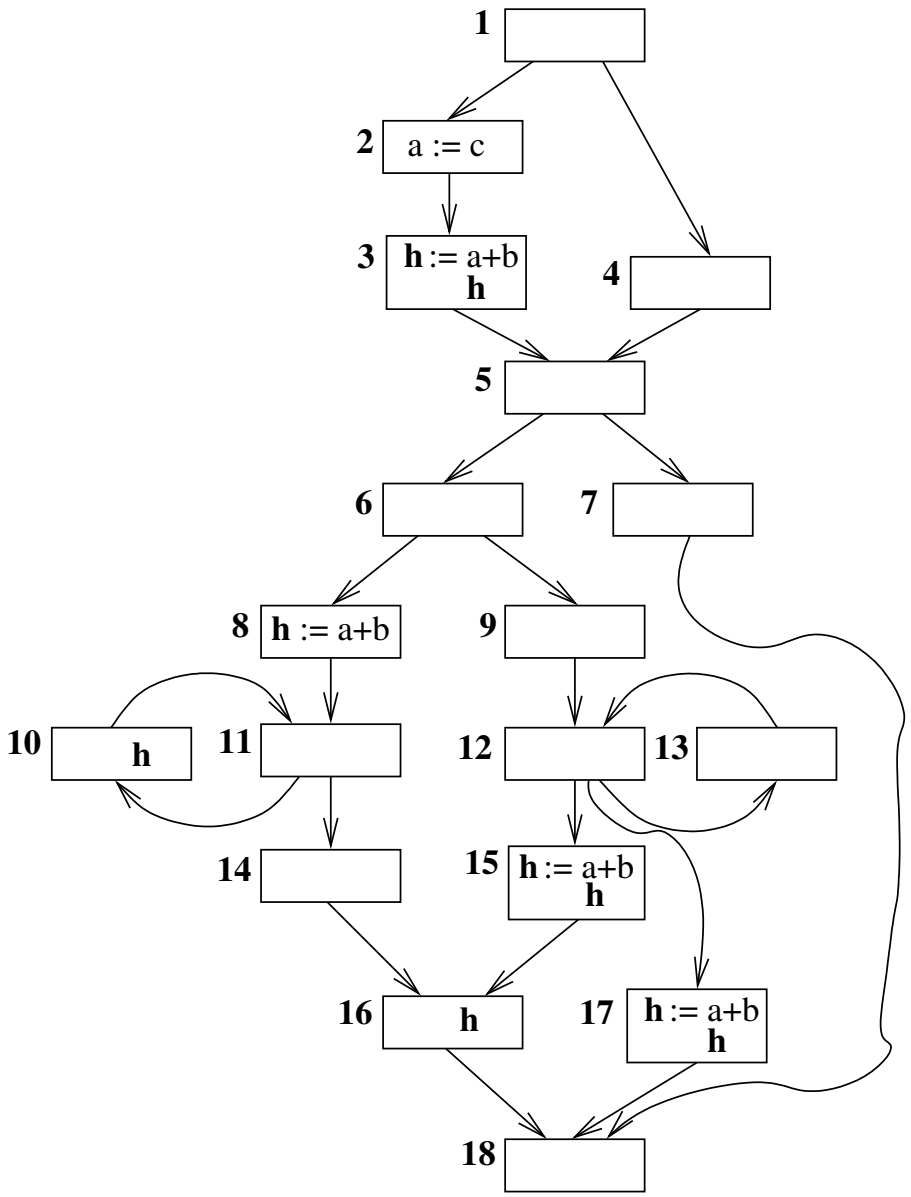
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Delayed and Latest Computation Points



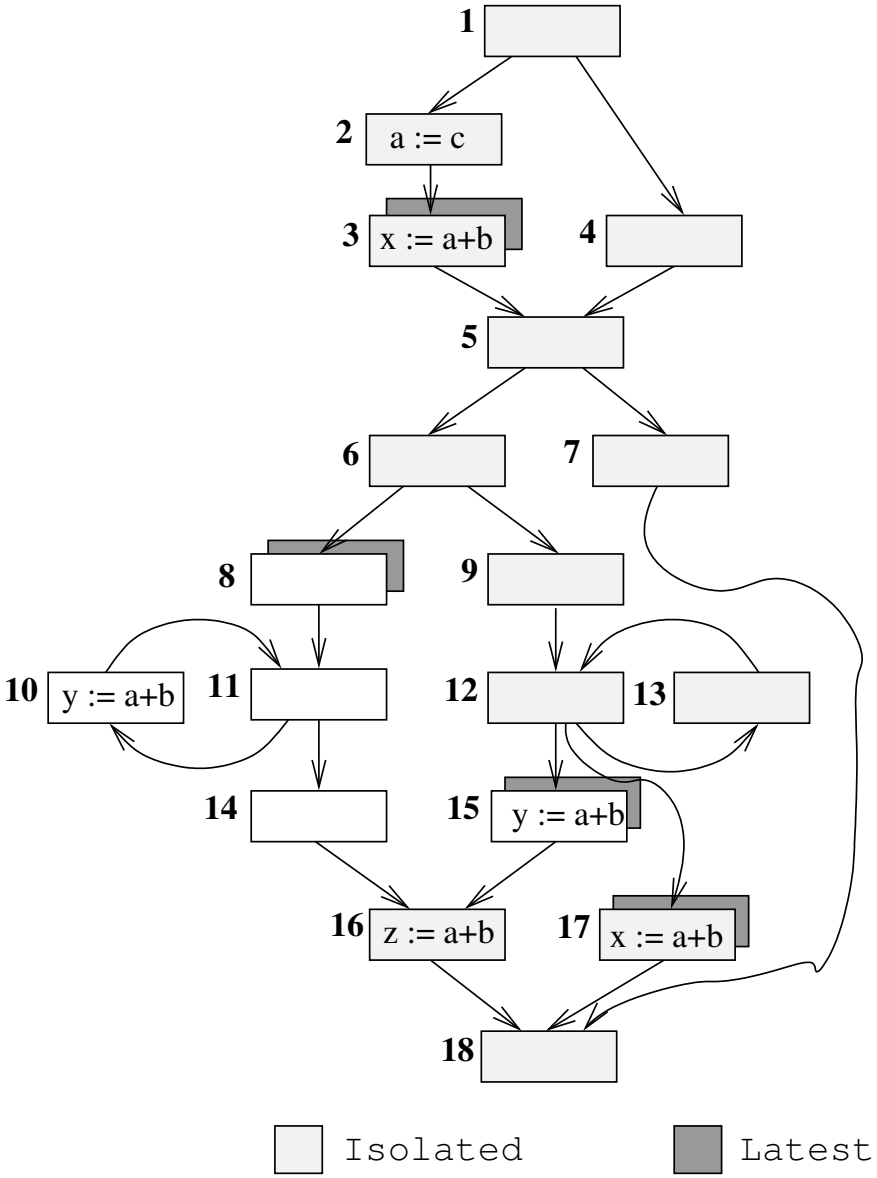
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Result of the *ALCM* Transformation



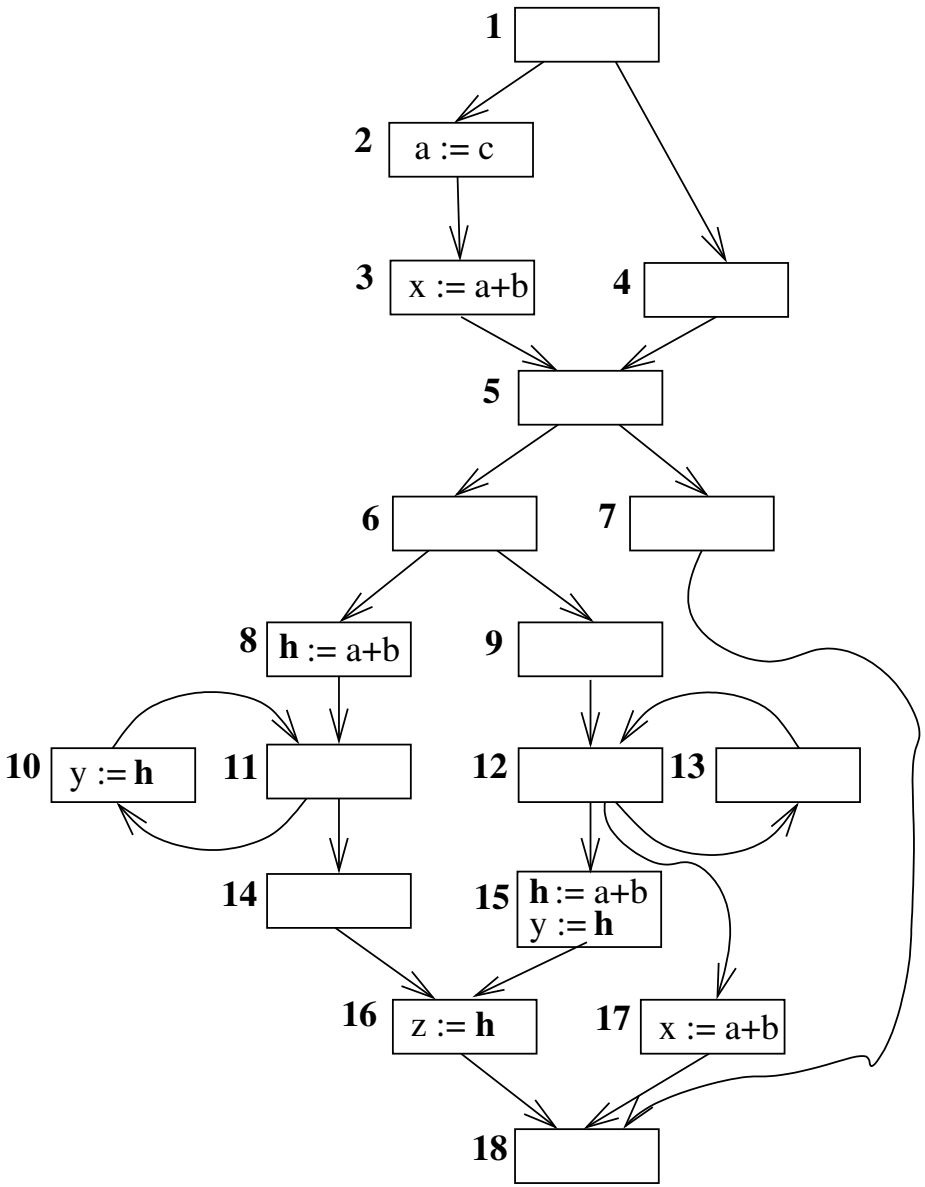
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Latest and Isolated Computation Points



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

The Result of the *LCM* Transformation



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

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)
- K.H. Drechsler, M. P. Stadel. *A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION*. ACM SIGPLAN Notices 28(5):29-38, 1993.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16

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.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 8(3)

Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 13.3, Partial-Redundancy Elimination – Lazy Code Motion)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
8.1
8.2
8.3
8.4
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15

Chapter 9

Implementing Busy and Lazy Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.1

Implementing *BCM* and *LCM* on SI-Graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.1.1

Preliminaries

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Busy and Lazy Code Motion

...for node-labelled SI-graphs:

- BCM_ι transformation
- LCM_ι transformation

Convention: For the following we assume that only critical edges are split. Therefore, BCM_ι and LCM_ι require insertions at both node entries and node exits (N-insertions and X-insertions).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Local Predicates for BCM_ι and LCM_ι

Local Predicates:

- $\text{COMP}_\iota(t)$: t is computed by ι.
- $\text{TRANSP}_\iota(t)$: No operand of t is modified by ι.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.1.2

Implementing BCM_ι

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_ι (1)

1. Analyses for Up-Safety and Down-Safety

The *MaxFP*-Equation System for Up-Safety:

$$\text{N-USAFE}_\iota = \begin{cases} \textbf{false} & \text{if } \iota = \textbf{s} \\ \prod_{\hat{\iota} \in pred(\iota)} \text{X-USAFE}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

$$\text{X-USAFE}_\iota = (\text{N-USAFE}_\iota + \text{COMP}_\iota) \cdot \text{TRANSP}_\iota$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_ι (2)

The *MaxFP*-Equation System for Down-Safety:

$$\text{N-DSAFE}_\iota = \text{COMP}_\iota + \text{X-DSAFE}_\iota \cdot \text{TRANSP}_\iota$$

$$\text{X-DSAFE}_\iota = \begin{cases} \textbf{false} & \text{if } \iota = \textbf{e} \\ \prod_{\hat{\iota} \in succ(\iota)} \text{N-DSAFE}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_ι (3)

2. The Transformation: Insertion&Replacement Points

Local Predicates:

- N-USAFE*, X-USAFE*, N-DSAFE*, X-DSAFE*:
  ...denote the greatest solutions of the equation systems for up-safety and down-safety of step 1.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_ι (4)

Computing Earliestness (no data flow analysis!):

$$\text{N-EARLIEST}_\iota \quad =_{df} \quad \text{N-DSAFE}^\star_\iota \cdot \prod_{\hat{\iota} \in pred(\iota)} \left(\overline{\text{X-USAFE}^\star_{\hat{\iota}} + \text{X-DSAFE}^\star_{\hat{\iota}}} \right)$$

$$\text{X-EARLIEST}_\iota \quad =_{df} \quad \text{X-DSAFE}^\star_\iota \cdot \overline{\text{TRANSP}_\iota}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_ι (5)

The BCM_ι Transformation:

$$\text{N-INSERT}_\iota^{\text{BCM}} =_{df} \text{N-EARLIEST}_\iota$$

$$\text{X-INSERT}_\iota^{\text{BCM}} =_{df} \text{X-EARLIEST}_\iota$$

$$\text{REPLACE}_\iota^{\text{BCM}} =_{df} \text{COMP}_\iota$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.1.3

Implementing LCM_ι

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_ι (1)

3. Analyses for Delayability and Isolation

The *MaxFP*-Equation System for Delayability:

$$\text{N-DELAYED}_\iota = \text{N-EARLIEST}_\iota + \begin{cases} \textbf{false} & \textit{if} \quad \iota = \textbf{s} \\ \prod_{\iota' \in pred(\iota)} \overline{\text{COMP}_{\iota'}} \cdot \text{X-DELAYED}_\iota & \textit{otherwise} \end{cases}$$

$$\text{X-DELAYED}_\iota = \text{X-EARLIEST}_\iota + \text{N-DELAYED}_\iota \cdot \overline{\text{COMP}_\iota}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_ι (2)

Computing Latestness (no data flow analysis!):

$$\text{N-LATEST}_\iota =_{df} \text{N-DELAYED}^\star_\iota \cdot \text{COMP}_\iota$$

$$\text{X-LATEST}_\iota =_{df} \text{X-DELAYED}^\star_\iota \cdot (\,\text{COMP}_\iota + \sum_{\iota' \in succ(\iota)} \overline{\text{N-DELAYED}^\star_{\iota'}}\,)$$

where

- $\text{N-DELAYED}^\star$, $\text{X-DELAYED}^\star$: ...denote the greatest solutions of the equation system for delayability.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_ι (3)

The $ALCM_\iota$ Transformation:

$$\begin{aligned} \text{N-INSERT}_\iota^{\text{ALCM}} &=_{df} \text{N-LATEST}_\iota \\ \text{X-INSERT}_\iota^{\text{ALCM}} &=_{df} \text{X-LATEST}_\iota \end{aligned}$$

$$\text{REPLACE}_\iota^{\text{ALCM}} =_{df} \text{COMP}_\iota$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_ι (4)

The *MaxFP*-Equation System for Isolation:

$$\text{N-ISOLATED}_\iota = \text{X-EARLIEST}_\iota + \text{X-ISOLATED}_\iota$$

$$\text{X-ISOLATED}_\iota = \prod_{\iota' \in succ(\iota)} \text{N-EARLIEST}_{\iota'} + \overline{\text{COMP}_{\iota'}} \cdot \text{N-ISOLATED}_{\iota'}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_ι (5)

4. The Transformation: Insertion&Replacement Points

Local Predicates:

- N-ISOLATED*, X-ISOLATED*: ...denote the greatest solutions of the equation system for isolation of step 3.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_ι (6)

The LCM_ι Transformation:

$$\text{N-INSERT}_\iota^{\text{LCM}} =_{df} \text{N-LATEST}_\iota \cdot \overline{\text{N-ISOLATED}_\iota^\star}$$
$$\text{X-INSERT}_\iota^{\text{LCM}} =_{df} \text{X-LATEST}_\iota \cdot \overline{\text{X-ISOLATED}_\iota^\star}$$

$$\text{REPLACE}_\iota^{\text{LCM}} =_{df} \text{COMP}_\iota \cdot \overline{\text{N-LATEST}_\iota \cdot \text{N-ISOLATED}_\iota^\star}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.2

Implementing *BCM* and *LCM* on BB-Graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.2.1

Preliminaries

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing Busy and Lazy Code Motion

...for node-labelled BB-graphs:

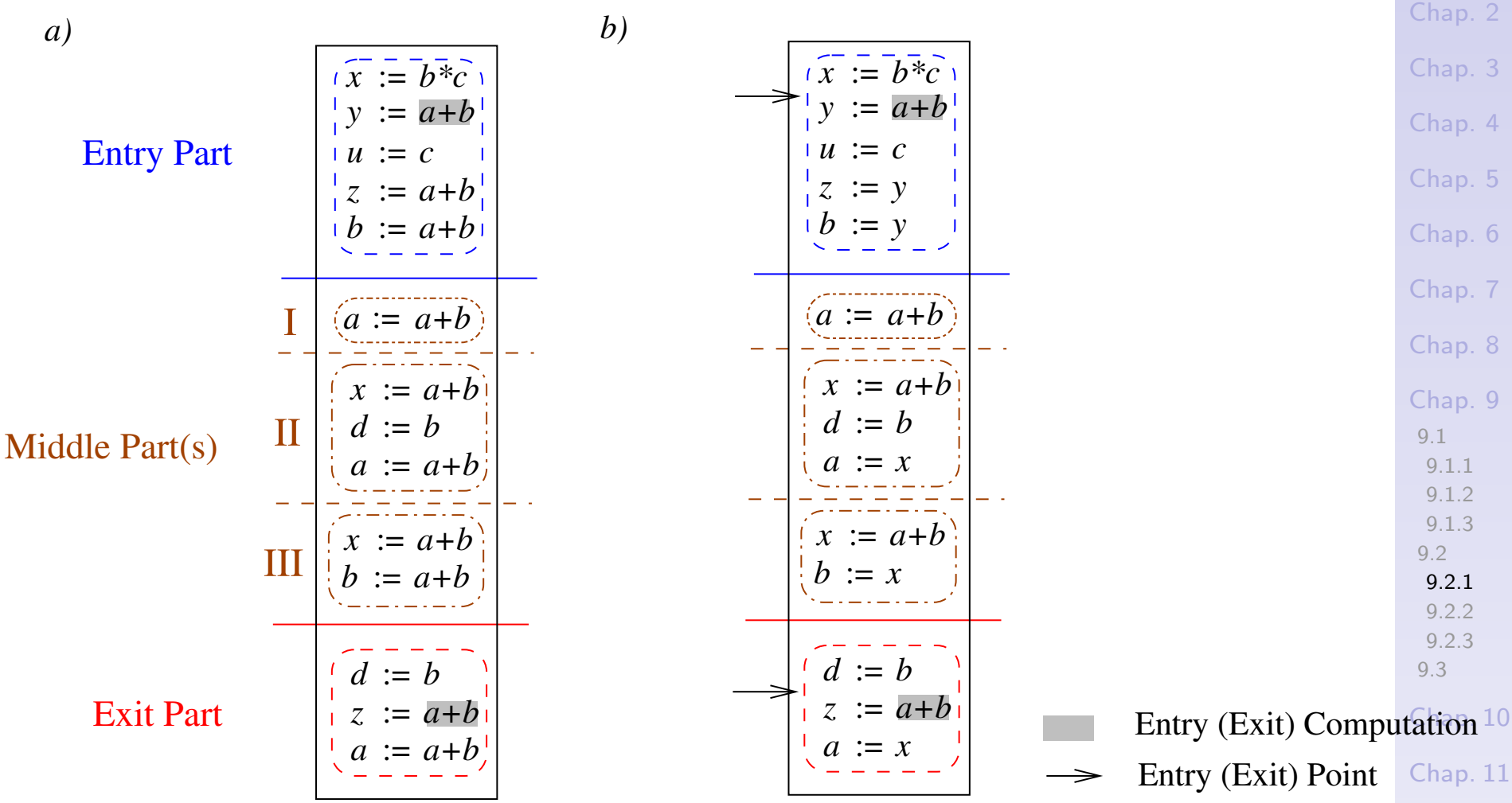
- BCM_β Transformation
- LCM_β Transformation

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-in- sertions), and that (2) all redundancies within a basic block have been removed by a preprocess.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Conceptual Splitting of a Basic Block

...into entry, middle, and exit part.



Original Basic Block

After Local Redundancy Elimination

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

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:

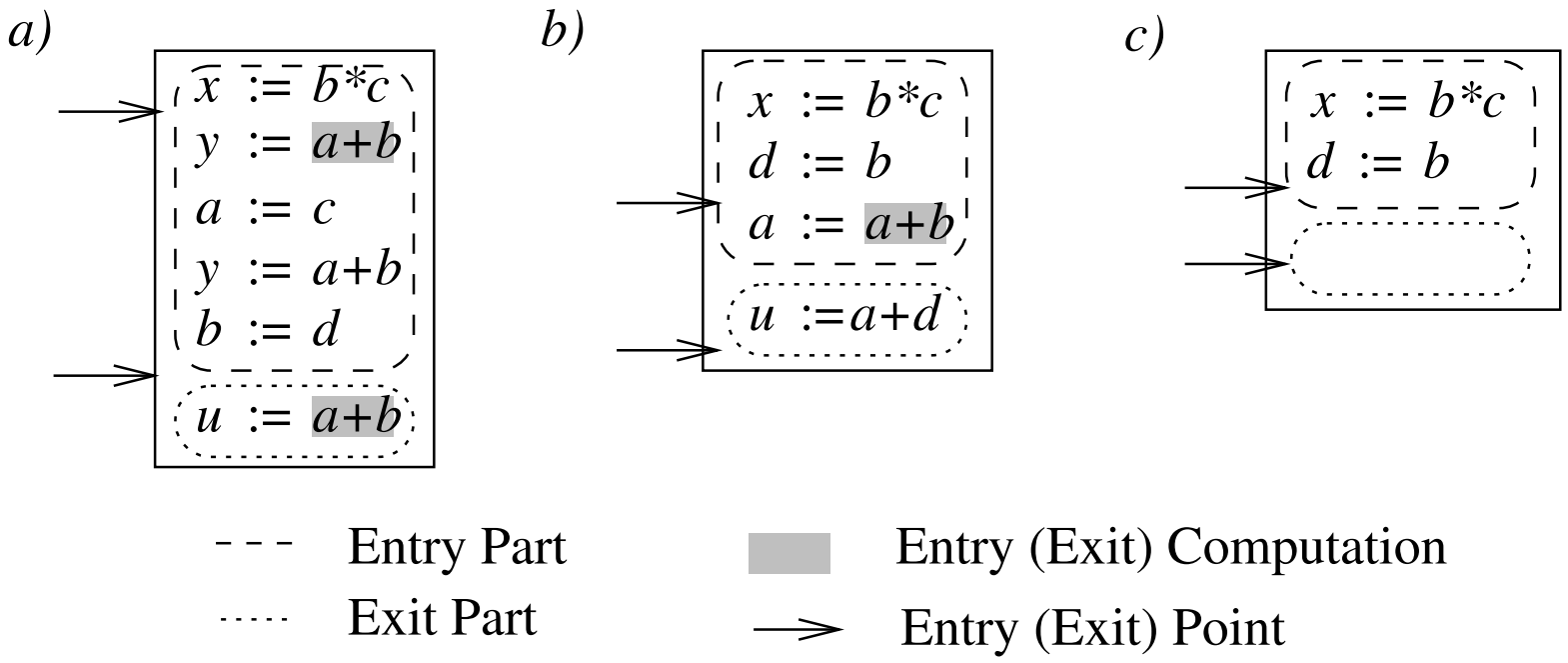
Given a computation t, a basic block **n** can be divided into two parts:

- 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 itself (this is illustrated in the following figure).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Illustrating Entry & Exit Part of a Basic Block



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

The General Pattern of CM on BB-Graphs

1. Introducing temporay
   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 art 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}

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.2.2

Implementing BCM_β

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_β (1)

1. Analyses for Up-Safety and Down-Safety

The *MaxFP*-Equation System for Up-Safety:

$$\text{BB-N-USAFE}_\beta = \begin{cases} \textbf{false} & \text{if } \beta = \textbf{s} \\ \prod_{\hat{\beta} \in \mathit{pred}(\beta)} (\text{BB-XCOMP}_{\hat{\beta}} + \text{BB-X-USAFE}_{\hat{\beta}}) & \text{otherwise} \end{cases}$$

$$\text{BB-X-USAFE}_\beta = (\text{BB-N-USAFE}_\beta + \text{BB-NCOMP}_\beta) \cdot \text{BB-TRANSP}_\beta$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_β (2)

The *MaxFP*-Equation System for Down-Safety:

$$\text{BB-N-DSAFE}_\beta = \text{BB-NCOMP}_\beta + \text{BB-X-DSAFE}_\beta \cdot \text{BB-TRANSP}_\beta$$

$$\text{BB-X-DSAFE}_\beta = \text{BB-XCOMP}_\beta + \begin{cases} \textbf{false} & \text{if } \beta = \textbf{e} \\ \prod\limits_{\hat{\beta} \in succ(\beta)} \text{BB-N-DSAFE}_{\hat{\beta}} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_β (3)

2. The Transformation: Insertion&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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_β (4)

Computing Earliestness (no data flow analysis!):

$$\text{N-EARLIEST}_\beta =_{df} \text{BB-N-DSAFE}^\star_\beta \cdot \prod_{\hat{\beta} \in pred(\beta)} (\overline{\text{BB-X-USAFE}^\star_{\hat{\beta}} + \text{BB-X-DSAFE}^\star_{\hat{\beta}}})$$

$$\text{X-EARLIEST}_\beta =_{df} \text{BB-X-DSAFE}^\star_\beta \cdot \overline{\text{BB-TRANSP}_\beta}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing BCM_β (5)

The BCM_β Transformation:

$$\text{N-INSERT}_\beta^{\text{BCM}} =_{df} \text{N-EARLIEST}_\beta$$
$$\text{X-INSERT}_\beta^{\text{BCM}} =_{df} \text{X-EARLIEST}_\beta$$

$$\text{N-REPLACE}_\beta^{\text{BCM}} =_{df} \text{BB-NCOMP}_\beta$$
$$\text{X-REPLACE}_\beta^{\text{BCM}} =_{df} \text{BB-XCOMP}_\beta$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.2.3

Implementing LCM_β

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_β (1)

3. Analyses for Delayability and Isolation

The *MaxFP*-Equation System for Delayability:

$$\text{N-DELAYED}_\beta = \text{N-EARLIEST}_\beta + \begin{cases} \textbf{false} & \textit{if} \quad \beta = \textbf{s} \\ \prod_{\hat{\beta} \in \textit{pred}(\beta)} \overline{\text{BB-XCOMP}_{\hat{\beta}}} \cdot \text{X-DELAYED}_{\hat{\beta}} & \textit{otherwise} \end{cases}$$

$$\text{X-DELAYED}_\beta = \text{X-EARLIEST}_\beta + \text{N-DELAYED}_\beta \cdot \overline{\text{BB-NCOMP}_\beta}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_β (2)

Computing Latestness (no data flow analysis!):

$$\text{N-LATEST}_\beta =_{df} \text{N-DELAYED}^\star_\beta \cdot \text{BB-NCOMP}_\beta$$

$$\text{X-LATEST}_\beta =_{df} \text{X-DELAYED}^\star_\beta \cdot (\text{BB-XCOMP}_\beta + \sum_{\hat\beta \in succ(\beta)} \overline{\text{N-DELAYED}^\star_{\hat\beta}})$$

where

- N-DELAYED*, X-DELAYED*: ...denote the greatest solutions of the equation system for delayability.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_β (3)

The $ALCM_\beta$ Transformation:

$$\text{N-INSERT}_\beta^{\text{ALCM}} =_{df} \text{N-LATEST}_\beta$$
$$\text{X-INSERT}_\beta^{\text{ALCM}} =_{df} \text{X-LATEST}_\beta$$

$$\text{N-REPLACE}_\beta^{\text{ALCM}} =_{df} \text{BB-NCOMP}_\beta$$
$$\text{X-REPLACE}_\beta^{\text{ALCM}} =_{df} \text{BB-XCOMP}_\beta$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_β (4)

The *MaxFP*-Equation System for Isolation:

$$\text{N-ISOLATED}_\beta = \text{X-EARLIEST}_\beta + \text{X-ISOLATED}_\beta$$

$$\text{X-ISOLATED}_\beta = \prod_{\hat{\beta} \in succ(\beta)} \text{N-EARLIEST}_{\hat{\beta}} + \overline{\text{BB-NCOMP}_{\hat{\beta}}} \cdot \text{N-ISOLATED}_{\hat{\beta}}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_β (5)

4. The Transformation: Insertion&Replacement Points

Local Predicates:

- N-ISOLATED*, X-ISOLATED*: ...denote the greatest solutions of the equation system for isolation of step 3.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Implementing LCM_β (6)

The LCM_β Transformation:

$$\text{N-INSERT}_\beta^{\text{LCM}} =_{df} \text{N-LATEST}_\beta \cdot \overline{\text{N-ISOLATED}_\beta^\star}$$

$$\text{X-INSERT}_\beta^{\text{LCM}} =_{df} \text{X-LATEST}_\beta \cdot \overline{\text{X-ISOLATED}_\beta^\star}$$

$$\text{N-REPLACE}_\beta^{\text{LCM}} =_{df} \text{BB-NCOMP}_\beta \cdot \overline{\text{N-LATEST}_\beta \cdot \text{N-ISOLATED}_\beta^\star}$$

$$\text{X-REPLACE}_\beta^{\text{LCM}} =_{df} \text{BB-XCOMP}_\beta \cdot \overline{\text{X-LATEST}_\beta \cdot \text{X-ISOLATED}_\beta^\star}$$

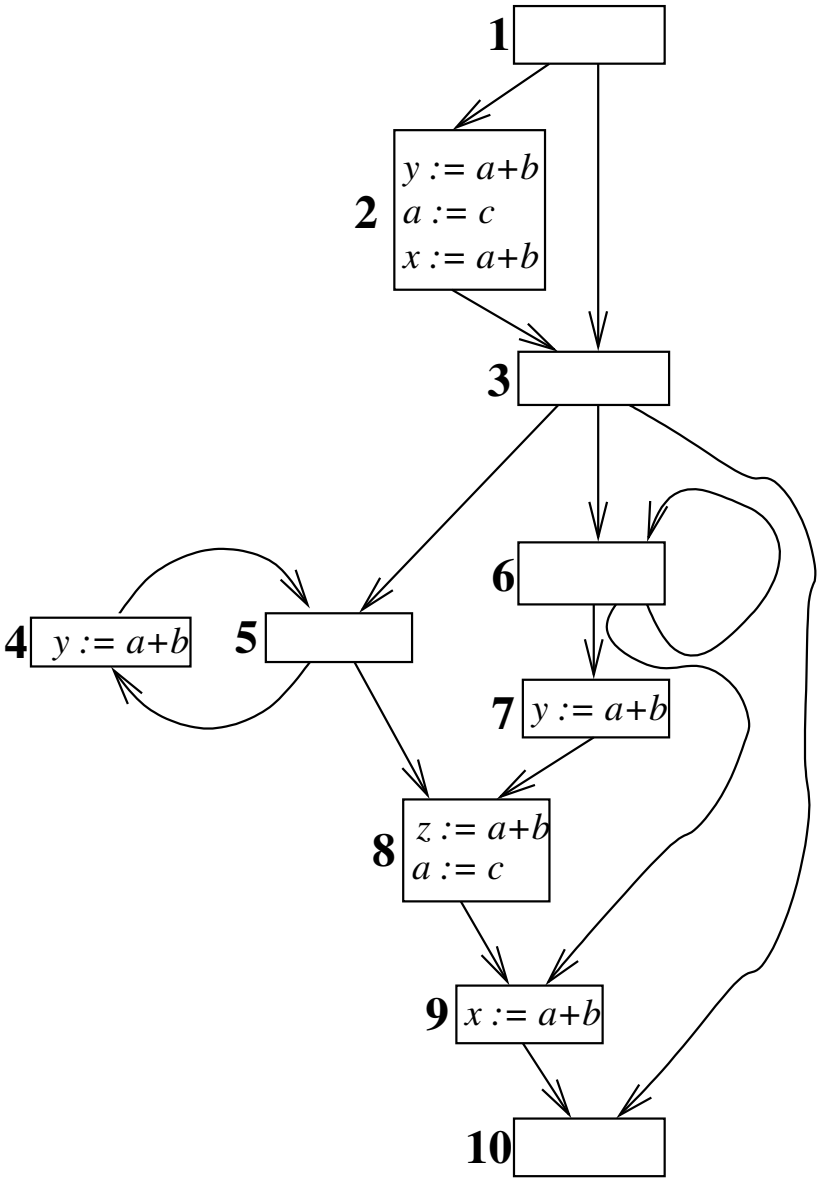
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 9.3

An Extended Example

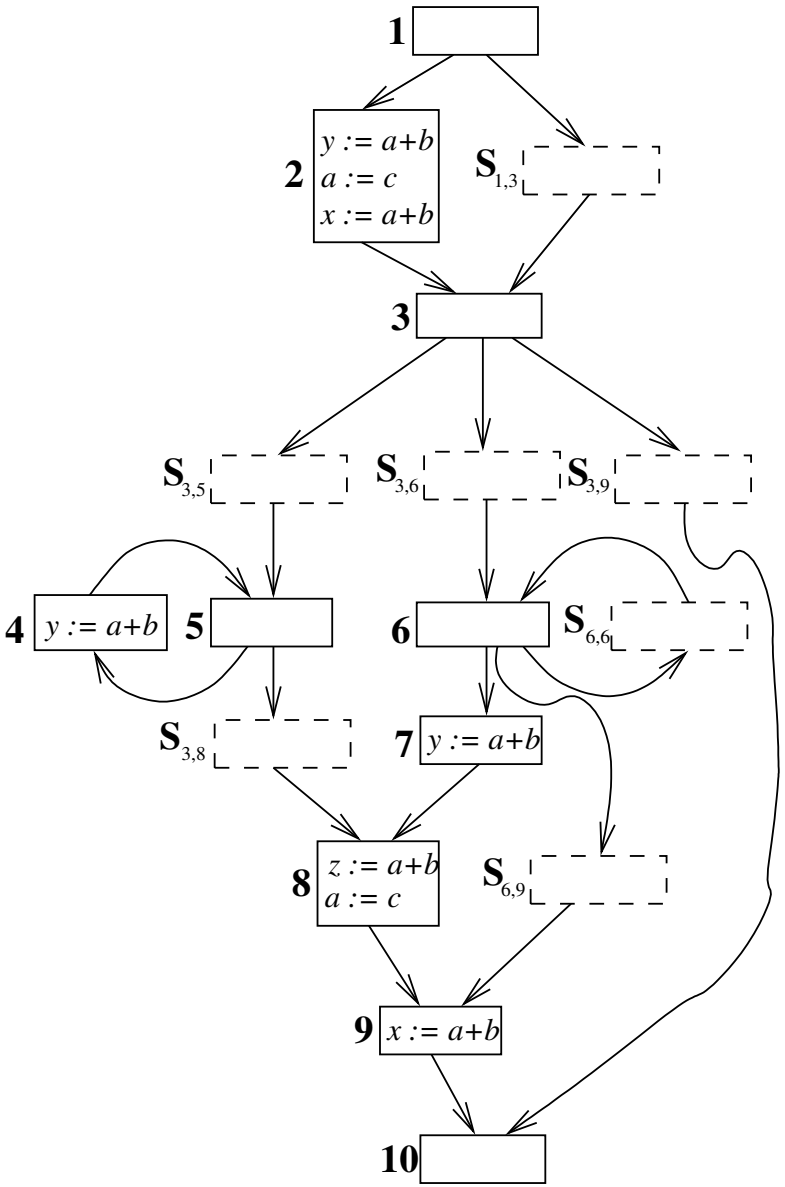
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

The Original Program



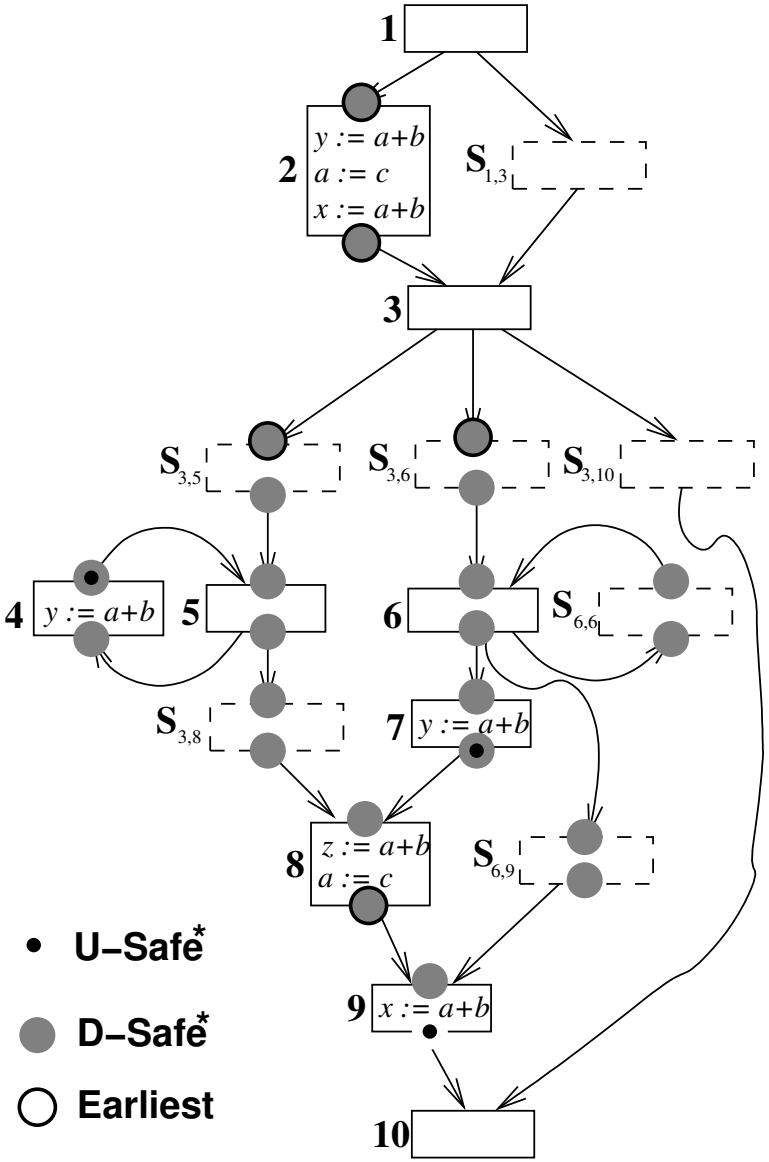
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

After the Splitting of Critical Edges



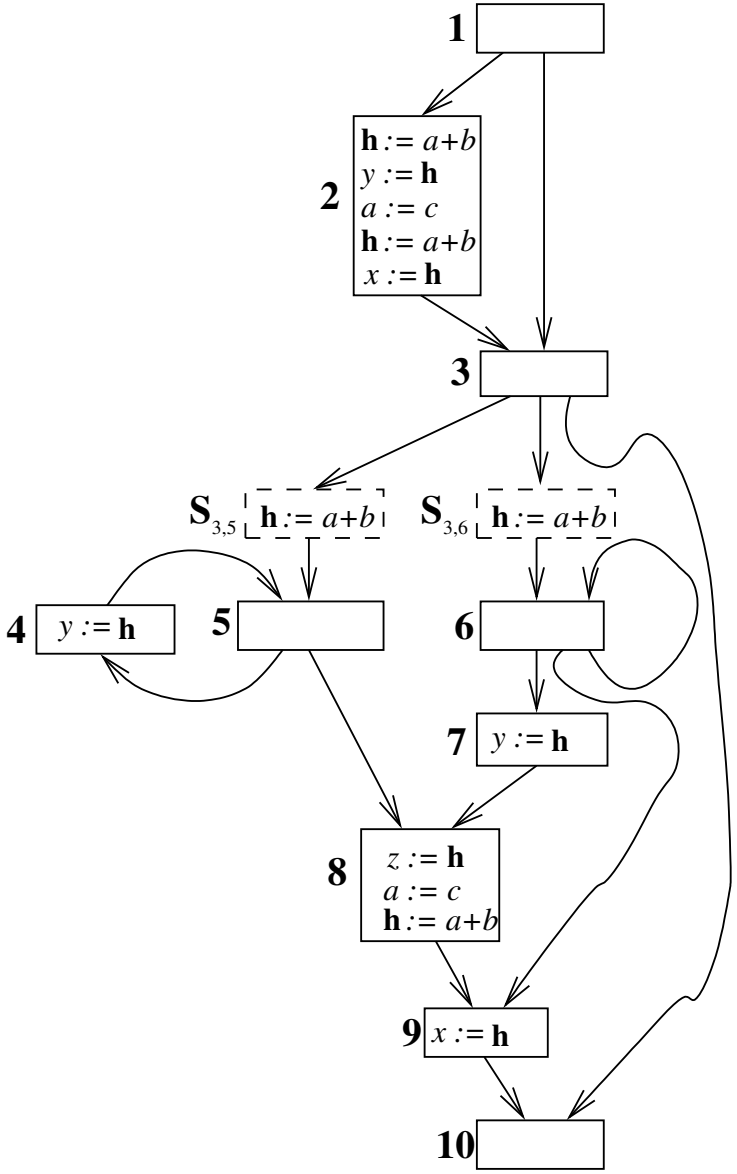
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Up/Down-Safe, Earliest Computation Points



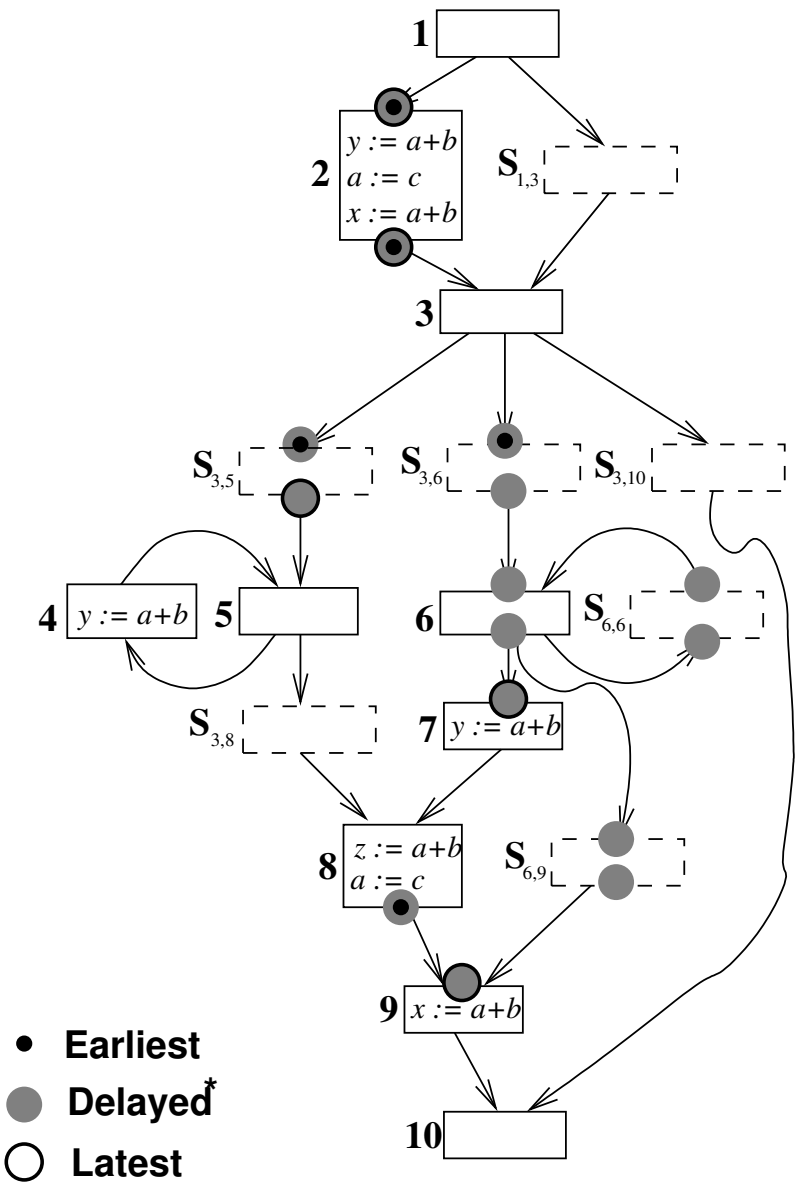
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

The Result of the BCM_β Transformation



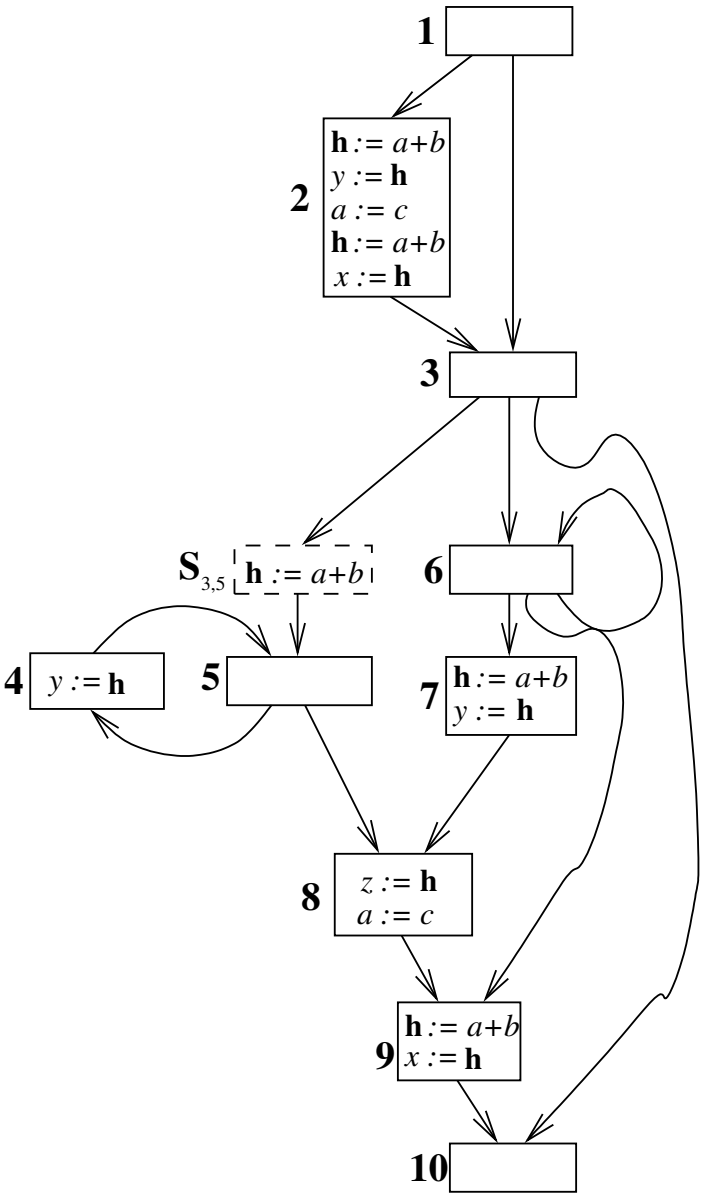
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Delayable and Latest Computation Points



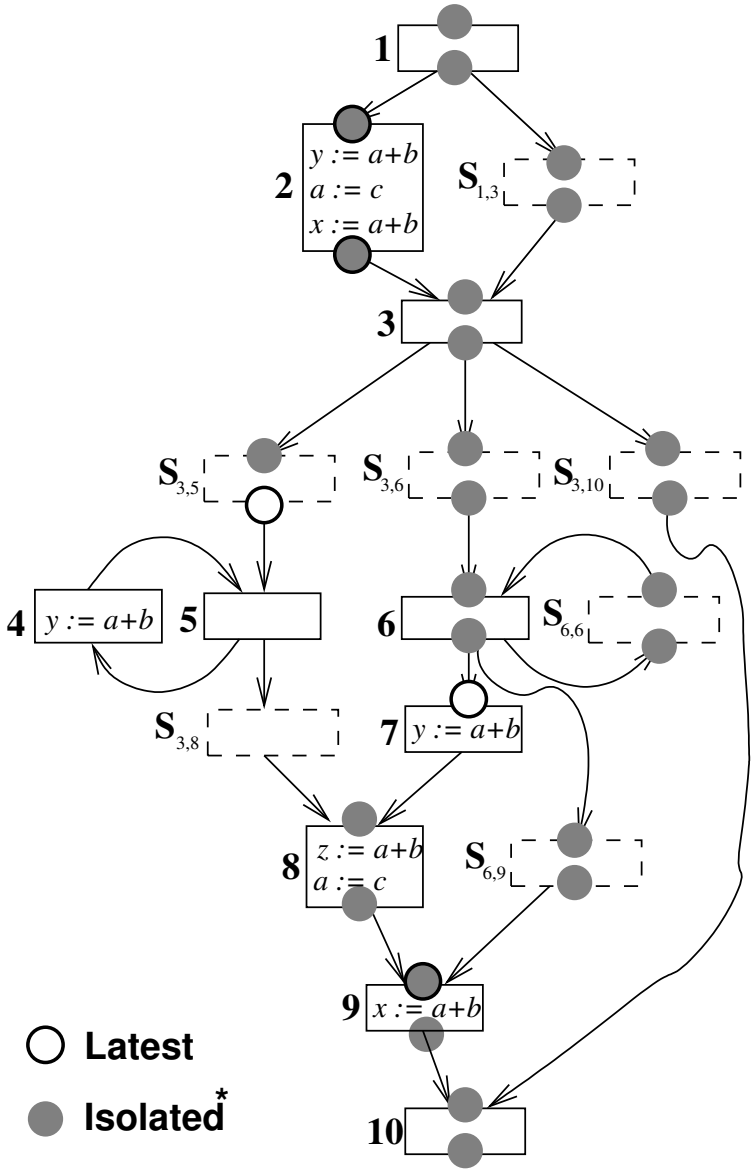
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

The Result of the $ALCM_\beta$-Transformation



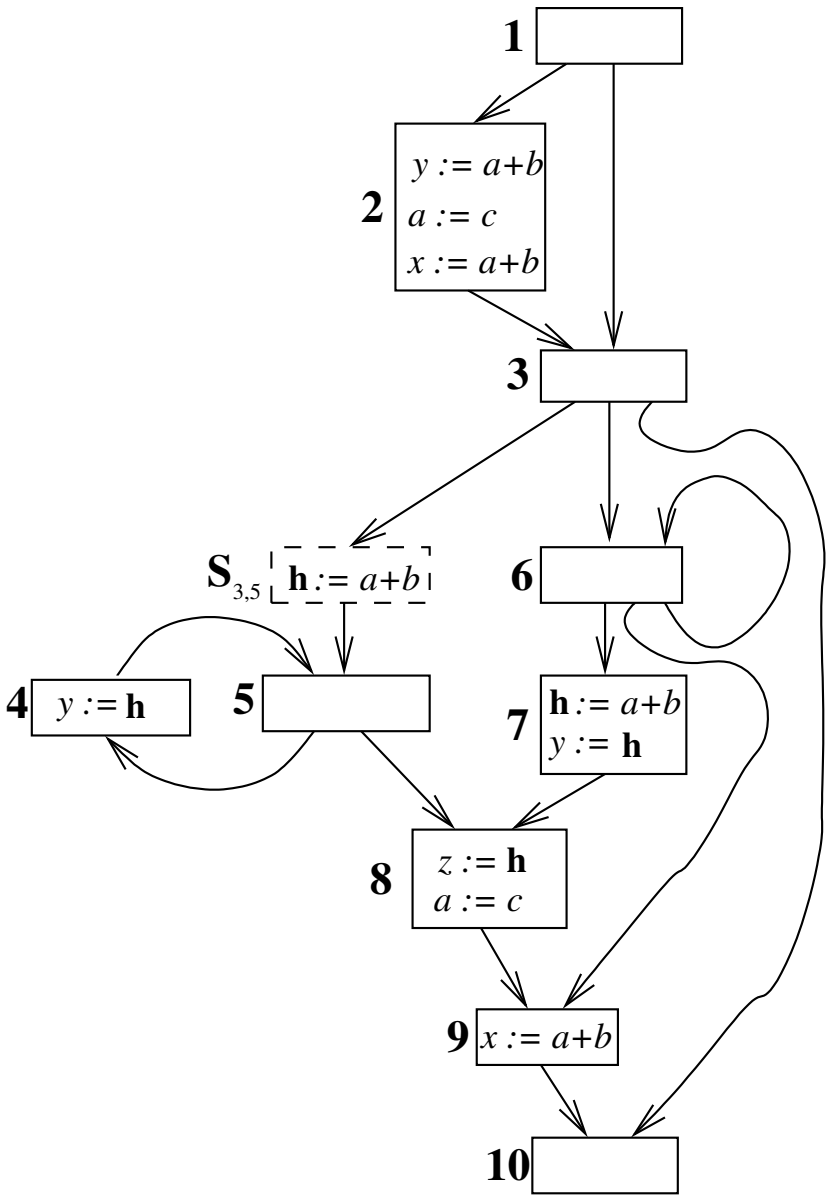
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Latest and Isolated Program Points



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

The Result of the LCM_β Transformation



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Further Reading for Chapter 9

- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
9.1
9.1.1
9.1.2
9.1.3
9.2
9.2.1
9.2.2
9.2.3
9.3
Chap. 10
Chap. 11
Chap. 12
Chap. 13

Chapter 10

Sparse Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Motivation

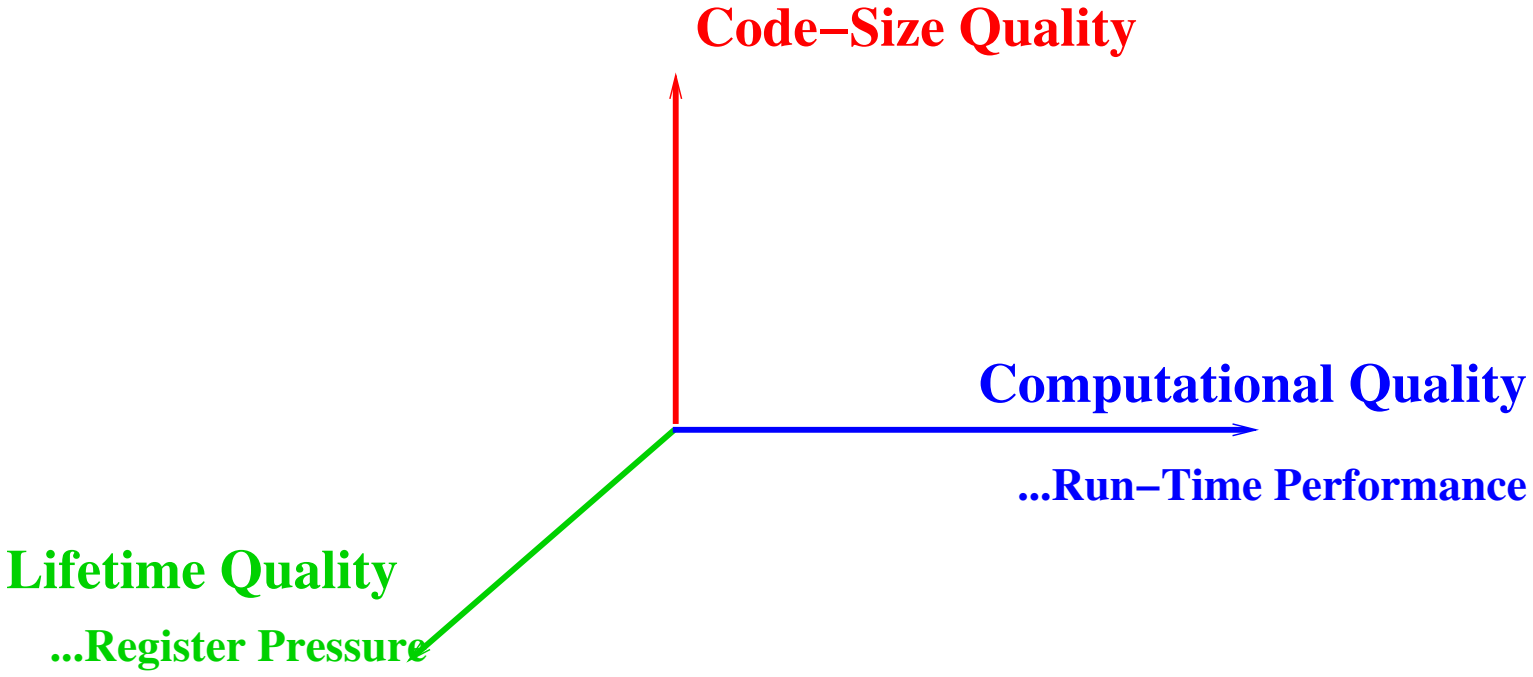
These days Lazy Code Motion is the

- *de-facto* standard algorithm for PRE that is used in current state-of-the-art compilers
  - Gnu compiler family
  - Sun Sparc compiler family
  - ...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

In the following

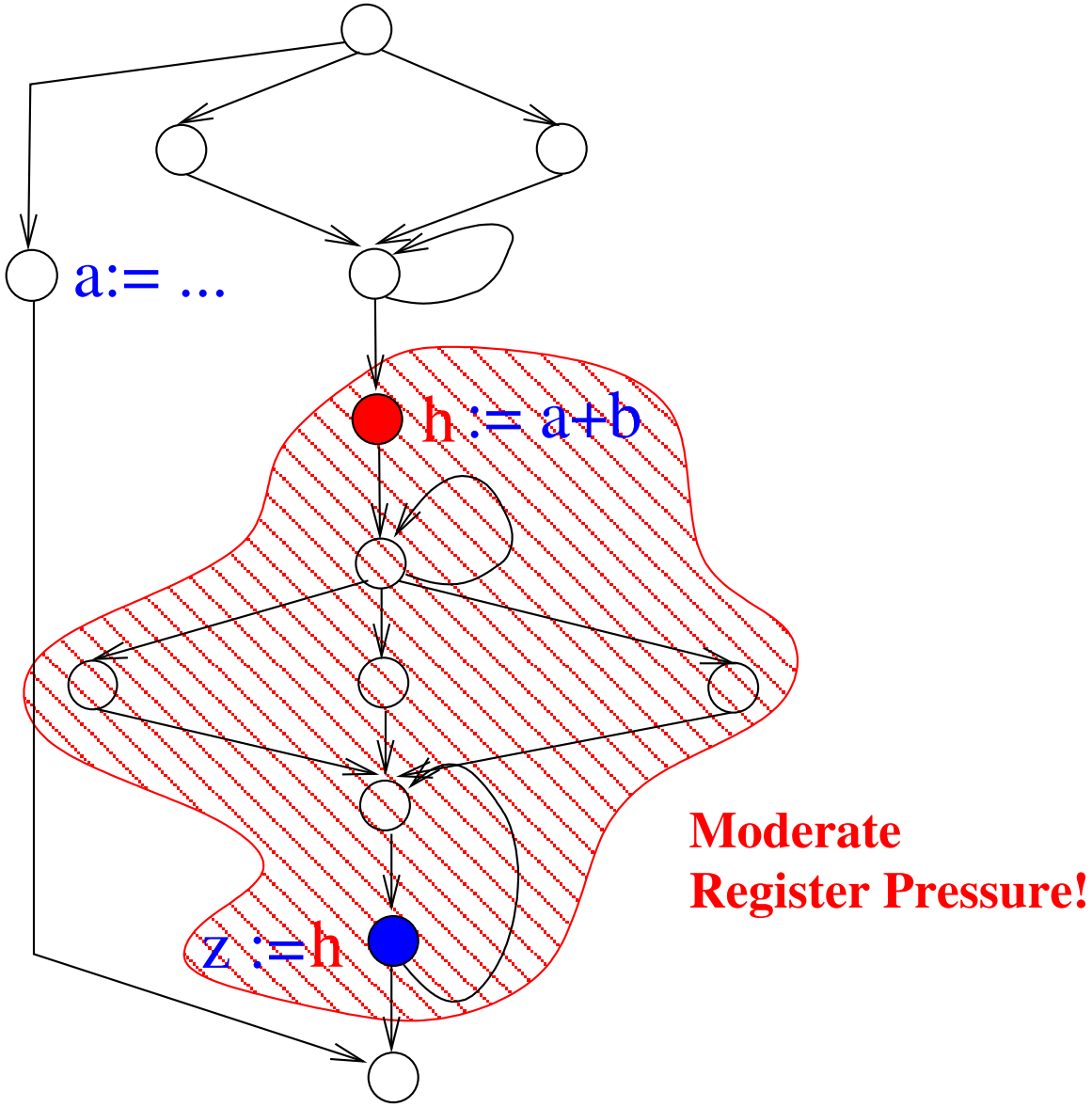
...we consider a (modular) extension of *LCM* in order to take user priorities into account!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

To render possible

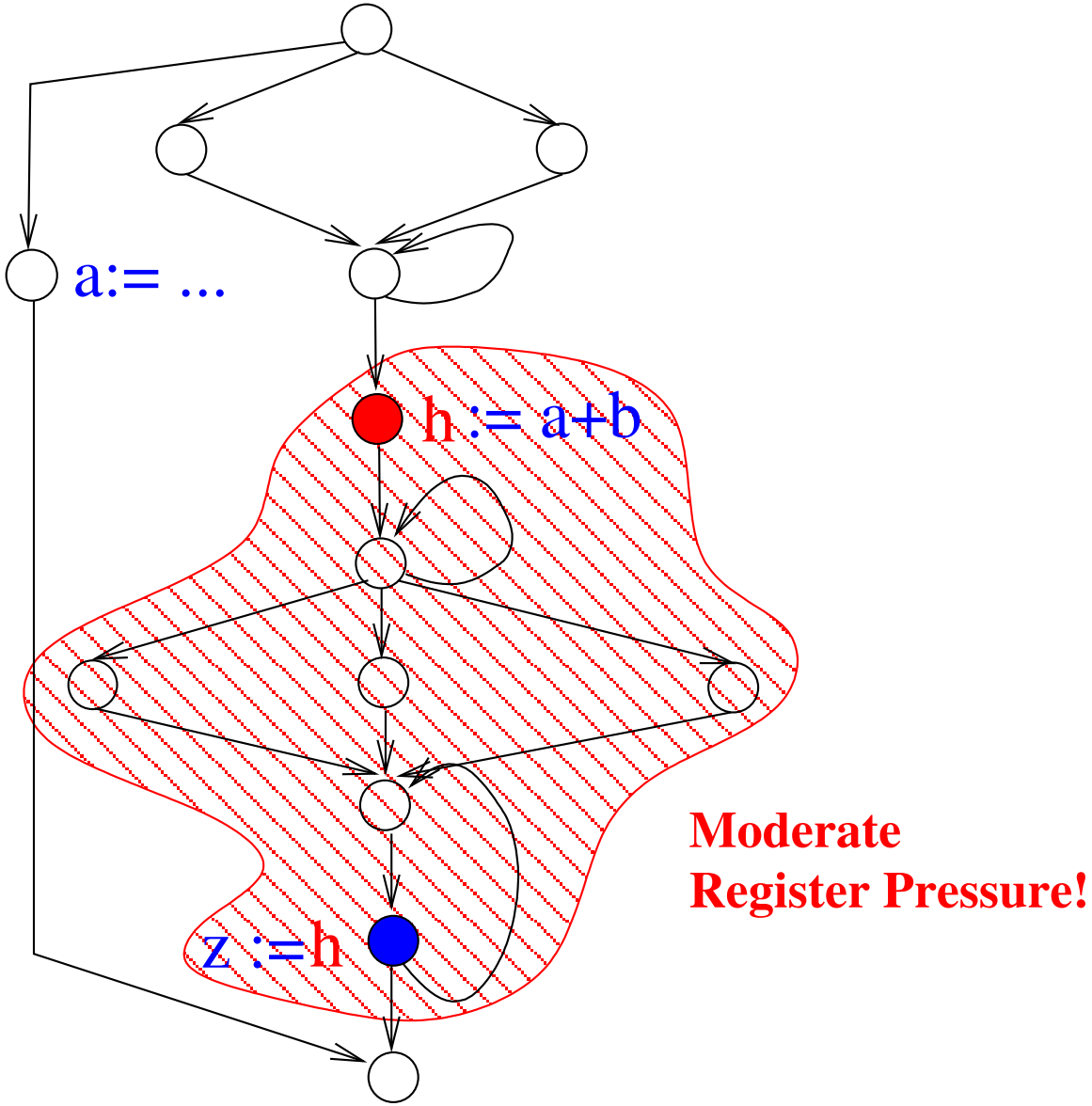
...also the below transformation:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

There is more than speed!

...for instance space!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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 Milliones

...David Tennenhouse (Intel Director of Research), key note lecture at the *20th IEEE Real-Time Systems Symposium* (*RTSS'99*), Phoenix, Arizona, December 1999.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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 Milliones

$\sim$ **2%**

...David Tennenhouse (Intel Director of Research), key note lecture at the *20th IEEE Real-Time Systems Symposium* (*RTSS'99*), Phoenix, Arizona, December 1999.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Think about

...domain-specific processors used in embedded systems:

- Telecommunication
  - Cellular phones, pagers,...
- Consumer electronics
  - MP3-players, cameras, game consoles,...
- Automative field
  - GPS navigation, airbags,...
- ...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Code for Embedded Systems

Demands:

- Performance (often real-time demands)
- Code size (system-on-chip, on-chip RAM/ROM)
- ...

For embedded systems:

- Code size is often more critical than speed!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Code for Embedded Systems (Cont'd)

Demands (and how they are often still addressed):

- Assembler programming
- Manual post-optimization

Shortcomings:

- Error prone
- Delayed time-to-market

...problems getting more severe with increasing complexity.

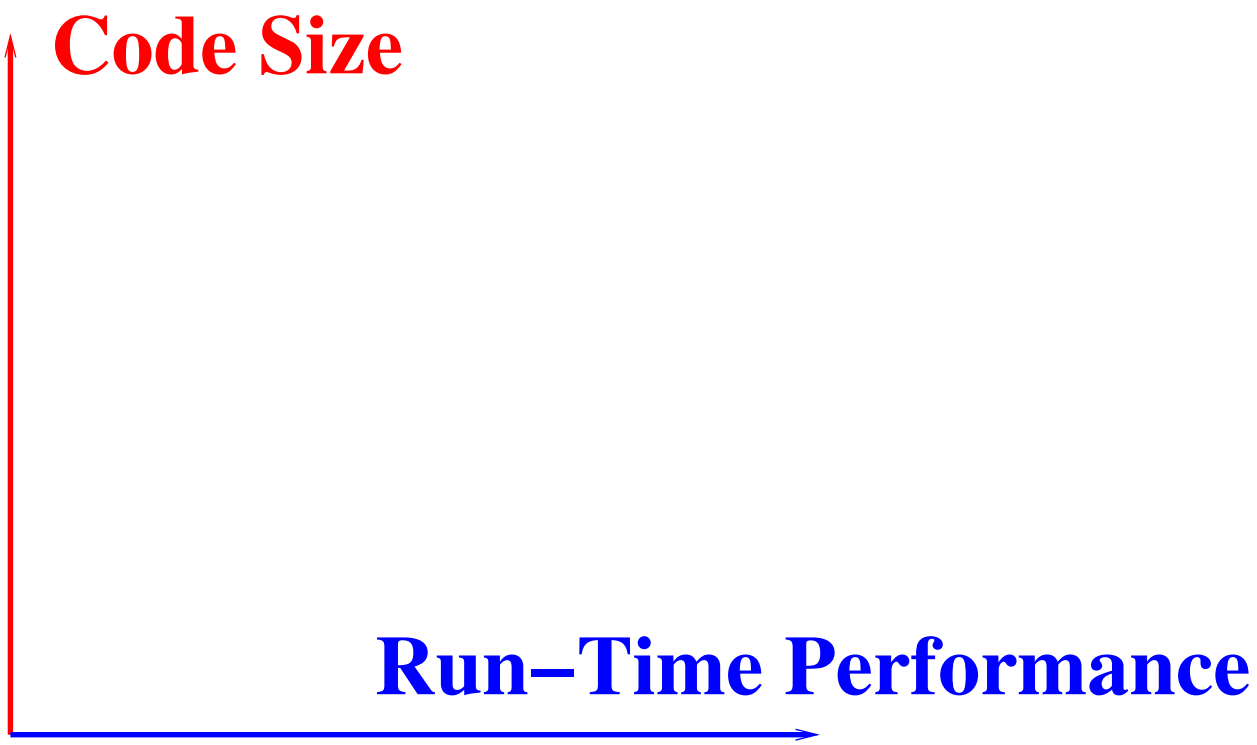
Generally, we observe:

- a trend towards high-level languages programming (C/C++)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

In View of this Trend

...how do traditional compiler and optimizer technologies support the specific demands of code for embedded systems?



...unfortunately, only little.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

W/out Doubt

Traditional Optimizations

- are almost exclusively tuned towards performance optimization
- are not code-size sensitive and in general do not provide any control on their impact on the code size

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

This holds especially

...for code motion based optimizations.

In particular, this includes:

- Partial redundancy elimination
- Partial dead-code elimination (cf. Lecture Course 185.A05 Analysis and Verification)
- Partial redundant-assignment elimination (cf. Lecture Course 185.A05 Analysis and Verification)
- Strength reduction
- ...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Recalling the Essence of PRE

PRE 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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Recalling the Essence of *LCM*

LCM can conceptually be considered the result of a two-stage process:

1. Hoisting Expressions
   ...to their "earliest" safe computation points
2. Sinking Expressions
   ...to their "latest" safe still computationally optimal computation points

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Road to Code-size Sensitive PRE

Classical PRE

- The PRE Algorithm of Morel&Renvoise (CACM 22, 1979)
- Computationally and Lifetime Optimal Code Motion
  - $\rightsquigarrow$ Busy Code Motion (BCM) / Lazy Code Motion (LCM) (Knoop, Rüthing, Steffen, PLDI'92)
    - Distinguished w/ the ACM SIGPLAN Most Influential PLDI Paper Award 2002 (for 1992)
    - Selected for the "20 Years of the ACM SIGPLAN PLDI: A Selection" (60 articles out of about 600 articles)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

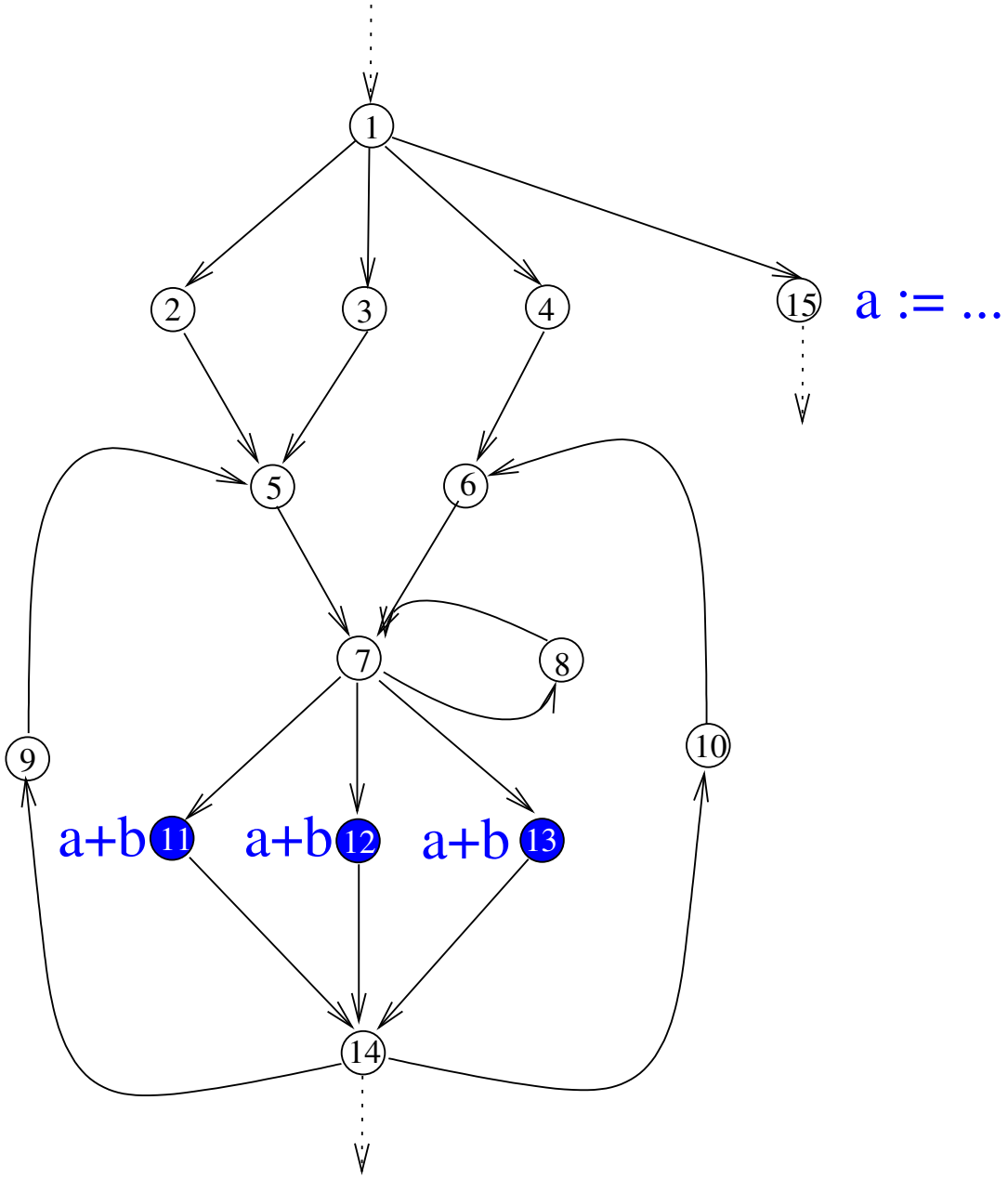
The Road to Code-size Sensitive PRE (Cont'd)

Non-Classical PRE

- Code-size Sensitive PRE
  - $\rightsquigarrow$ Sparse Code Motion (SpCM)
    (Knoop, Rüthing, Steffen, POPL'00)
    - ...modular extension of BCM/LCM
      - $\rightsquigarrow$ Modelling and solving the problem:
        ...based on graph-theoretic means
      - $\rightsquigarrow$ Main Results:
        ...Correctness, Optimality

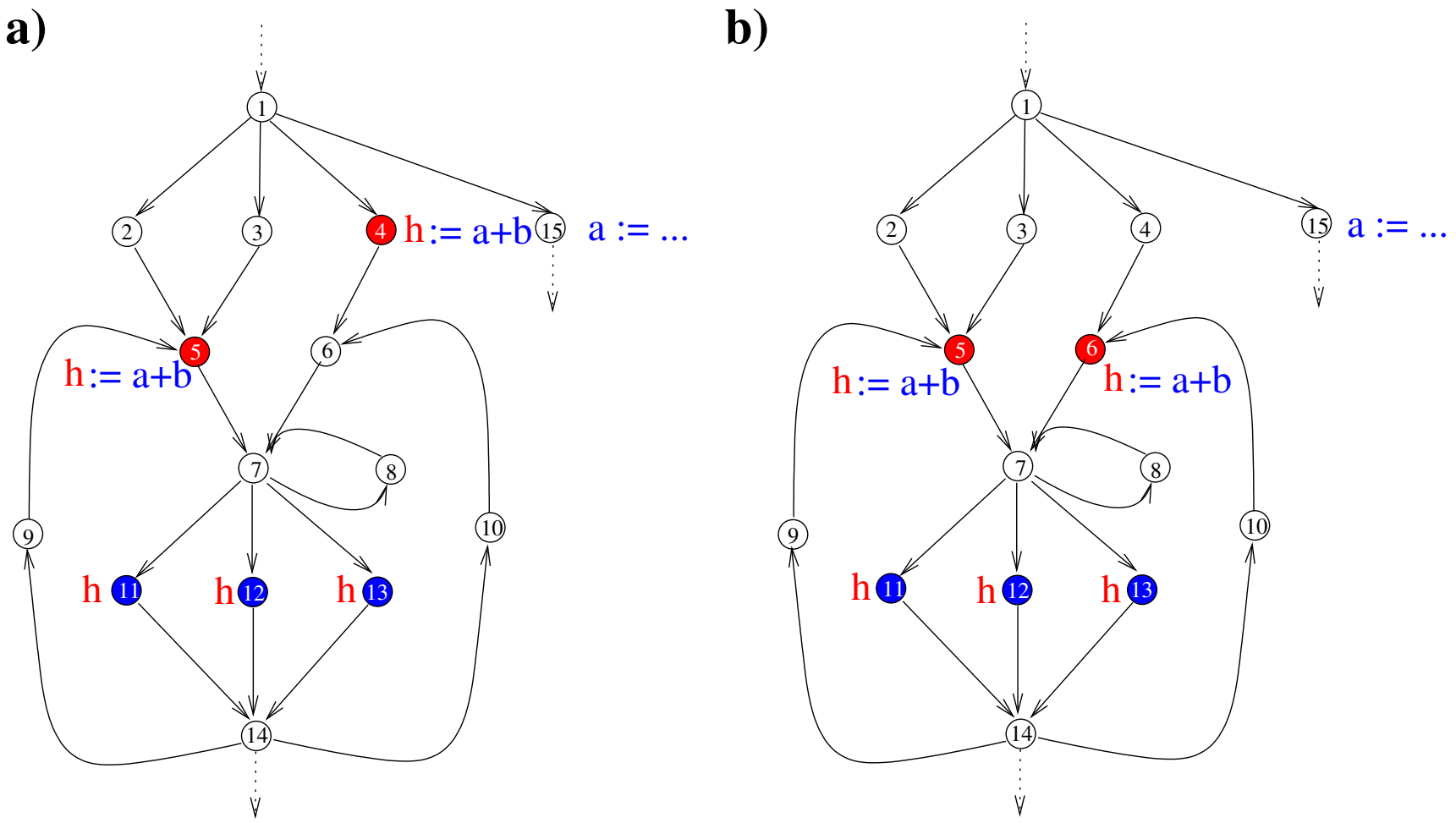
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Running Example (1)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

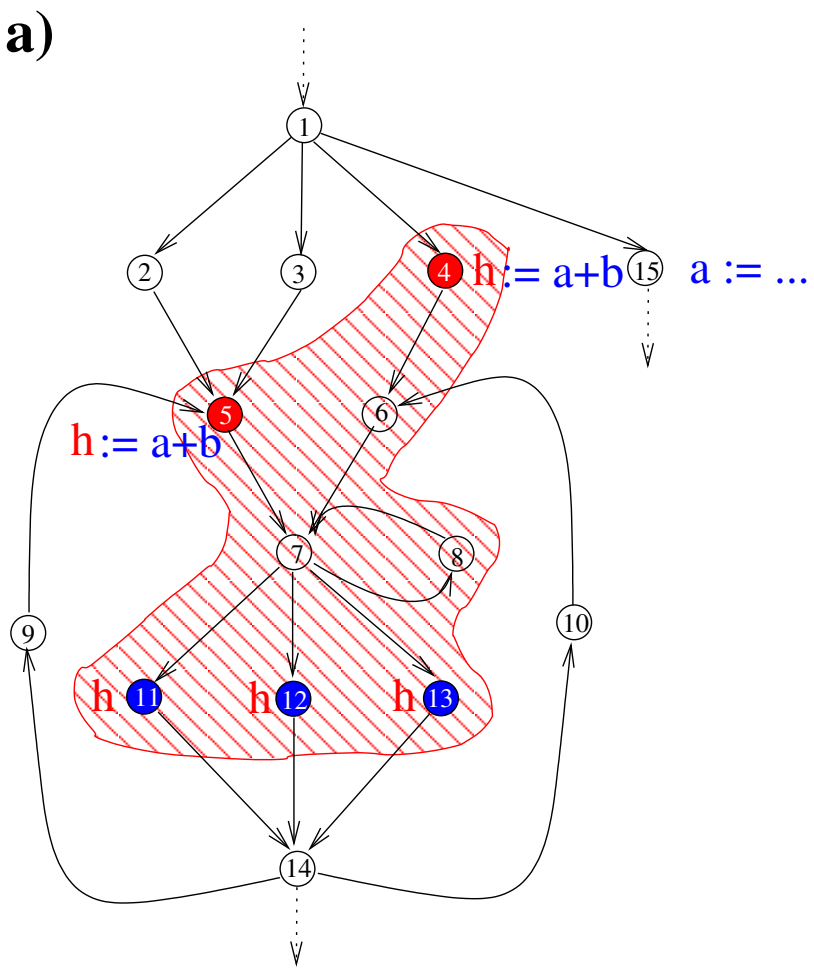
The Running Example (2)



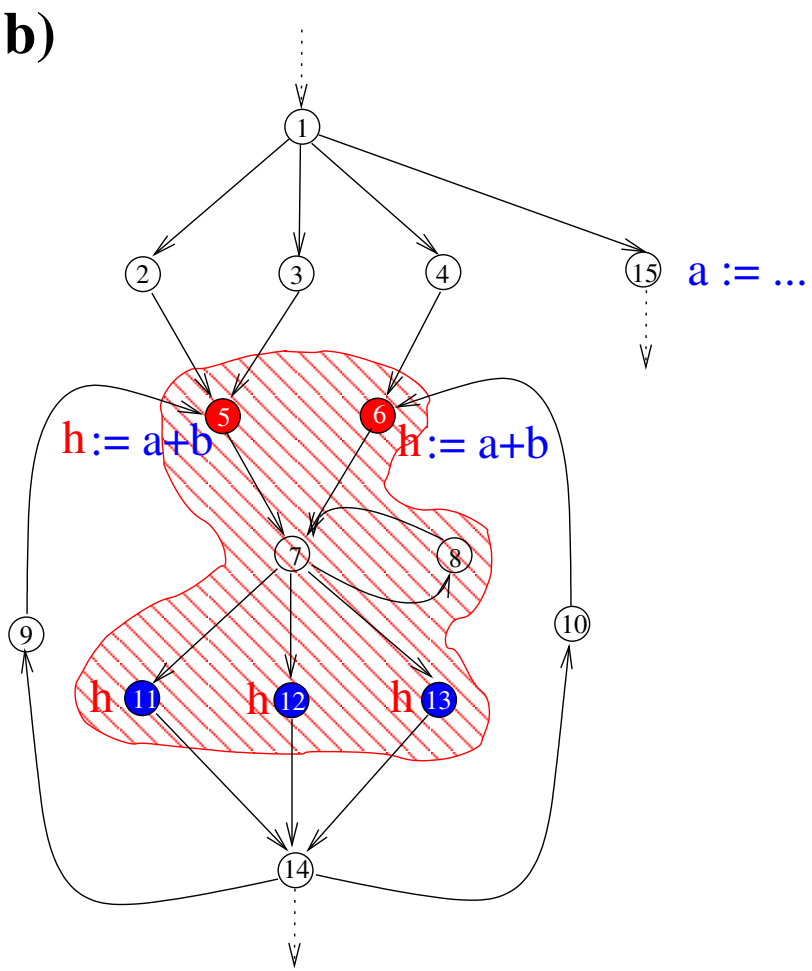
Two Code–size Optimal Programs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The Running Example (3)



SQ > CQ > LQ

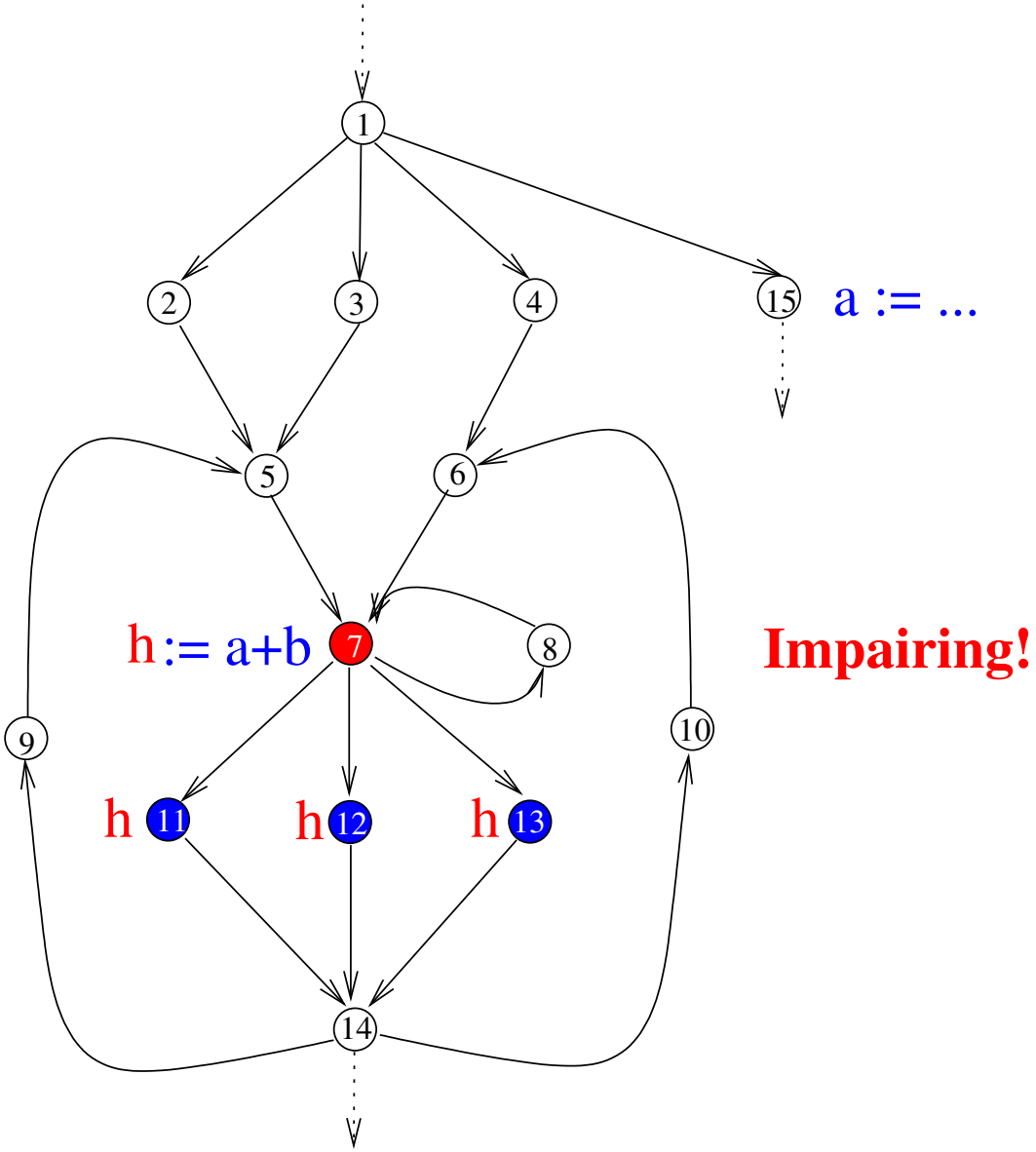


SQ > LQ > CQ

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Running Example (4)

Recall: The below transformation is not desired!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Code-size Sensitive PRE

- ⇝ The Problem
  ...how do we get code-size minimal placement of the computations, i.e., a placement that is
  - admissible (semantics & performance preserving)
  - code-size minimal?
- ⇝ The Solution: A new View to PRE
  ...consider PRE as a trade-off problem: Exchange original computations for newly inserted ones!
- ⇝ The Clou: Use Graph Theory!
  ...reduce the trade-off problem to the computation of tight sets in bipartite graphs based on maximum matchings!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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?

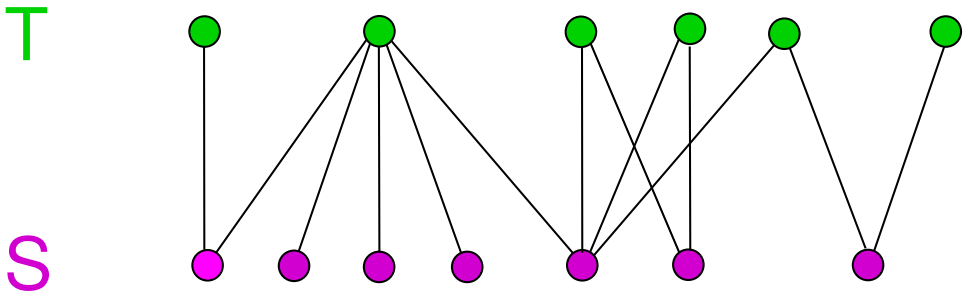
...and to prove:

- 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!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

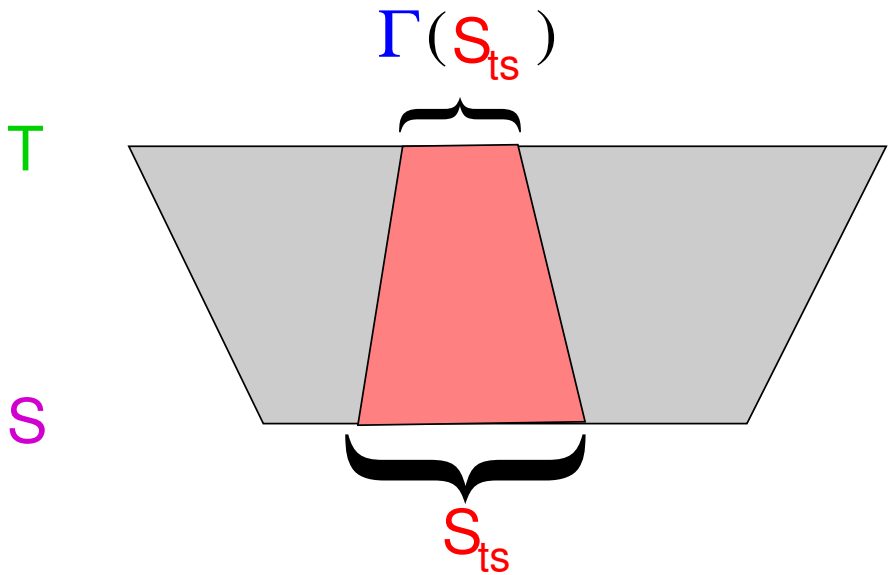
Bipartite Graphs



Tight Set

...of a bipartite graph $(S \cup T, E)$: Subset $S_{ts} \subseteq S$ w/

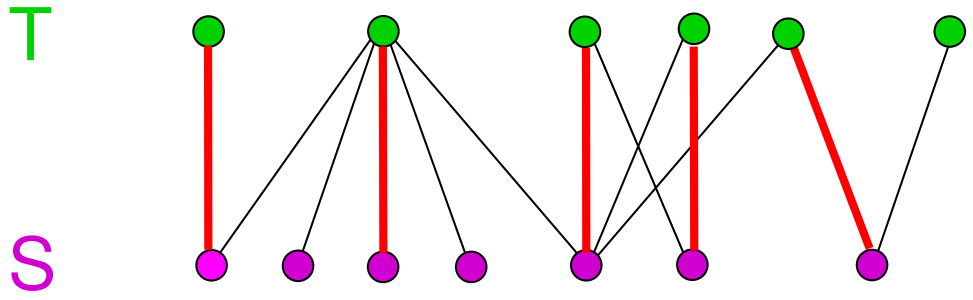
$$\forall\, S' \subseteq S.\ |S_{ts}| - |\Gamma(S_{ts})| \ \geq\ |S'| - |\Gamma(S')|$$



Two Variants: (1) Largest Tight Sets (2) Smallest Tight

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

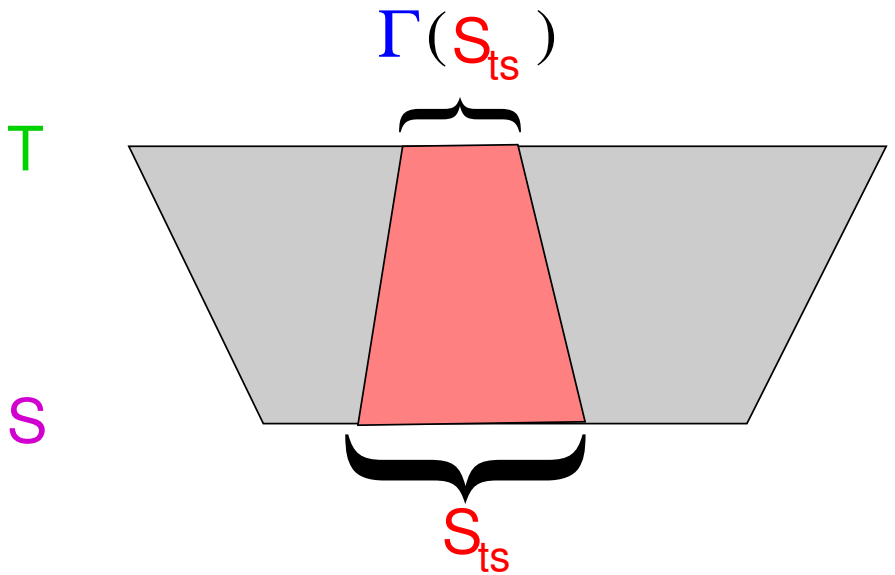
Bipartite Graphs



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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Obviously

...we can make use of off-the-shelve algorithms from graph theory in order to compute

- Maximum matchings and
- Tight sets

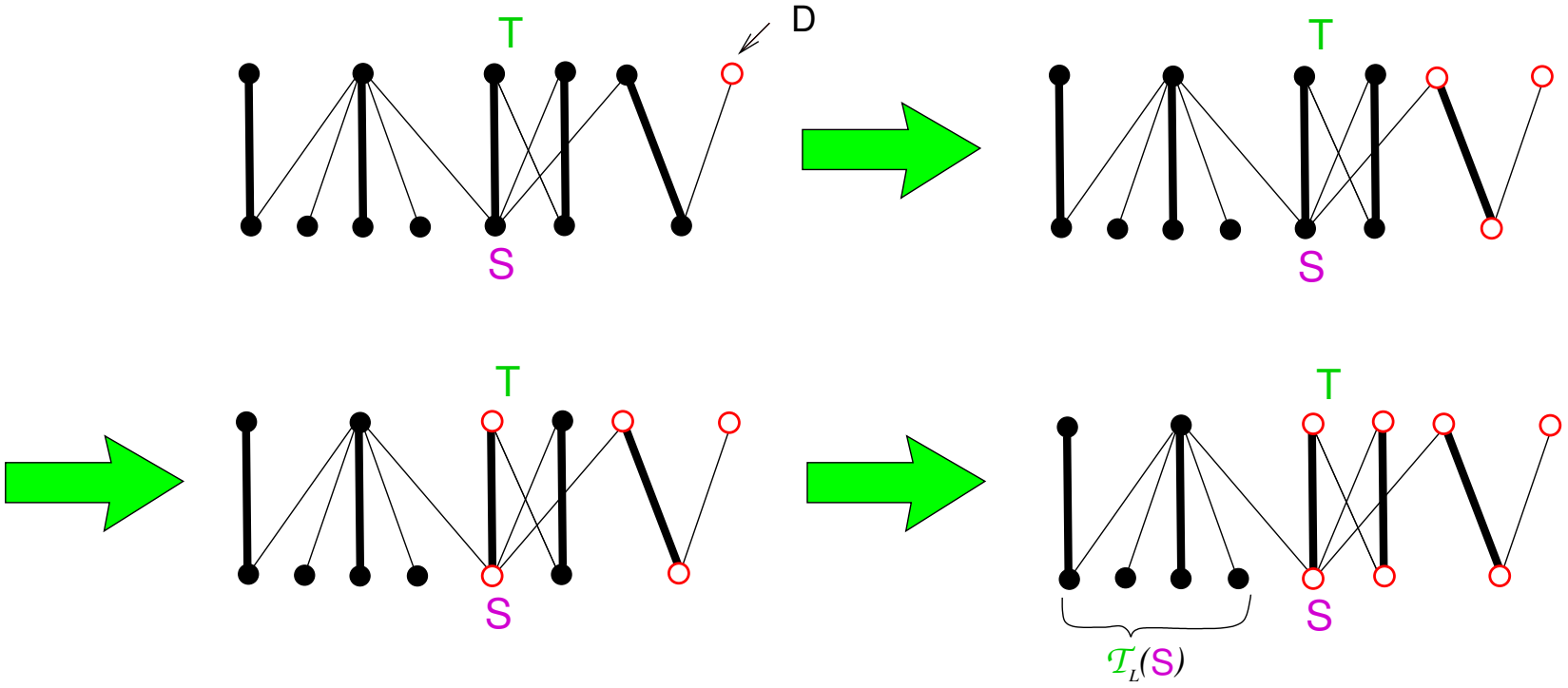
This way the PRE problem boils down to

- constructing the bipartite graph that models the problem!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Computing Largest/Smallest Tight Sets

...based on maximum matchings:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

LTS-Algorithm 10.1: 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`;
$\mathcal{T}_{LaTS}(S) := S_M$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

STS-Algorithmus 10.2: Smallest Tight Sets

Input: A bipartite graph $(S \dot\cup T, E)$, a maximum matching M.

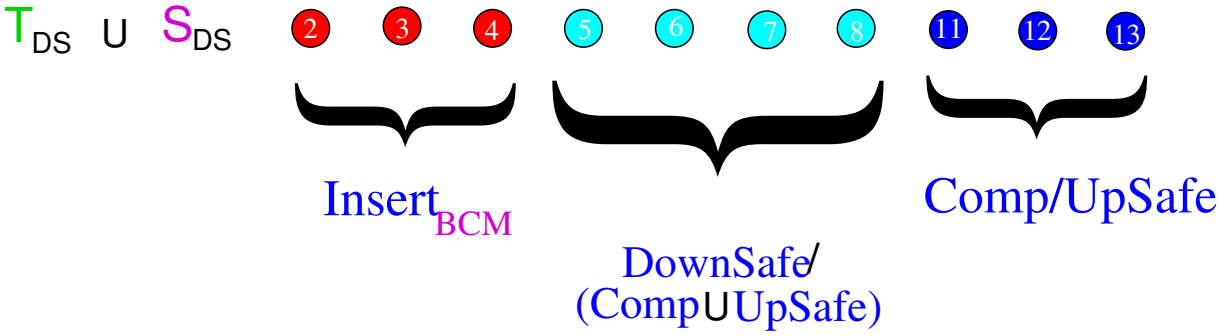
Output: The smallest tight set $\mathcal{T}_{SmTS}(S) \subseteq S$.

$S_M := \emptyset$; $A := \{s \in S \mid s \text{ is unmatched}\}$;
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;
$\mathcal{T}_{SmTS}(S) := S_M$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Modelling the Trade-off Problem

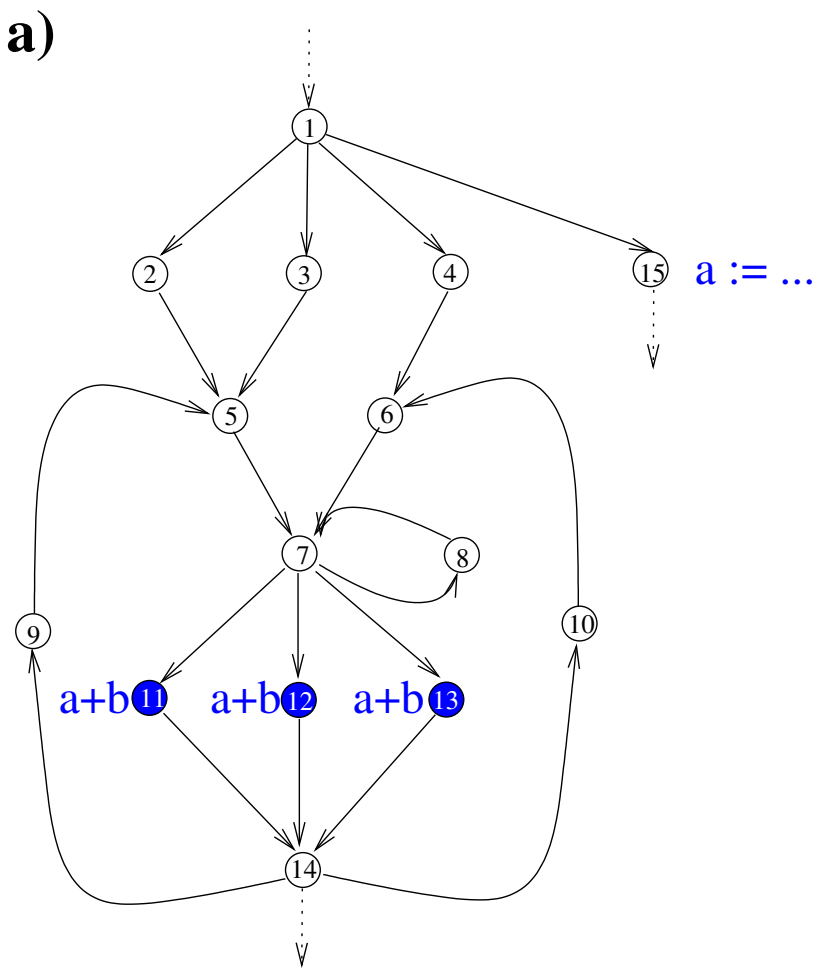
The Set of Nodes

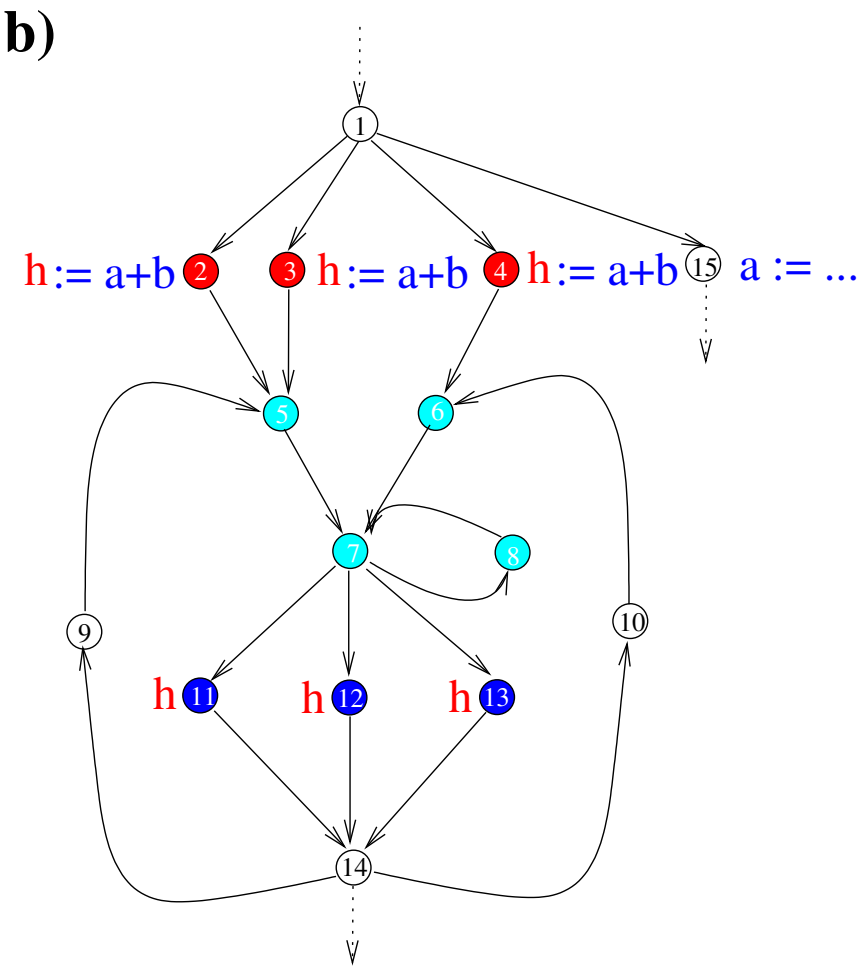


The Set of Edges...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Set of Nodes

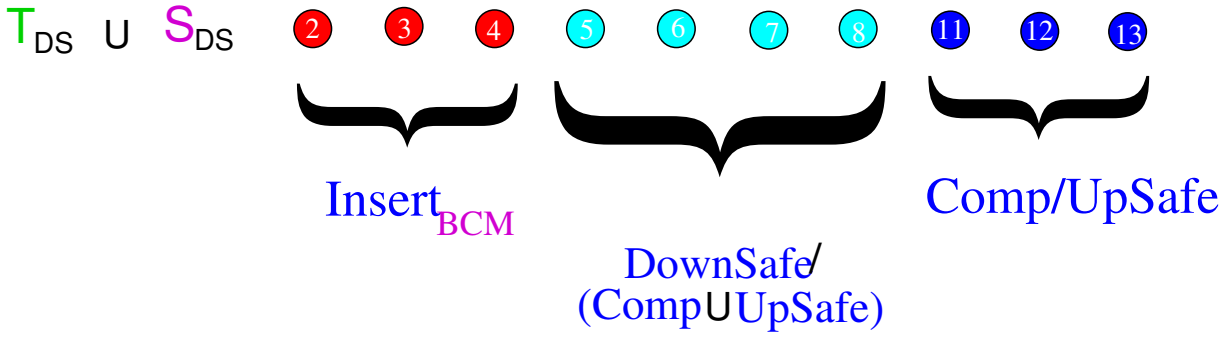




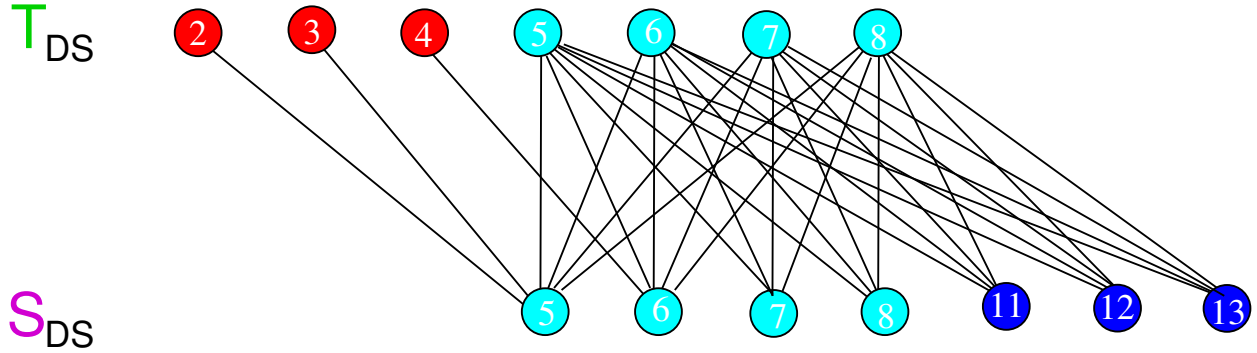
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Modelling the Trade-off Problem

The Set of Nodes



The Bipartite Graph



The Set of Edges

$$\ldots \forall\, n \in S_{DS}\ \forall\, m \in T_{DS}.$$
$$\{n, m\} \in E_{DS} \iff_{df} m \in \textbf{Closure}(pred(n))$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Down-Safety Closures

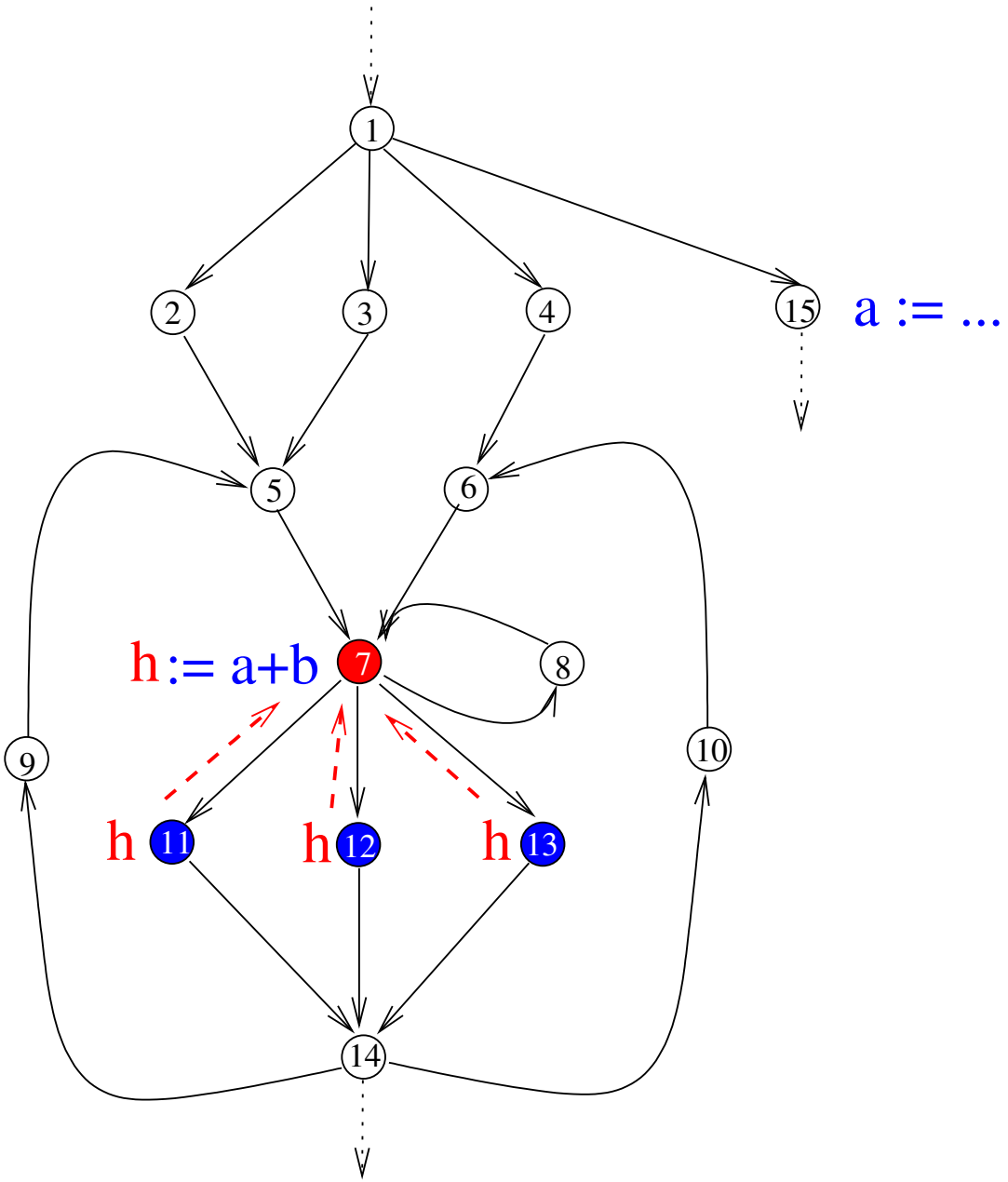
Definition (10.3, Down-Safety Closure)

Let $n \in DownSafe/Upsafe$. Then the Down-Safety Closure $Closure(n)$ is the smallest set of nodes such that

1. $n \in Closure(n)$
2. $\forall\, m \in Closure(n) \setminus Comp.\ succ(m) \subseteq Closure(n)$
3. $\forall\, m \in Closure(n).\ pred(m) \cap Closure(n) \neq \emptyset \ \Rightarrow$
$$pred(m) \setminus UpSafe \subseteq Closure(n)$$

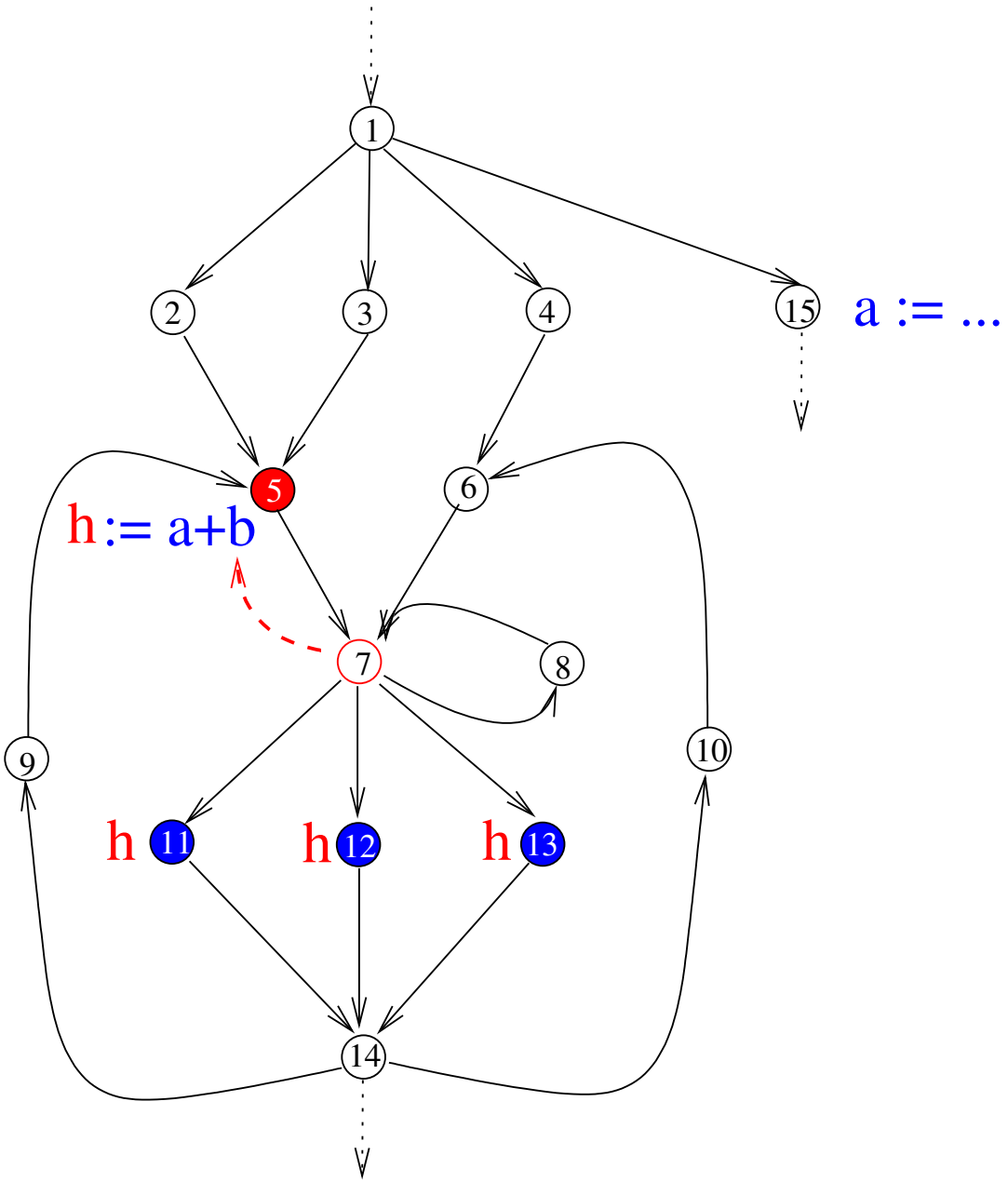
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Down-Safety Closures: The Intuition (1)



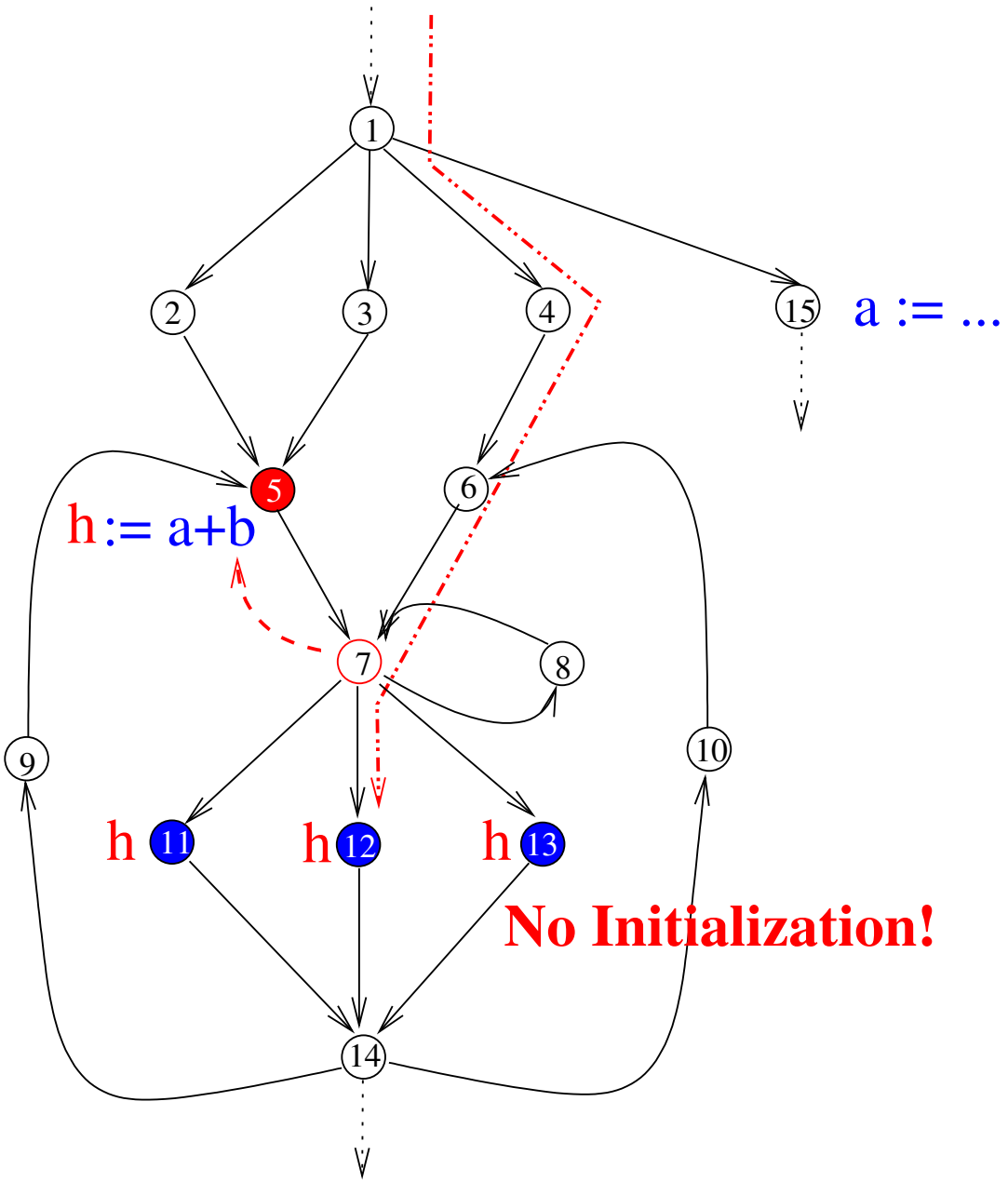
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Down-Safety Closures: The Intuition (2)



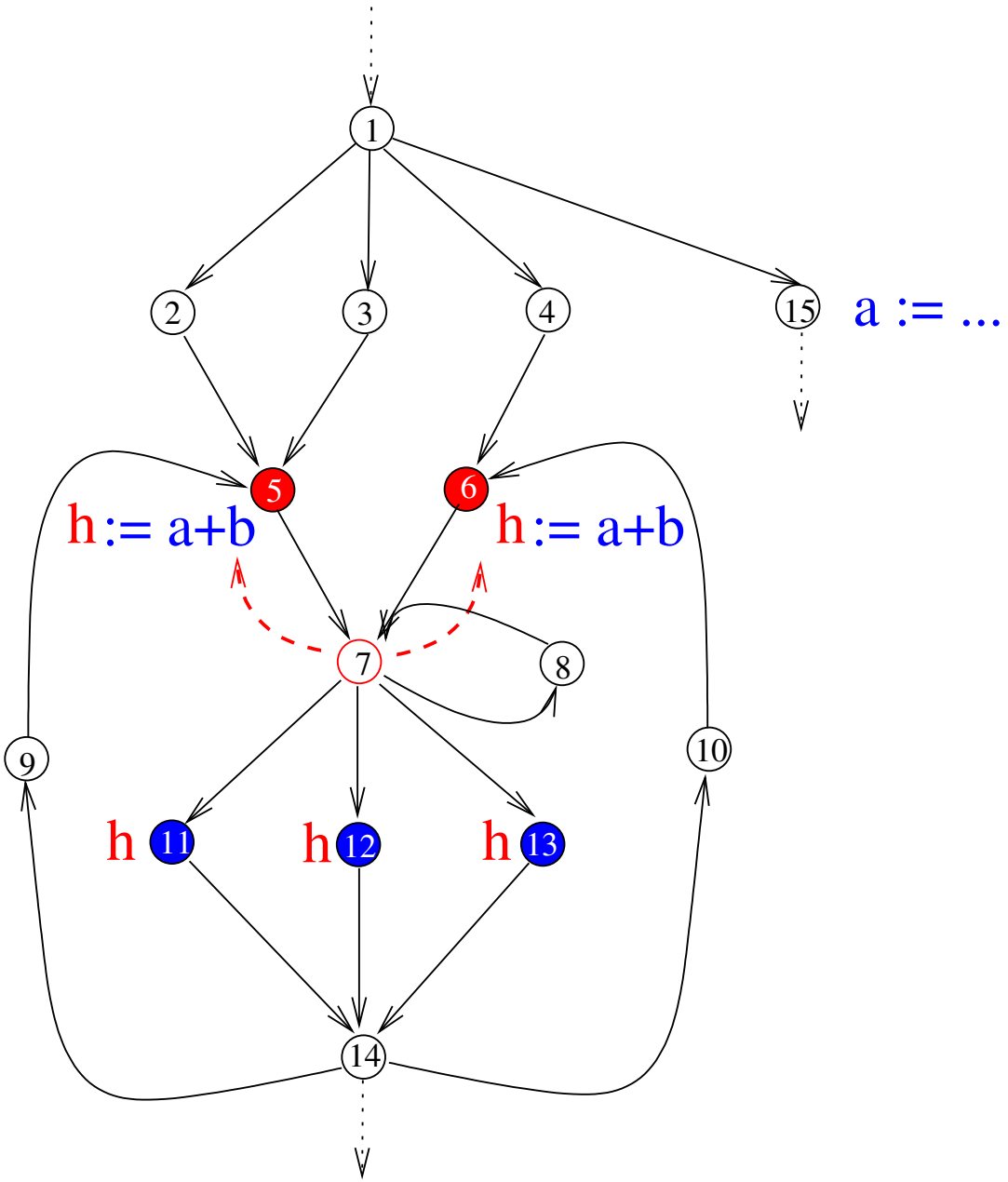
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Down-Safety Closures: The Intuition (3)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Down-Safety Closures: The Intuition (4)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

This intuition

...is condensed in the notion of down-safety closures. Recall:

Definition (10.3, Down-Safety Closure)

Let $n \in \mathit{DownSafe}/\mathit{Upsafe}$. Then the Down-Safety Closure $\mathit{Closure}(n)$ is the smallest set of nodes such that

1. $n \in \mathit{Closure}(n)$
2. $\forall\, m \in \mathit{Closure}(n) \setminus \mathit{Comp}.\ \mathit{succ}(m) \subseteq \mathit{Closure}(n)$
3. $\forall\, m \in \mathit{Closure}(n).\ \mathit{pred}(m) \cap \mathit{Closure}(n) \neq \emptyset \ \Rightarrow$
$\mathit{pred}(m) \setminus \mathit{UpSafe} \subseteq \mathit{Closure}(n)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Down-Safety Regions

...lead to a characterization of semantics-preserving PRE transformations via their insertion points.

Definition (10.4, Down-Safety Region)

A set $\mathcal{R} \subseteq N$ of nodes is a down-safety region iff

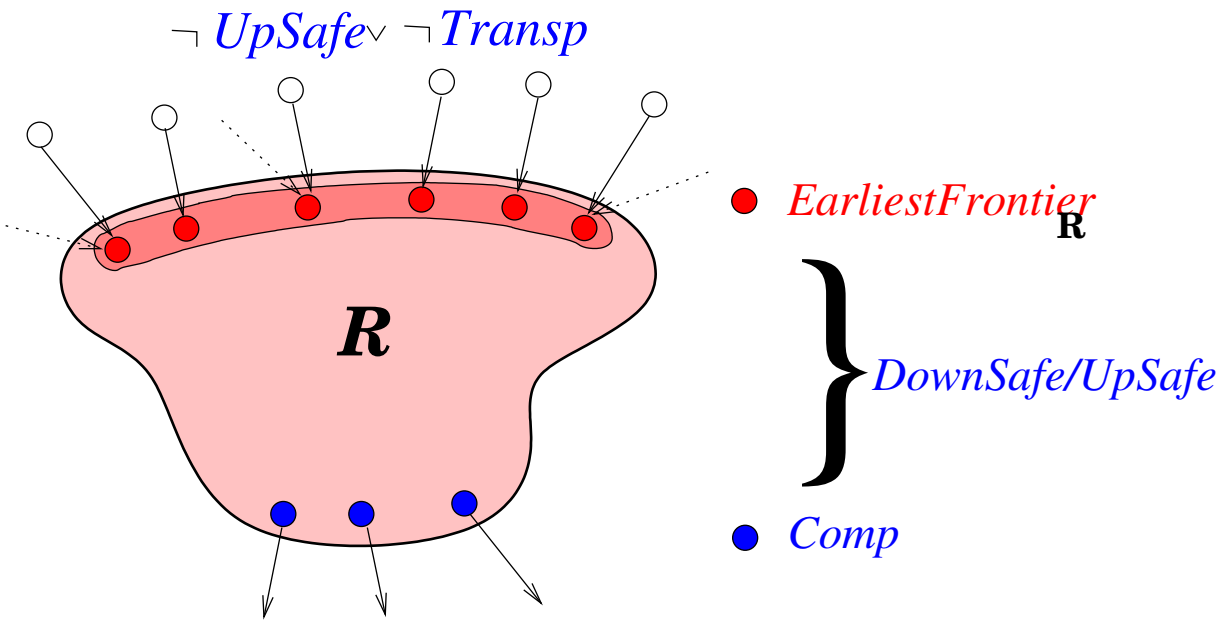
1. $Comp \setminus UpSafe \subseteq \mathcal{R} \subseteq DownSafe \setminus UpSafe$
2. $Closure(\mathcal{R}) = \mathcal{R}$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Fundamental

Theorem (10.5, Initialization Theorem)

Initializations of admissible PRE transformationen are always at the earliestness frontiers of down-safety regions.



...characterizes exactly the set of semantics preserving PRE transformations.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Main Results / Question 1

1. Where to insert computations, why is it correct?

Intuitively: At the earliestness frontier of the DS-region induced by the tight set.

Theorem (10.6, Tight Sets: Insertion Points)

Let $TS \subseteq S_{DS}$ *be a tight set.*
Then $\mathcal{R}_{TS} =_{df} \Gamma(TS) \cup (Comp \setminus UpSafe)$
is a down-safety region w/ $Body_{\mathcal{R}_{TS}} = TS$

Correctness

- An immediate corollary of Theorem 10.6 and the Initialization Theorem 10.5

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Main Results / Question 2

2. What is the impact on the code size?

Intuitively: The difference between the number of inserted and replaced computations.

Theorem (10.7, Down-Safety Regions: Space Gain)

Let $\mathcal{R}$ be a down-safety region w/ $Body_{\mathcal{R}} =_{df} \mathcal{R} \setminus EarliestFrontier_{\mathcal{R}}$

Then

- *Space Gain by Inserting at $EarliestFrontier_{\mathcal{R}}$:* $|Comp \setminus UpSafe| - |EarliestFrontier_{\mathcal{R}}| = |Body_{\mathcal{R}}| - |\Gamma(Body_{\mathcal{R}})| \quad_{df} = defic(Body_{\mathcal{R}})$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Main Results / Question 3

3. Why is the result optimal, i.e., code-size minimal?

Intuitively: Due to a property inherent to tight sets (non-negative deficiency!).

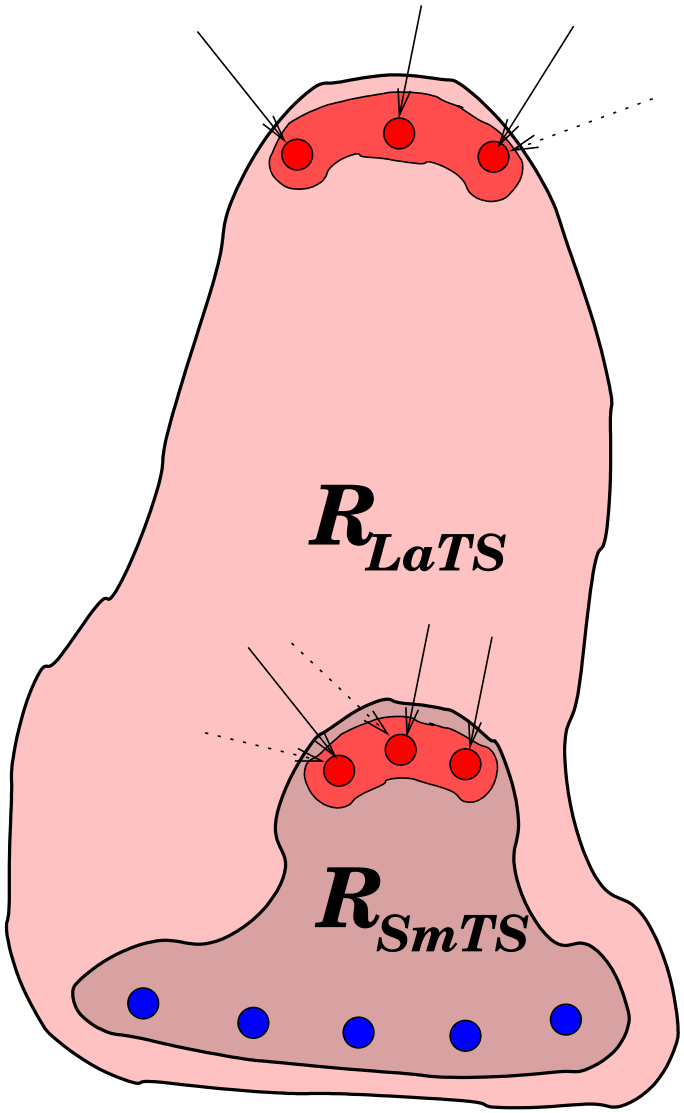
Theorem (10.8, Optimality: Transformation)

Let $TS \subseteq S_{DS}$ be a tight set.

- *Insertion Points:*
  $Insert_{SpCM} =_{df} EarliestFrontier_{\mathcal{R}_{TS}} = \mathcal{R}_{TS} \setminus TS$
- *Space Gain:*
  $defic(TS) =_{df} |TS| - |\Gamma(TS)| \geq 0$ *max.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Largest vs. Smallest Tight Sets: The Impact



- *EarliestFrontier*$_{R_{LaTS}}$

Largest tight sets favor

Computational Quality

→ **Earliestness Principle**

- *EarliestFrontier*$_{R_{SmTS}}$

Smallest tight sets favor

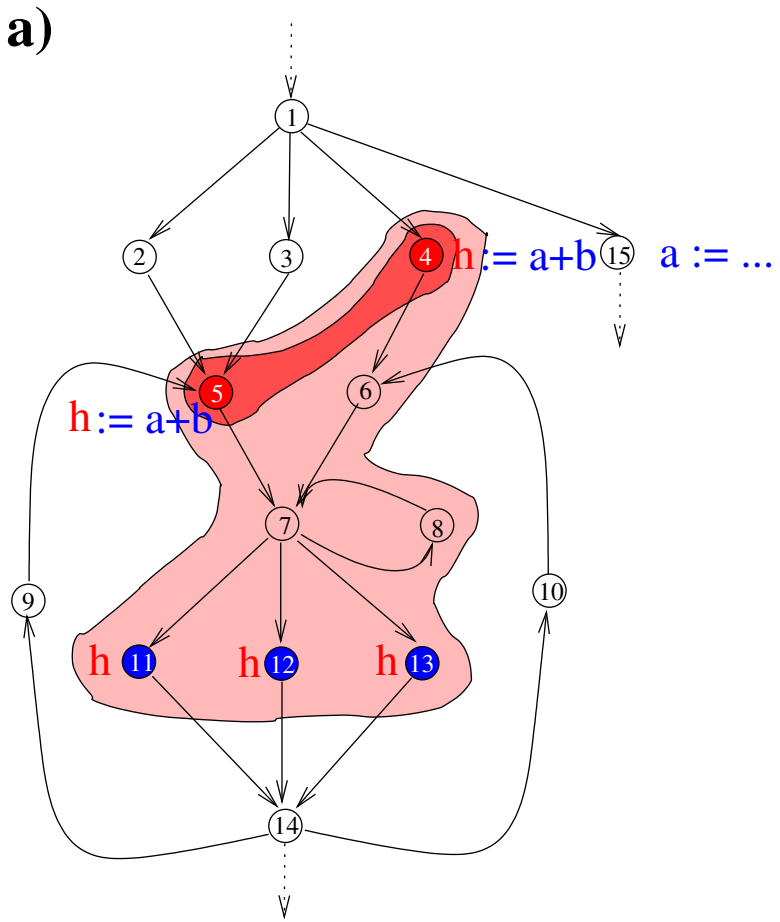
Lifetime Quality

→ **Latestness Principle**

- *Comp*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

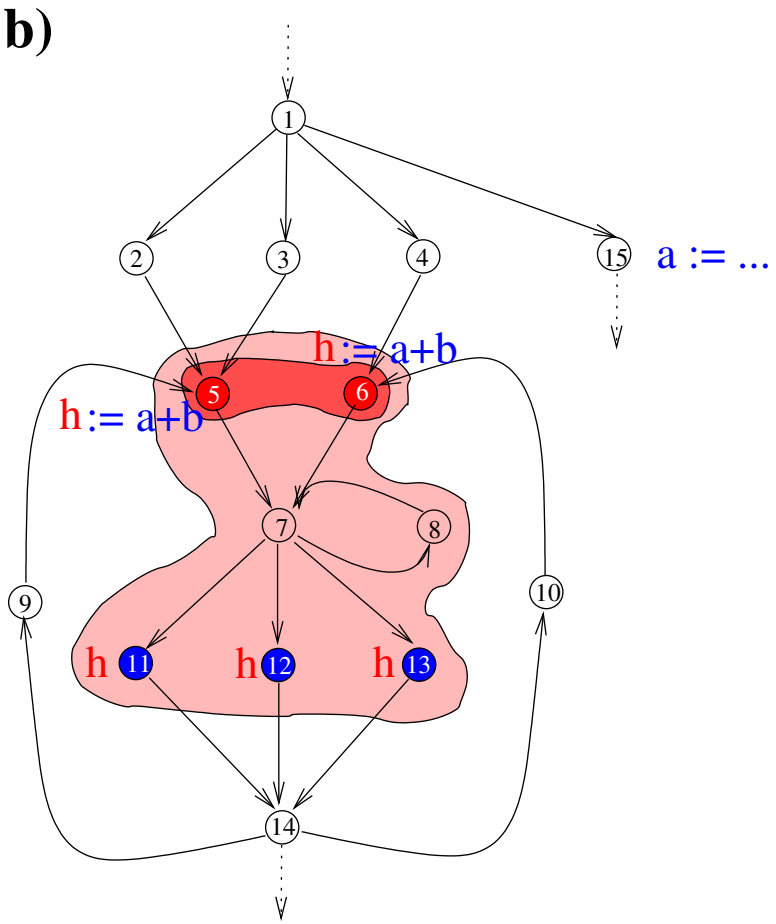
The Impact illustrated on the Running Exam.



Largest Tight Set

(SQ > CQ)

Earliestness Principle



Smallest Tight Set

(SQ > LQ)

Latestness Principle

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Code-size Sensitive PRE at a Glance

Preprocess

- **Optional:** Perform LCM (3 GEN/KILL–DFAs)
- Compute Predicates of LCM 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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

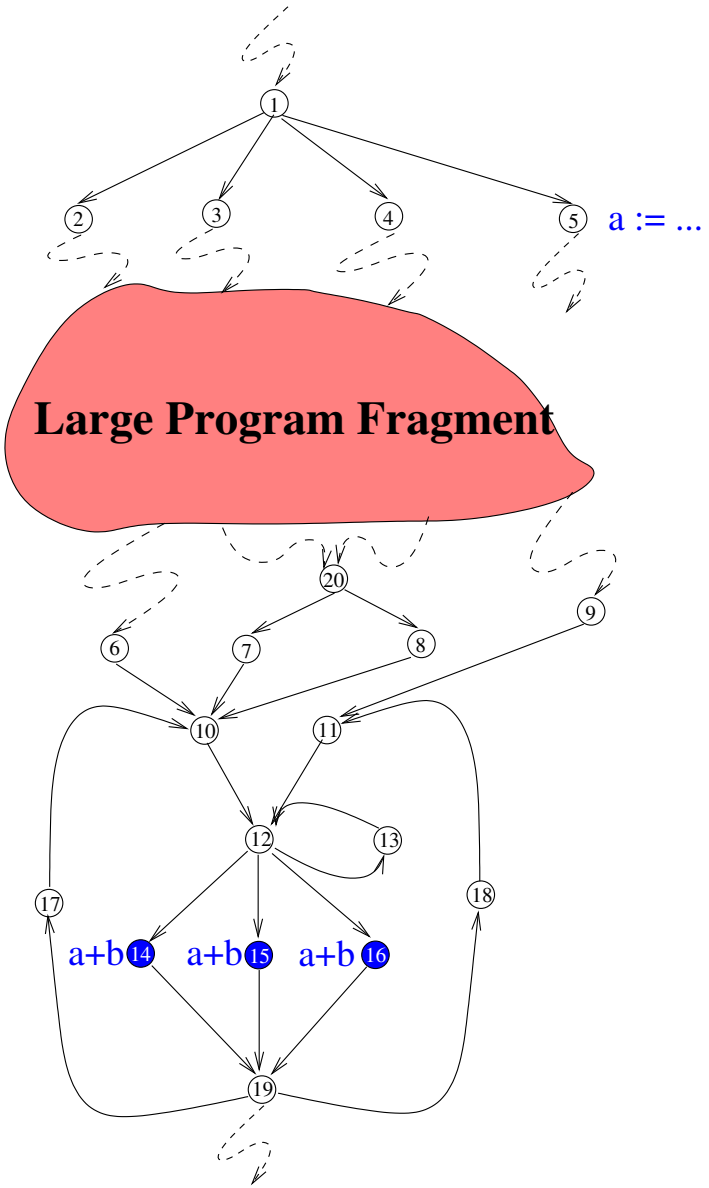
The Cookbook of Recipes for Prioritization

Choice of Priority	Apply	To	Using	Yields	Auxiliary Information Required
$\mathcal{LQ}$	**Not meaningful: The identity, i.e., G itself is optimal!**				
$\mathcal{SQ}$	**Subsumed by $\mathcal{SQ} > \mathcal{CQ}$ and $\mathcal{SQ} > \mathcal{LQ}$!**				
$\mathcal{CQ}$	**BCM**	G			UpSafe(G), DownSafe(G)
$\mathcal{CQ} > \mathcal{LQ}$	**LCM**	G		**LCM**(G)	UpSafe(G), DownSafe(G), Delay(G)
$\mathcal{SQ} > \mathcal{CQ}$	**SpCM**	G	**Largest tight set**	**SpCM**$_{LTS}$(G)	UpSafe(G), DownSafe(G)
$\mathcal{SQ} > \mathcal{LQ}$	**SpCM**	G	**Smallest tight set**		UpSafe(G), DownSafe(G)
$\mathcal{CQ} > \mathcal{SQ}$	**SpCM**	**LCM**(G)	**Largest tight set**		UpSafe(G), DownSafe(G), Delay(G) UpSafe(**LCM**(G)), DownSafe(**LCM**(G))
$\mathcal{CQ} > \mathcal{SQ} > \mathcal{LQ}$	**SpCM**	**LCM**(G)	**Smallest tight set**		UpSafe(G), DownSafe(G), Delay(G) UpSafe(**LCM**(G)), DownSafe(**LCM**(G))
$\mathcal{SQ} > \mathcal{CQ} > \mathcal{LQ}$	**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)))

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Flexibility as the Reward of SpCM (1)

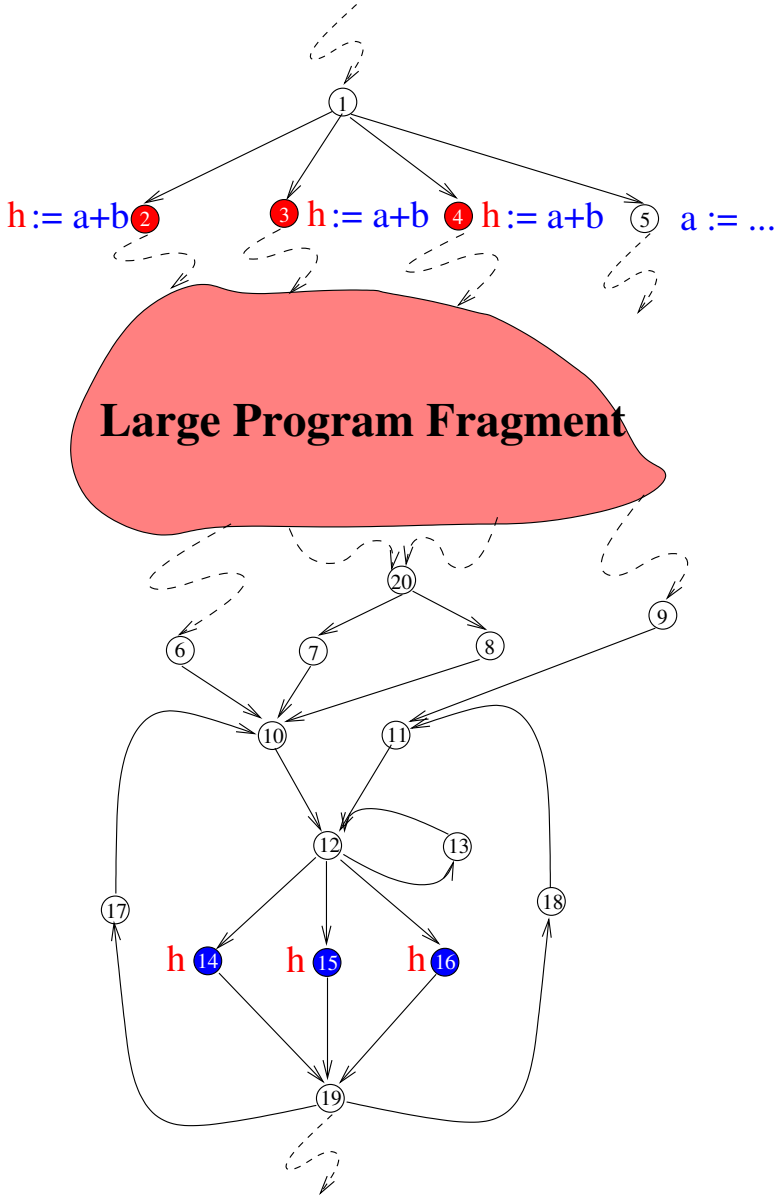
The original program:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Flexibility as the Reward of SpCM (2)

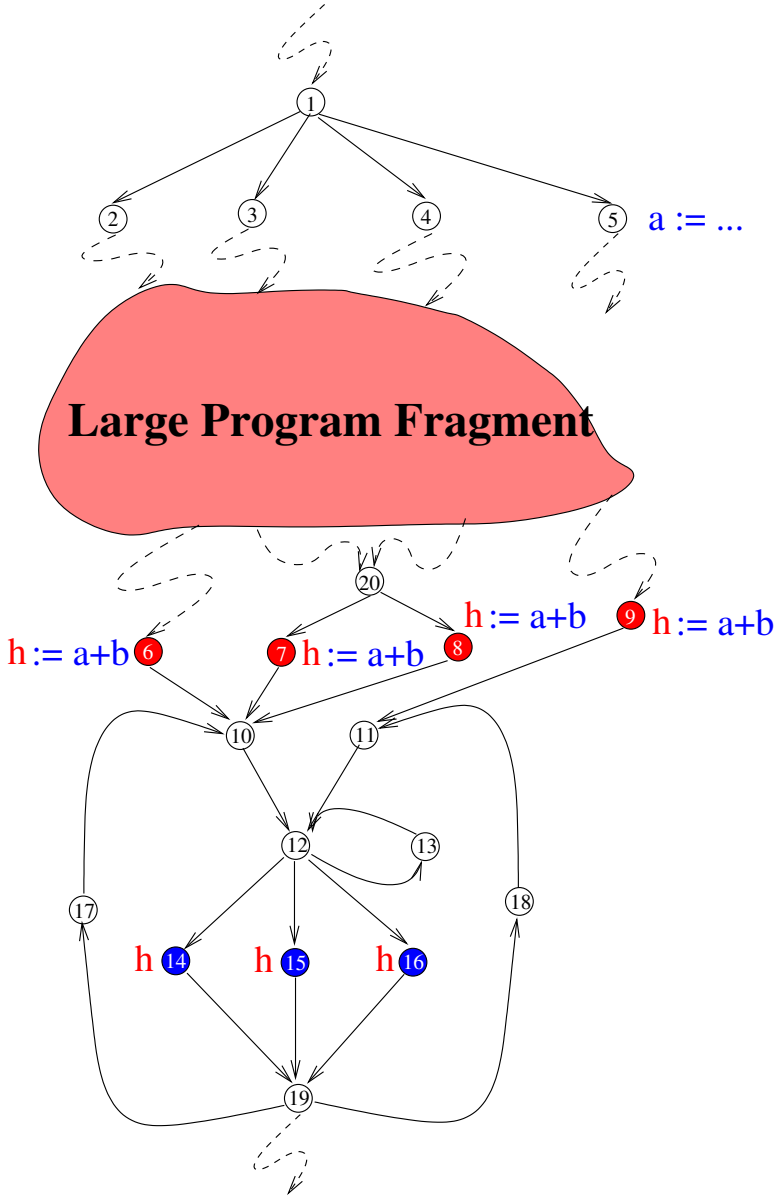
BCM: A computationally optimal program ($\mathcal{CQ}$)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Flexibility as the Reward of SpCM (3)

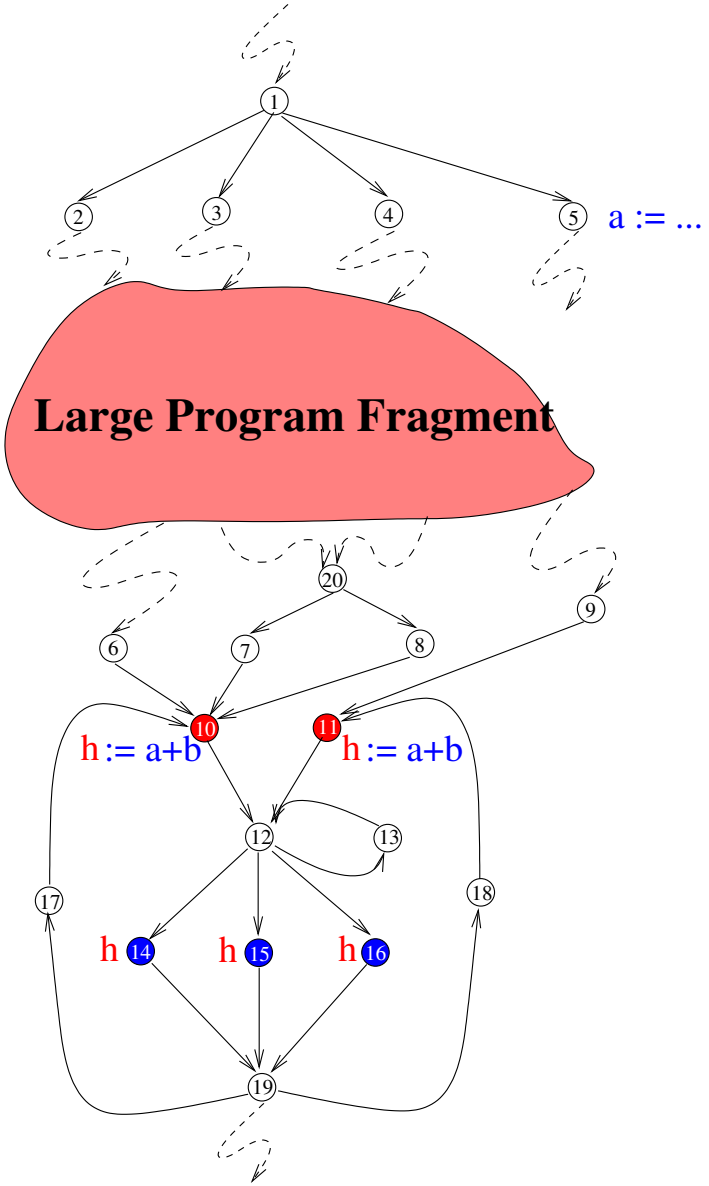
LCM: A computationally & lifetime opt. program ($\mathcal{CQ} > \mathcal{LQ}$)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Flexibility as the Reward of SpCM (4)

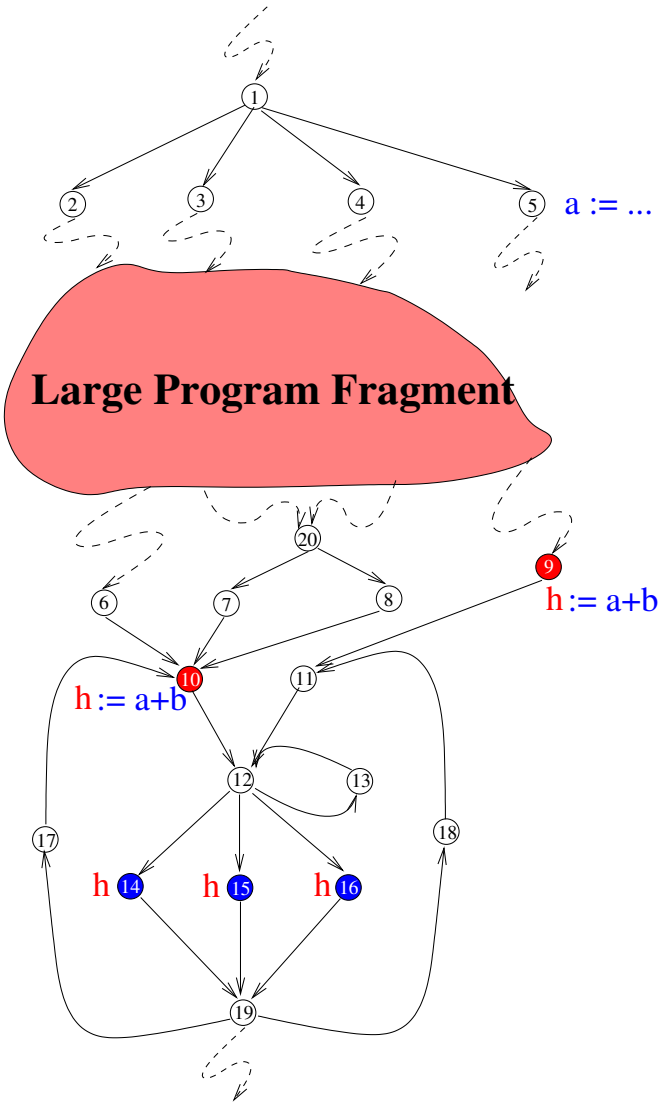
SpCM: A code-size & lifetime opt. program ($\mathcal{SQ} > \mathcal{LQ}$)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Flexibility as the Reward of SpCM (5)

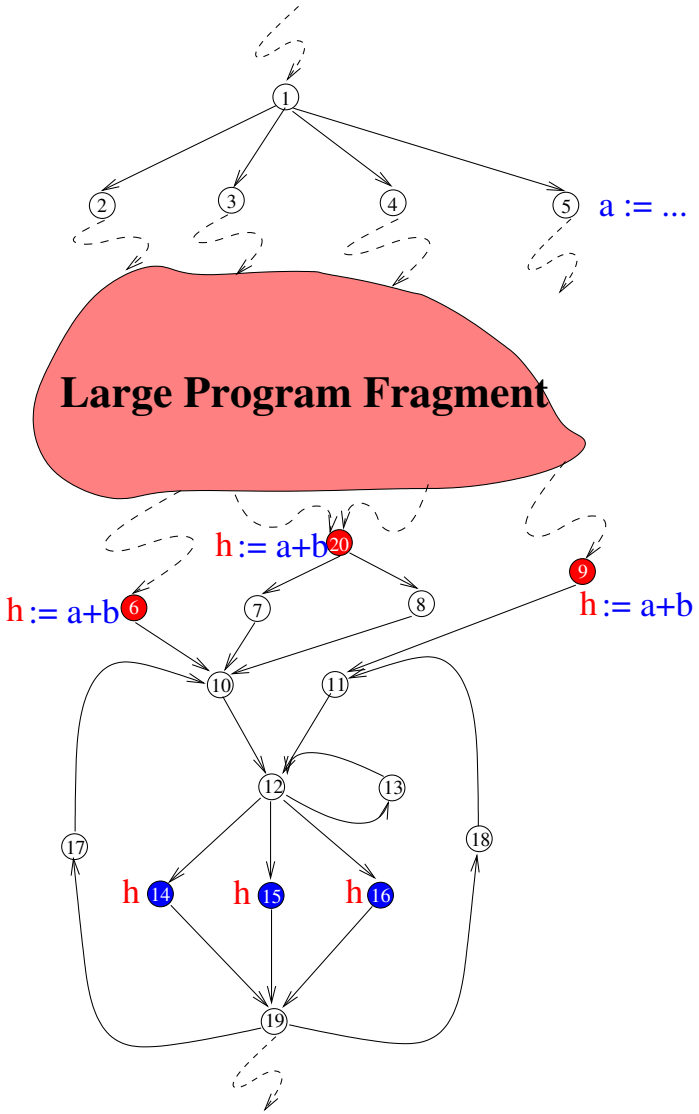
SpCM: A computationally & lifetime best code-size optimal program ($\mathcal{SQ} > \mathcal{CQ} > \mathcal{LQ}$)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Flexibility as the Reward of SpCM (6)

SpCM: A code-size & lifetime best computationally optimal program ($\mathcal{CQ} > \mathcal{SQ} > \mathcal{LQ}$)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

On the Origin and Advancement of PRE (1)

- 1958: A first glimpse of PRE
  - $\rightsquigarrow$ Ershov's work on "On Programming of Arithmetic Operations."
- $<$ 1979: Special techniques
  - $\rightsquigarrow$ Total redundancy elimination, loop invariant code motion
- 1979: The origin of modern PRE
  - $\rightsquigarrow$ Morel/Renvoise's seminal work on PRE
- $<$ ca. 1992: Heuristic improvements of the PRE algorithm of Morel and Renvoise
  - $\rightsquigarrow$ Dhamdhere [1988, 1991]; Drechsler, Stadel [1988]; Sorkin [1989]; Dhamdhere, Rosen, Zadeck [1992], Briggs, Cooper [1994],...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

On the Origin and Advancement of 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
  - ⇝ first to rigorously be proven correct and optimal
- 2000: *SpCM*: The origin of code-size sensitive PRE [Knoop, Rüthing, Steffen (POPL 2000)]
  - ⇝ first to allow prioritization of goals
  - ⇝ rigorously be proven correct and optimal
  - ⇝ first to bridge the gap between traditional compilation and compilation for embedded systems

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

On the Origin and Advancement of PRE (3)

- Since ca. 1997: A new strand of research on PRE
  - $\rightsquigarrow$ Speculative PRE: Gupta, Horspool, Soffa, Xue, Scholz, Knoop,...
- 2005: Another fresh look at PRE (as maximum flow problem)
  - $\rightsquigarrow$ Unifying PRE and Speculative PRE [Xue, Knoop (CC 2006)]

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Further Reading for Chapter 10

- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter 11

Lazy Strength Reduction

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Objective

Developing a program optimization that

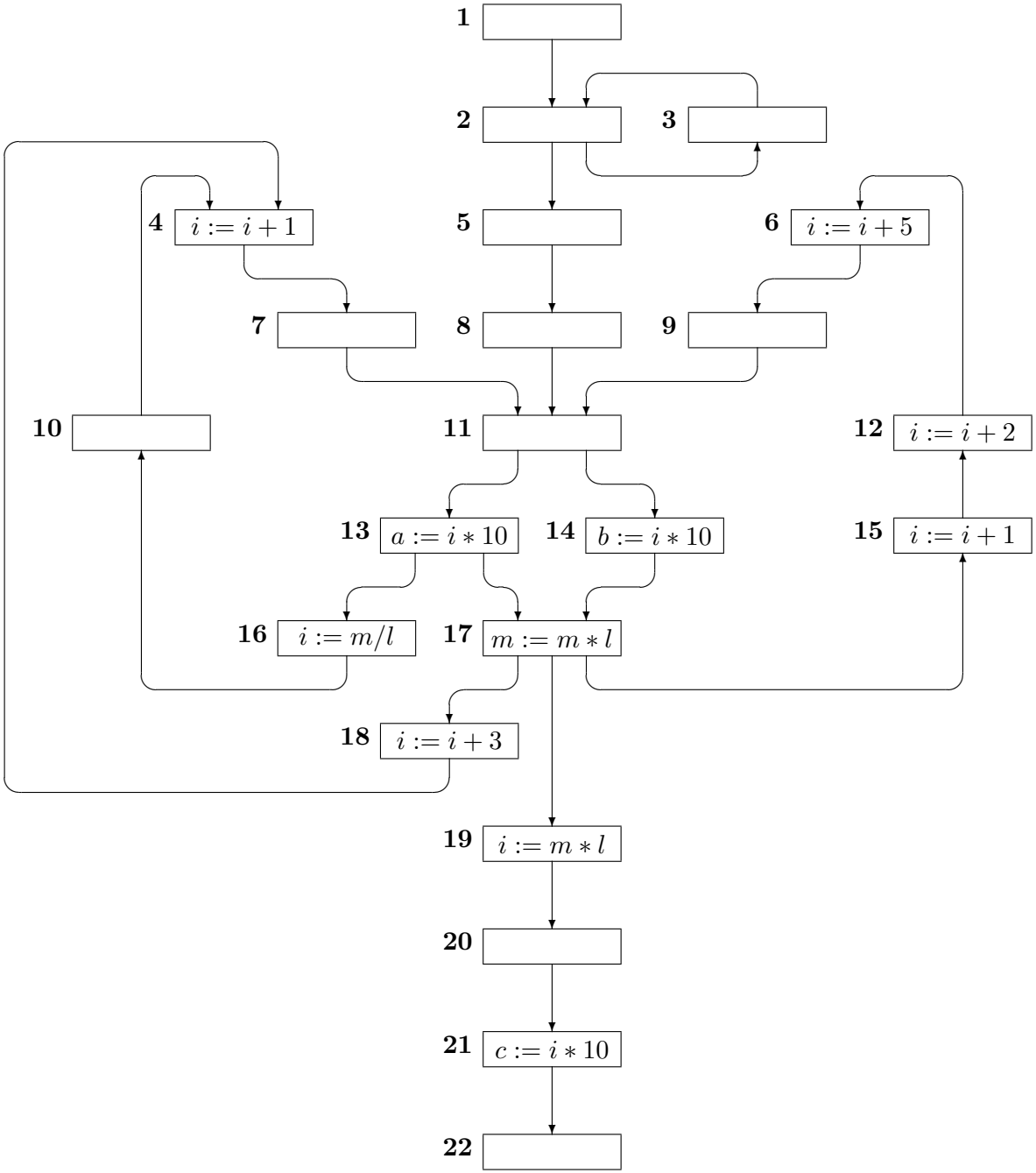
- uniformly covers
  - Partial Redundancy Elimination (PRE) and
  - Strength Reduction (SR)
- avoids superfluous register pressure due to unnecessary code motion
- requires only uni-directional data flow analyses

The Approach:

- Stepwise and modularly extending the *BCM* and the *LCM* to arrive at the
  - Busy (*BSR*) and Lazy Strength Reduction (*LSR*)

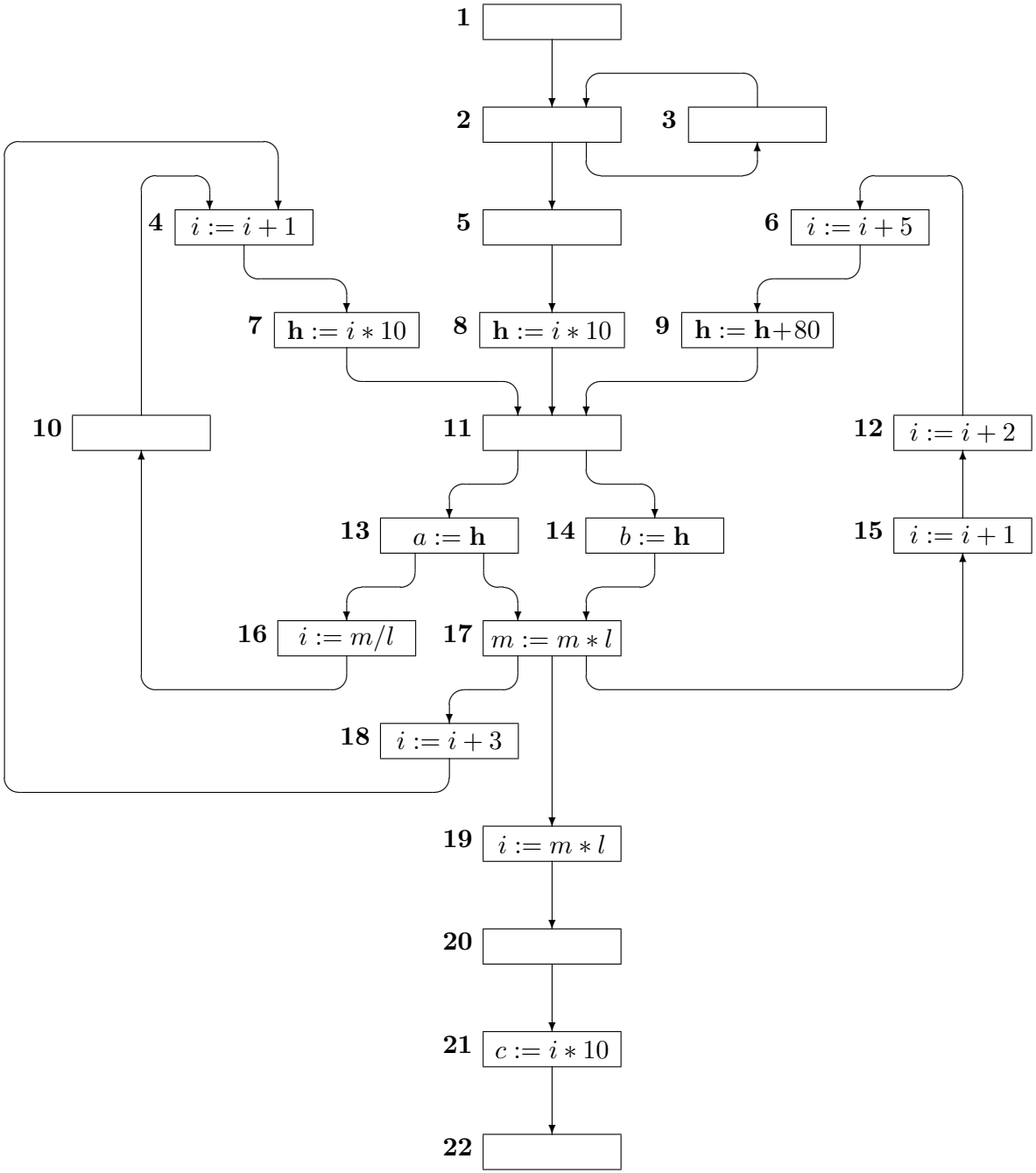
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Illustration: The Original Program



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Result of Lazy Strength Reduction



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

From PRE towards LSR (1)

First, the notion of a candidate expression has to be adapted:

Candidate expressions for

- PRE: Each term t
- SR: Terms of the form $v * c$, where
  - v is a variable
  - c is a source code constant

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

From PRE towards LSR (2)

Second, the set of local predicates has to be extended:

- $Used(n) =_{df} v * c \in SubTerms(t)$
- $Transp(n) =_{df} x \not\equiv v$
- $SR\text{-}Transp(n) =_{df} Transp(n) \vee t \equiv v + d$ with $d \in \mathbf{C}$

Intuitively

The value of a candidate expression is

- killed at a node n, if $\neg(Transp(n) \vee SR\text{-}Transp(n))$
- injured at a node n, if $\neg Transp(n) \wedge SR\text{-}Transp(n)$

Important: Injured but not killed values can be

- cured by inserting an update assignment of the form $\mathbf{h} := \mathbf{h} + Eff(n)$ where $Eff(n) =_{df} c * d$.

  Note that $Eff(n)$ can be computed at compile time since both c and d are source code constants.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Extending BCM straigthforward to SR

...leads to Simple Strength Reduction (*SSR*).

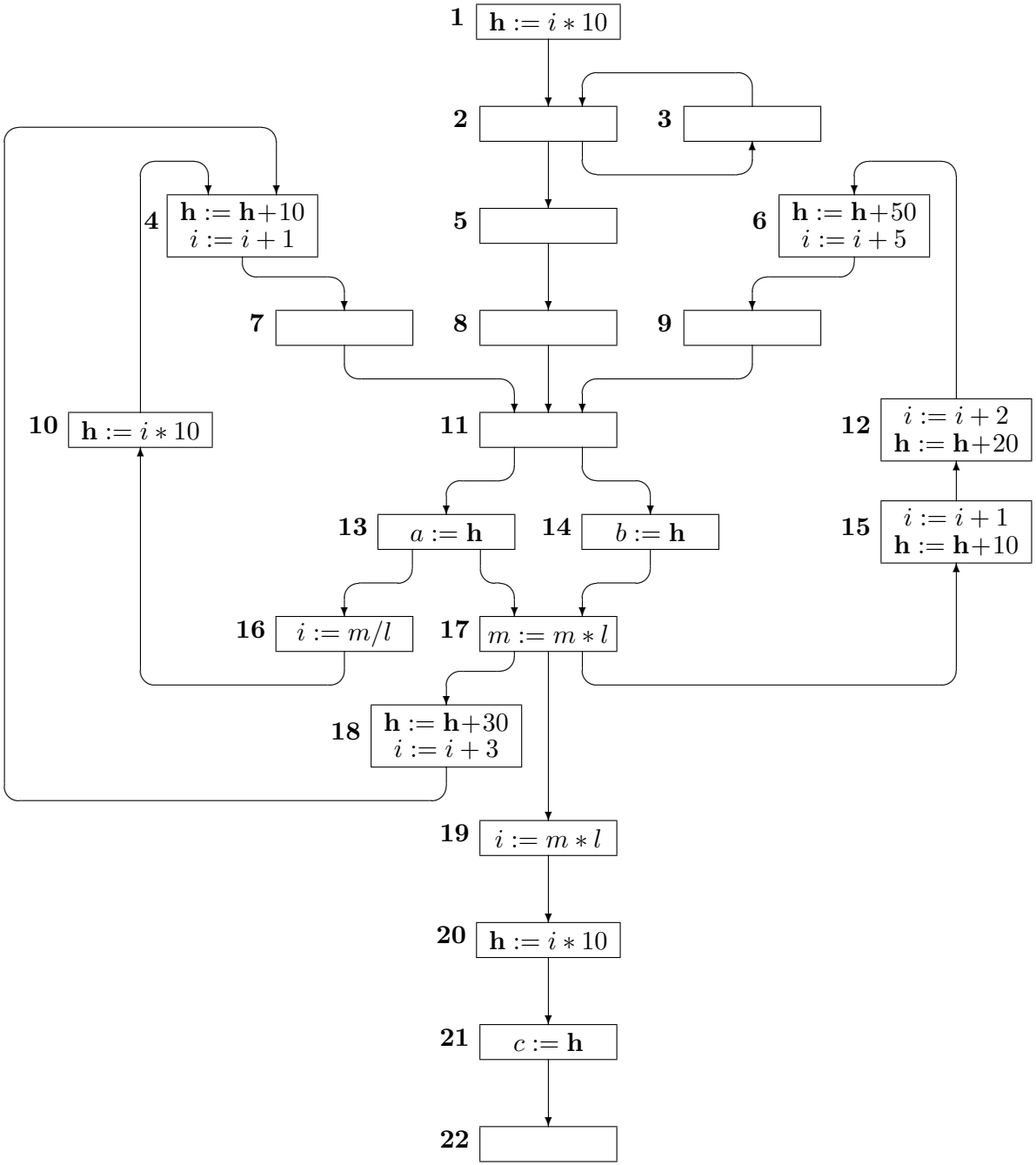
The *SSR*-Transformation:

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   2.1 $\mathtt{Ins}_{\mathtt{SSR}}$ the assignment $\mathbf{h} := v * c$
   2.2 $\mathtt{InsUpd}_{\mathtt{SSR}}$ the assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

Note: If both $\mathtt{Ins}_{\mathtt{SSR}}$ and $\mathtt{InsUpd}_{\mathtt{SSR}}$ hold, the initialization statement $\mathbf{h} := v * c$ must precede the update assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The Result of *SSR*



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Discussing the Effect of *SSR*

Shortcoming

- The multiplication-addition-deficiency

Remedy:

- Moving critical insertion points in the direction of the control flow to "earliest" non-critical ones.

Intuitively:

- A program point is critical if there is a $v * c$-free program path from this point to a modification of v

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

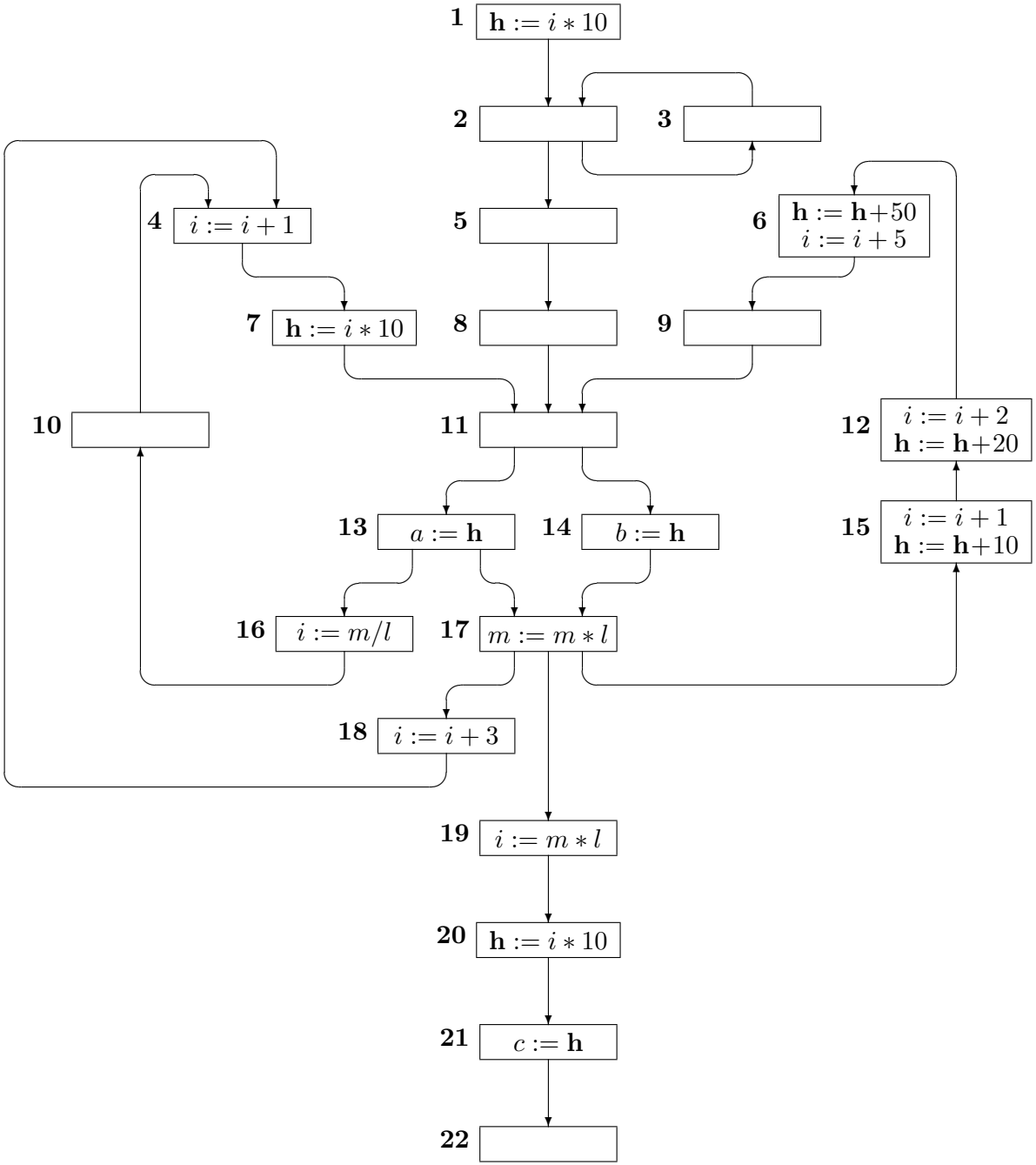
The 1st Refinement of *SSR*

The SSR_{FstRef}-Transformation:

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   2.1 $\mathtt{Ins}_{FstRef}$ the assignment $\mathbf{h} := v * c$
   2.2 $\mathtt{InsUpd}_{FstRef}$ the assignment $\mathbf{h} := \mathbf{h} + Eff(n)$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

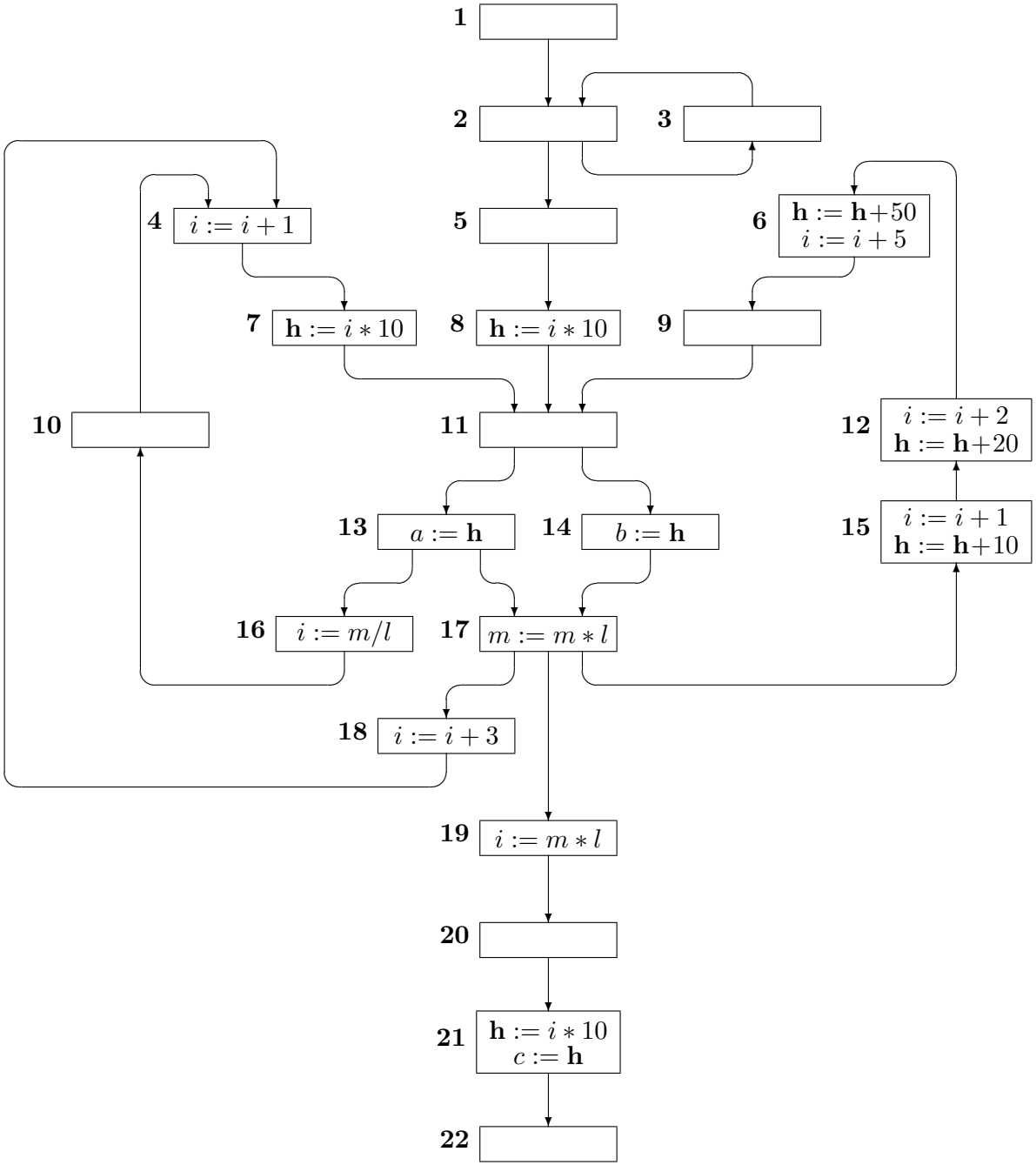
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Result of SSR_{FstRef}



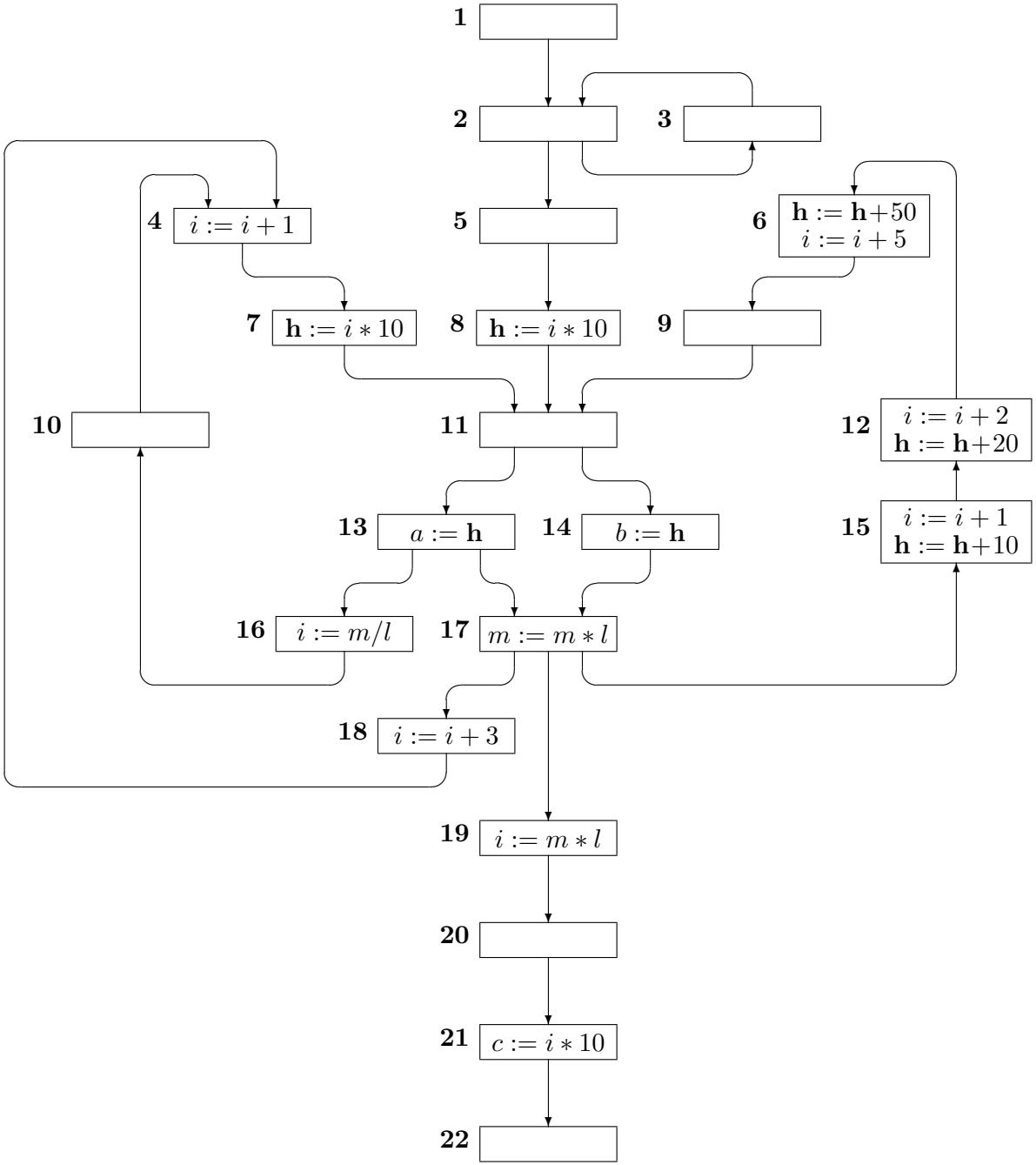
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Adding Laziness



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

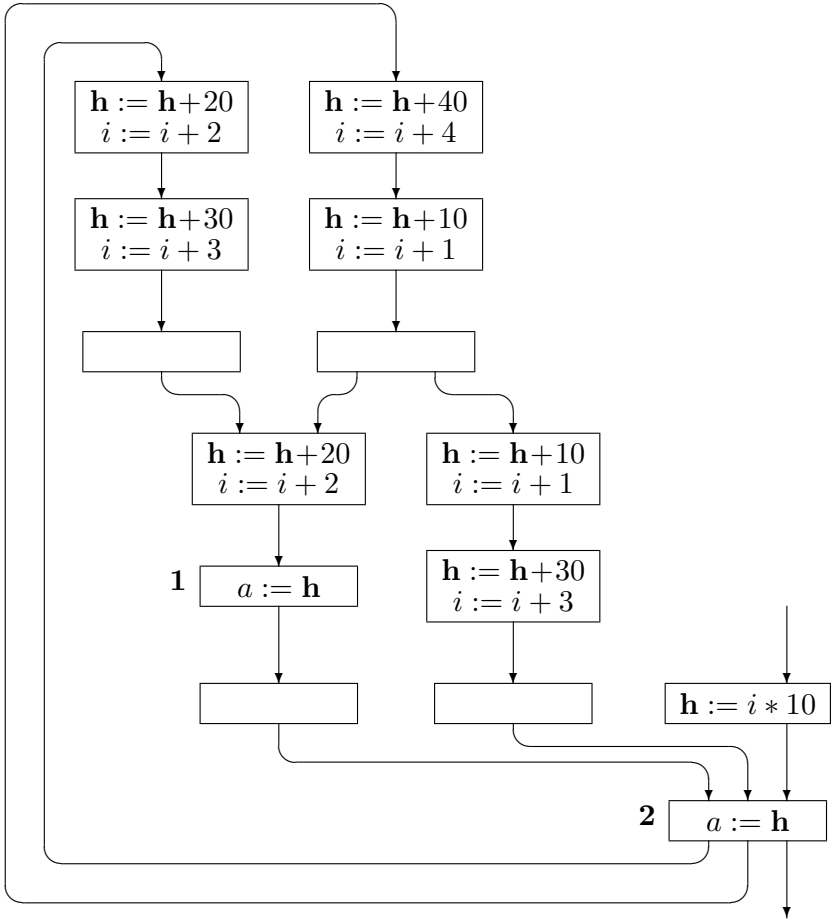
The Result of SSR_{SndRef}



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Multiple-Addition Deficiency

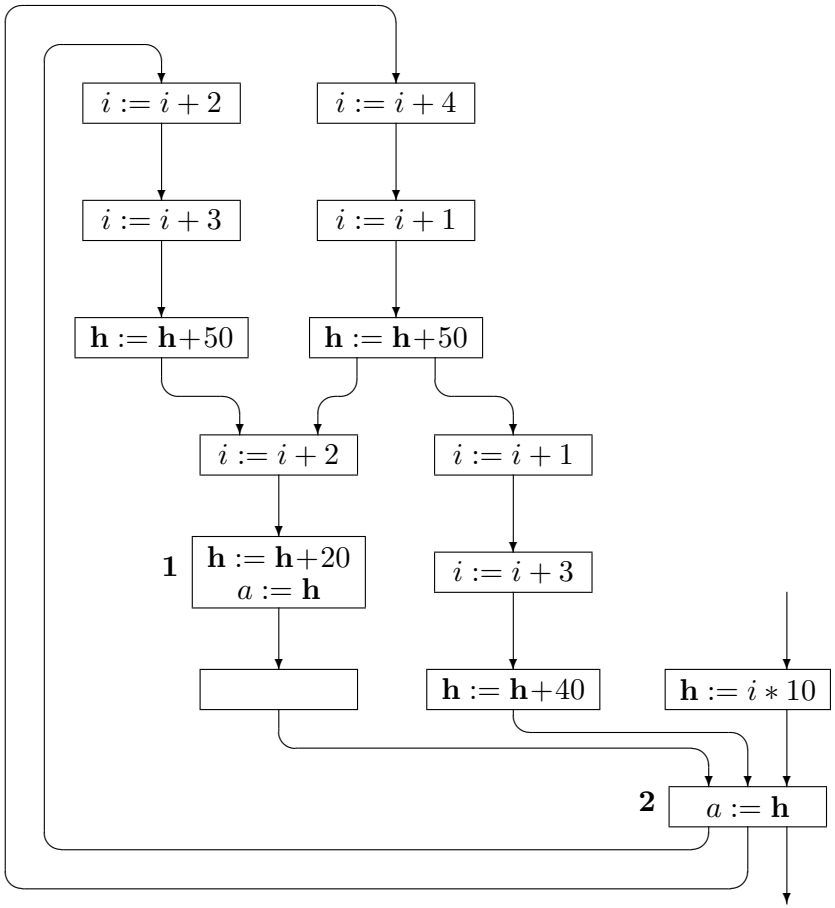
Illustration:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

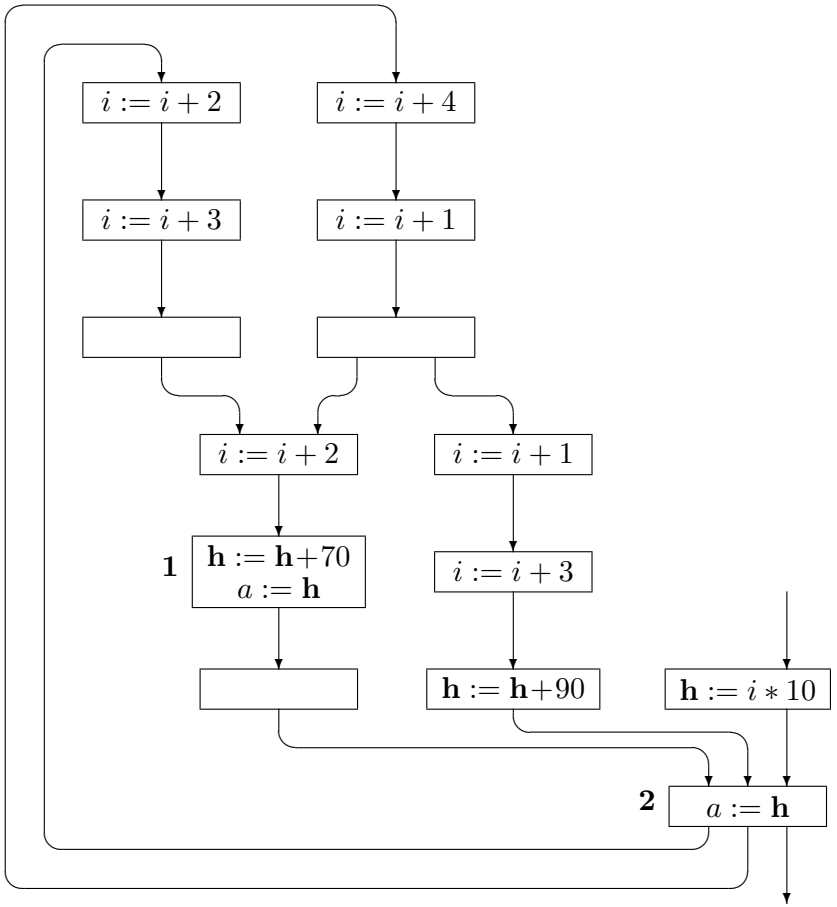
Overcoming the Multiple-Addition Deficiency

Accumulating the effect of *cure* assignments:



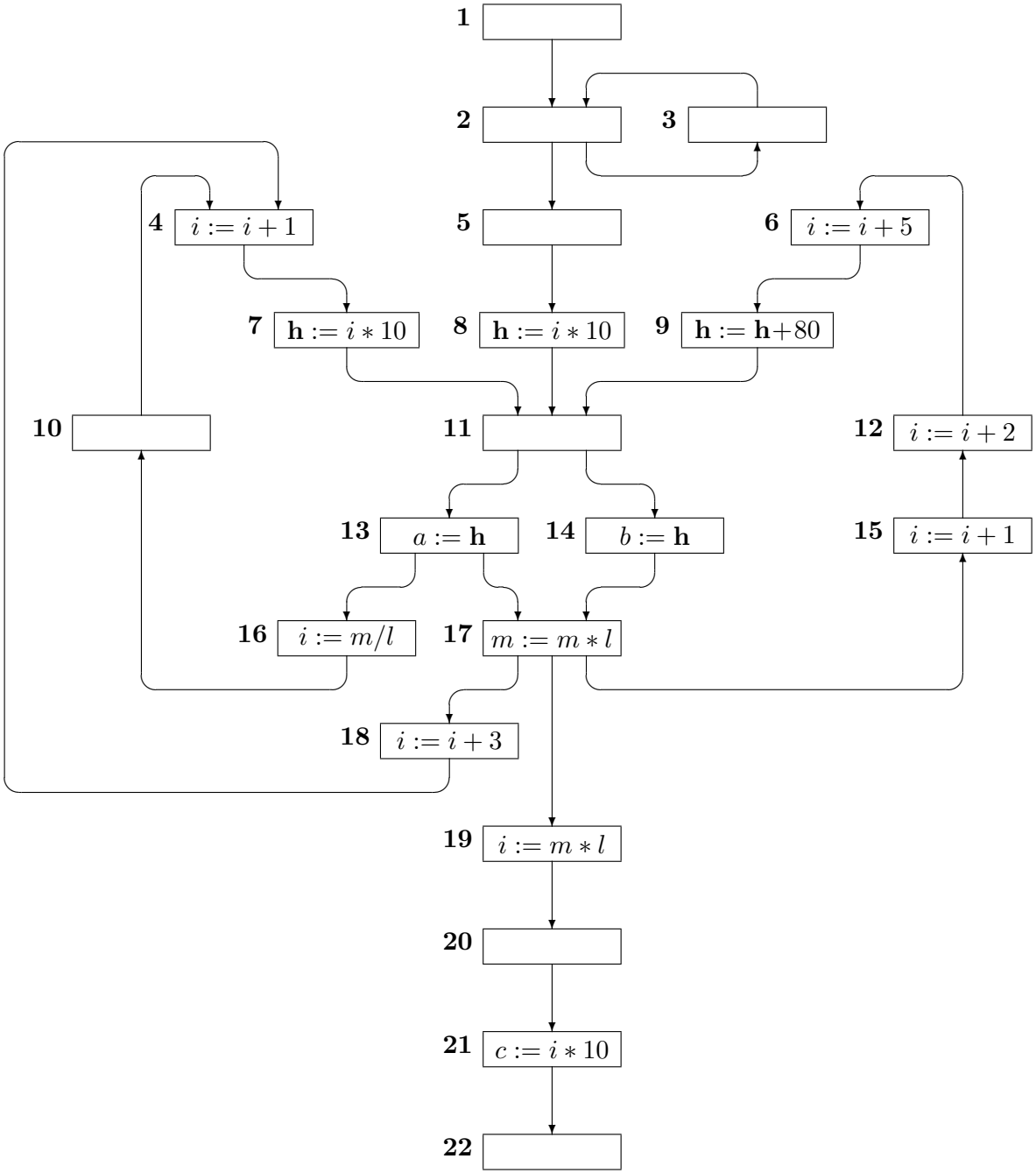
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Refined Accumulation of Cure Assignments



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The 3rd Refinement of *SSR*: *LSR*



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Homework

Assignment 5:

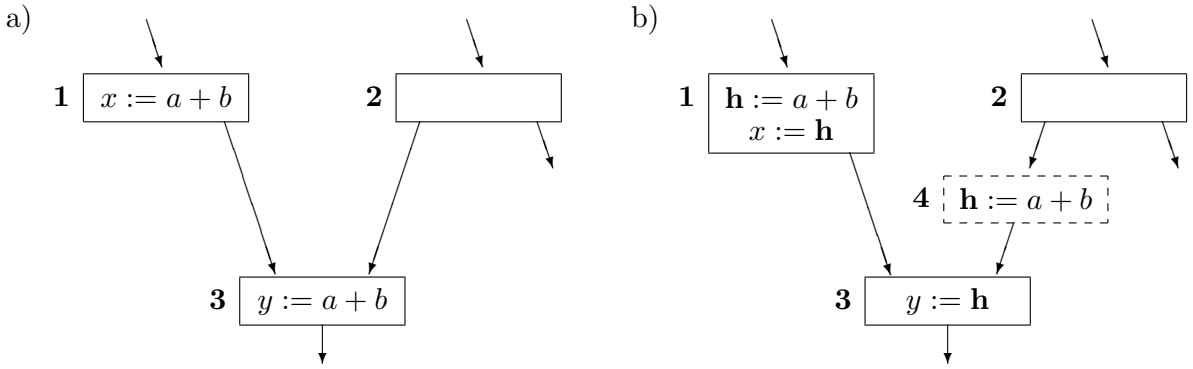
1. Specify the data flow analyses and transformations for
   - SSR
   - SSR_{FstRef} (overcoming the multiplication-addition deficiency)
   - SSR_{SndRef} (overcoming the register-pressure deficiency)
   - $SSR_{ThdRef} = LSR$ (overcoming the multiple-addition deficiency)
2. implement them in PAG, and
3. validate them on the running example of this chapter (or an example coming close to it).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Critical Edges

Like for *BCM* and *LCM* critical edges need to be split in order to get the full power of

- Lazy Strength Reduction (LSR)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

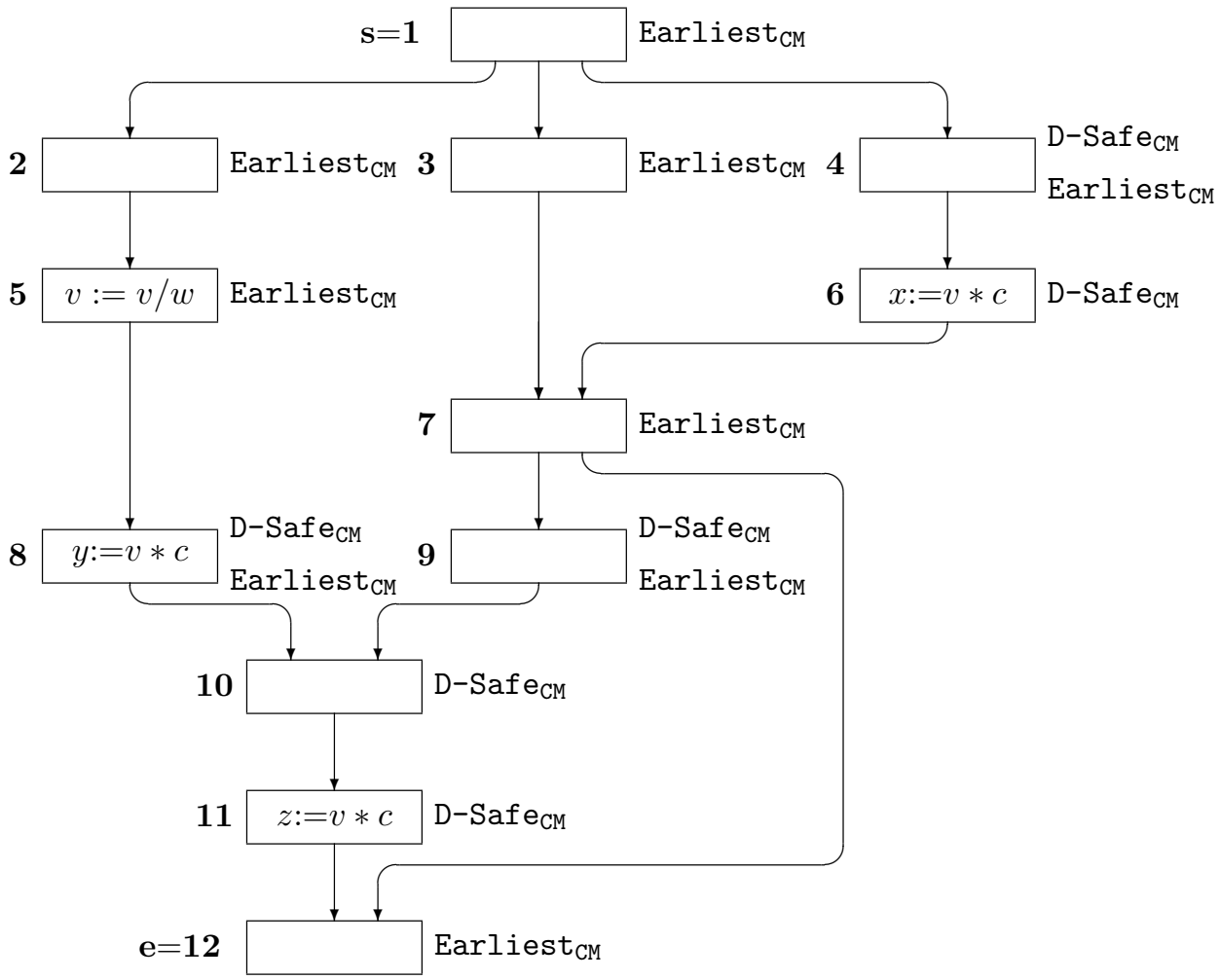
Summary of Predicate Values

...of the analyses of the LSR transformation:

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
Safe$_{\texttt{CM}}$	1	1	1	0	1	0	1	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0
Earliest$_{\texttt{CM}}$	1	0	0	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0
Insert$_{\texttt{CM}}$	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
Safe$_{\texttt{SR}}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	0
Earliest$_{\texttt{SR}}$	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
Insert$_{\texttt{SSR}}$	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
Critical	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0
Subst-Crit	0	0	0	1	0	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0
Insert$_{\texttt{FstRef}}$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Delay	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
Latest	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Isolated	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1
Update$_{\texttt{SndRef}}$	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	1	0
Insert$_{\texttt{SndRef}}$	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Accumulating	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1	0
Insert$_{\texttt{LSR}}$	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
InsUpd$_{\texttt{LSR}}$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Delete$_{\texttt{LSR}}$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Illustrating Down-Safety and Earliestness



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 11 (1)

- F. 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)
- D. M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. International Journal of Computer Mathematics 27:1-14,31-32, 1989.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Further Reading for Chapter 11 (2)

- D. 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, D. 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.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. Journal of Programming Languages 1(1):71-91, 1993.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 11 (2)

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 12

More on Code Motion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Chapter 12.1

Motivation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Motivation

Why is it rewarding to consider PRE?

Because PRE is

- General: A family of optimizations rather than a single optimization
- Well understood: Proven correct and optimal
- Relevant: Widely used in practice because of its power
- Truly classical: Looks back to a long history beginning with
  - Etienne Morel, Claude Renvoise. Global Optimization by Suppression of Partial Redundancies. Communications of the ACM 22(2):96-103, 1979.
  - Andrei P. Ershov. On Programming of Arithmetic Operations. Communications of the ACM 1(8):3-6, 1958.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Motivation (Cont'd)

Last but not least, PRE is

- Challenging: Conceptually simple but exhibits a variety of thought provoking phenomenons

Some of these challenges we are going to sketch next.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Code Motion Reconsidered

Traditionally:

- Code (C) means expressions
- Motion (M) means hoisting

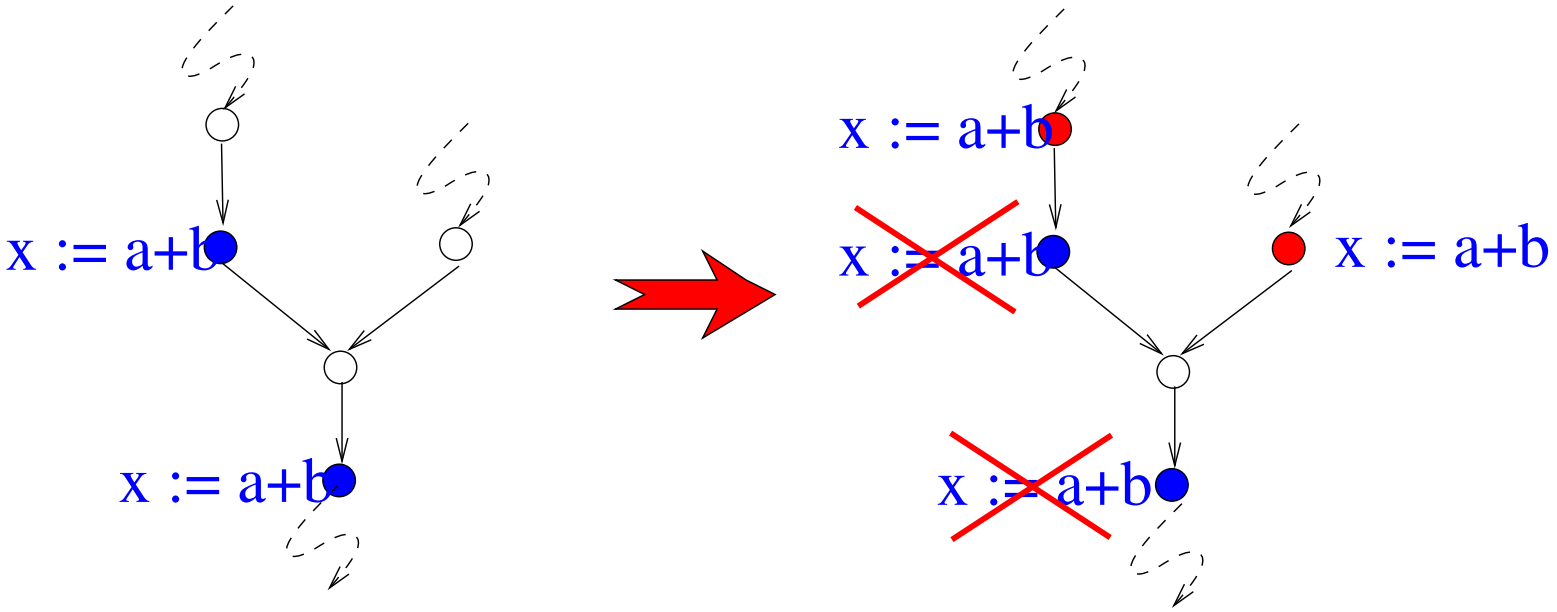
But:

- CM is more than hoisting of expressions and PR(E)E!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Obviously

...assignments are code, too.

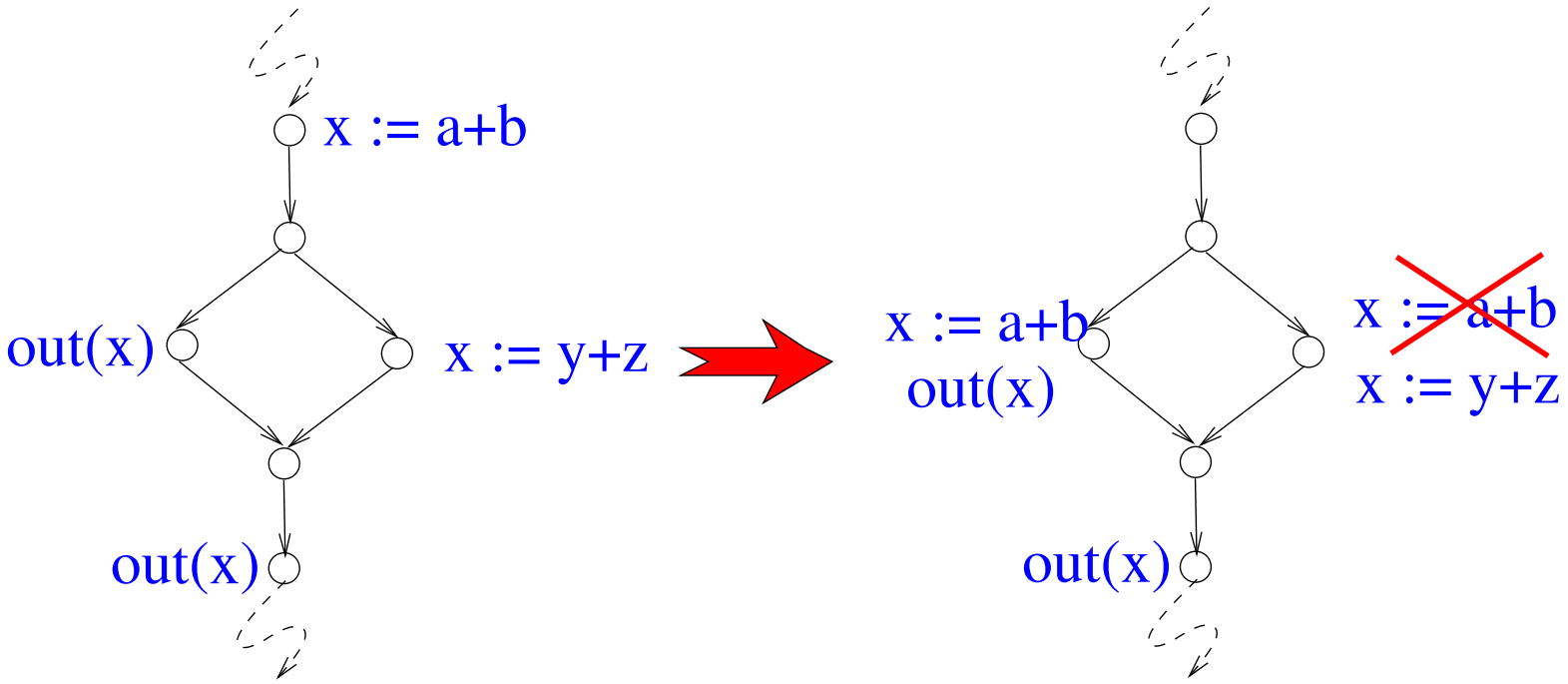


- Here, CM means eliminating partially redundant assignments (PRAE)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Differently from expressions...

...assignments can also be sunk.



- Here, CM means eliminating partially dead code (PDCE)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Design Space of CM-Algorithms (1)

This results in the following design space of CM-algorithms:

Generally:

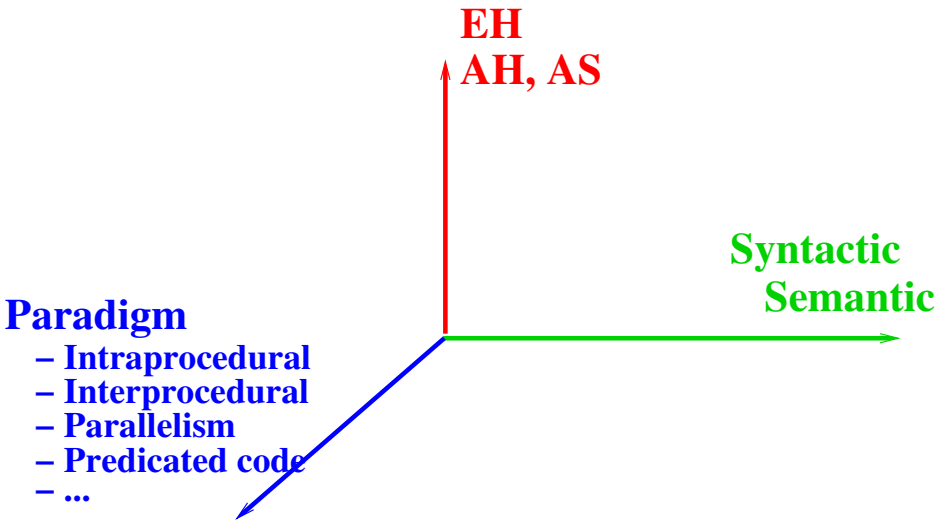
- Code means expressions/assignments
- Motion means hoisting/sinking

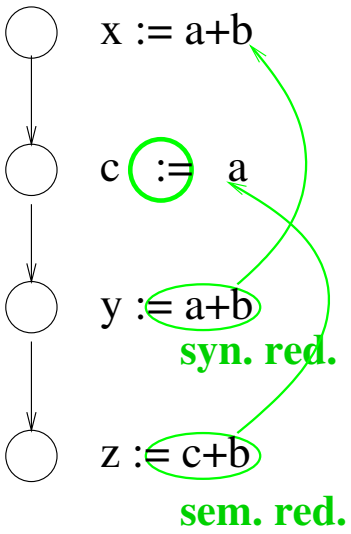
Code / Motion	Hoisting	Sinking
Expressions	EH	·/·
Assignments	AH	AS

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Design Space of CM-Algorithms (2)

Adding further dimensions to the design space of CM-algorithms:



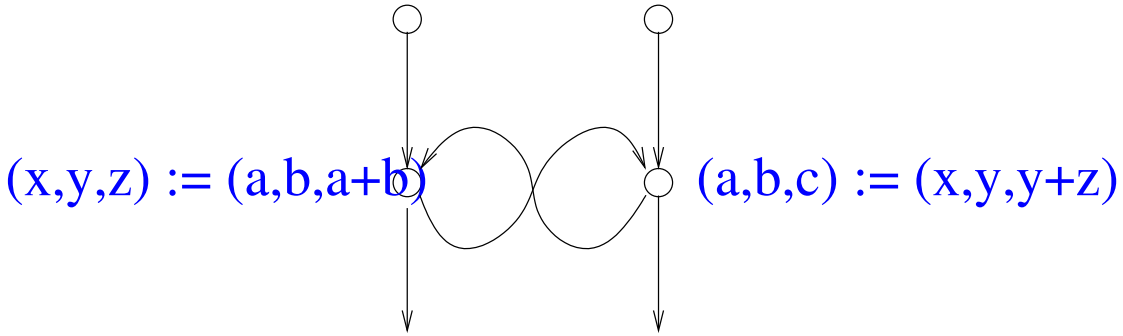


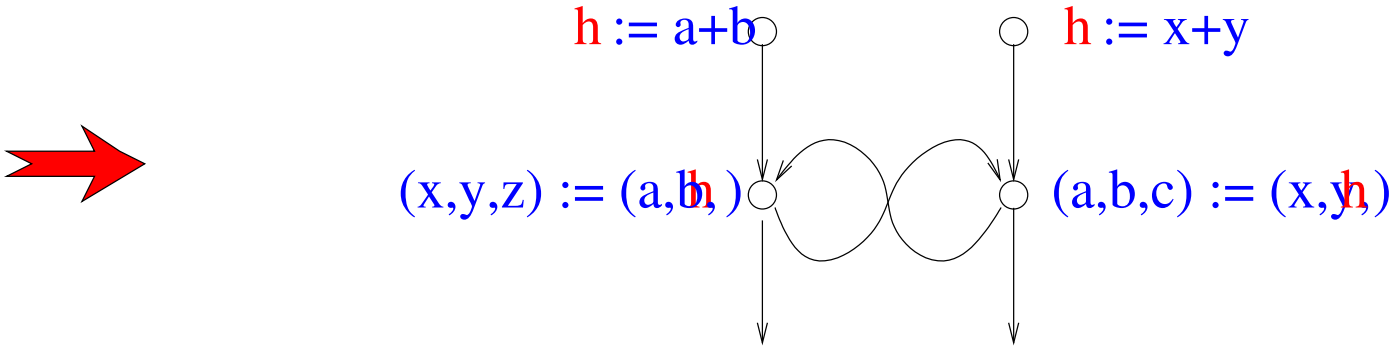
Introducing semantics... !

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Semantic Code Motion

...enables more powerful optimizations!





(Example from B. Steffen, TAPSOFT'87)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

What is the Impact on Optimality?

Optimality statements are quite sensitive towards setting changes!

Three examples shall provide evidence for this:

- Code motion vs. code placement
- Interdependencies of elementary transformations
- Paradigm dependencies

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 12.1

- Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Proceedings of the 2nd Joint Conference on Theory and Prac- tice 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

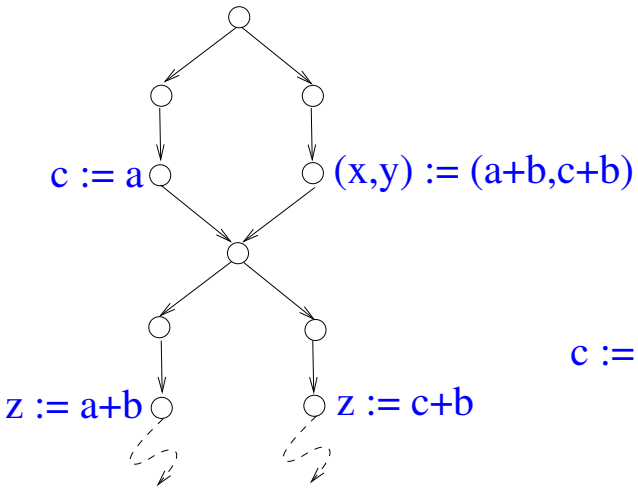
Chapter 12.2

Code Motion vs. Code Placement

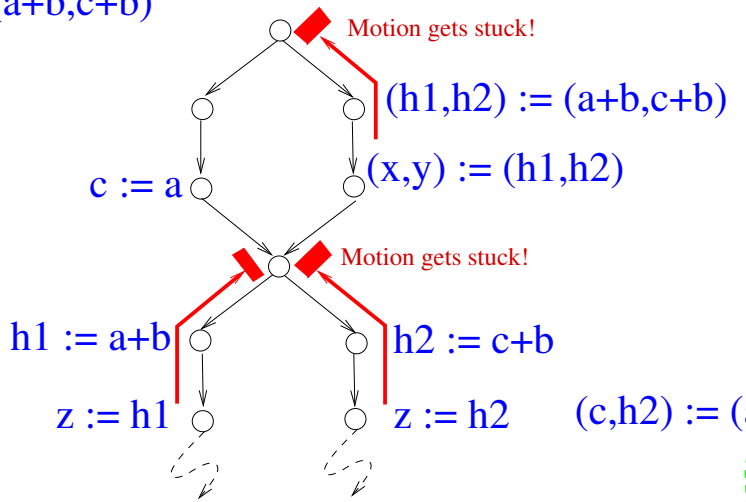
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Code Motion (CM) vs. Code Placement (CP)

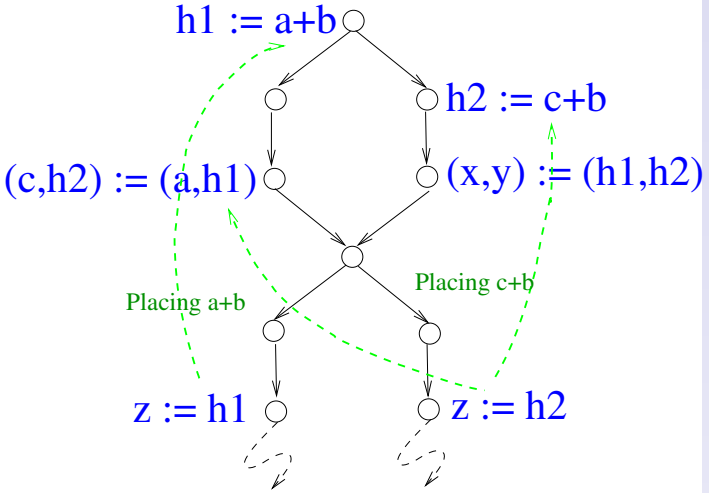
CM and CP are no synonyms!



Original Program



After Sem. Code Motion

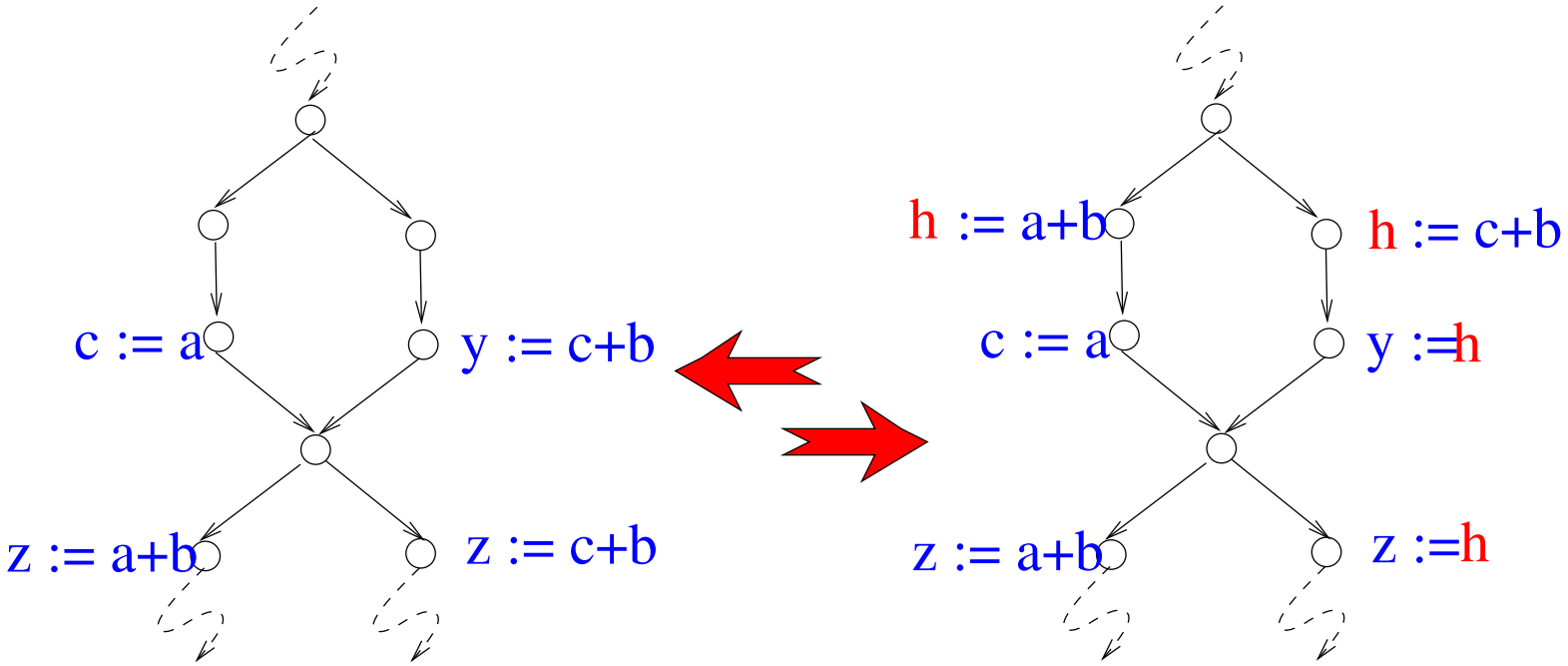


After Sem. Code Placement

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Even worse

Optimality is lost!

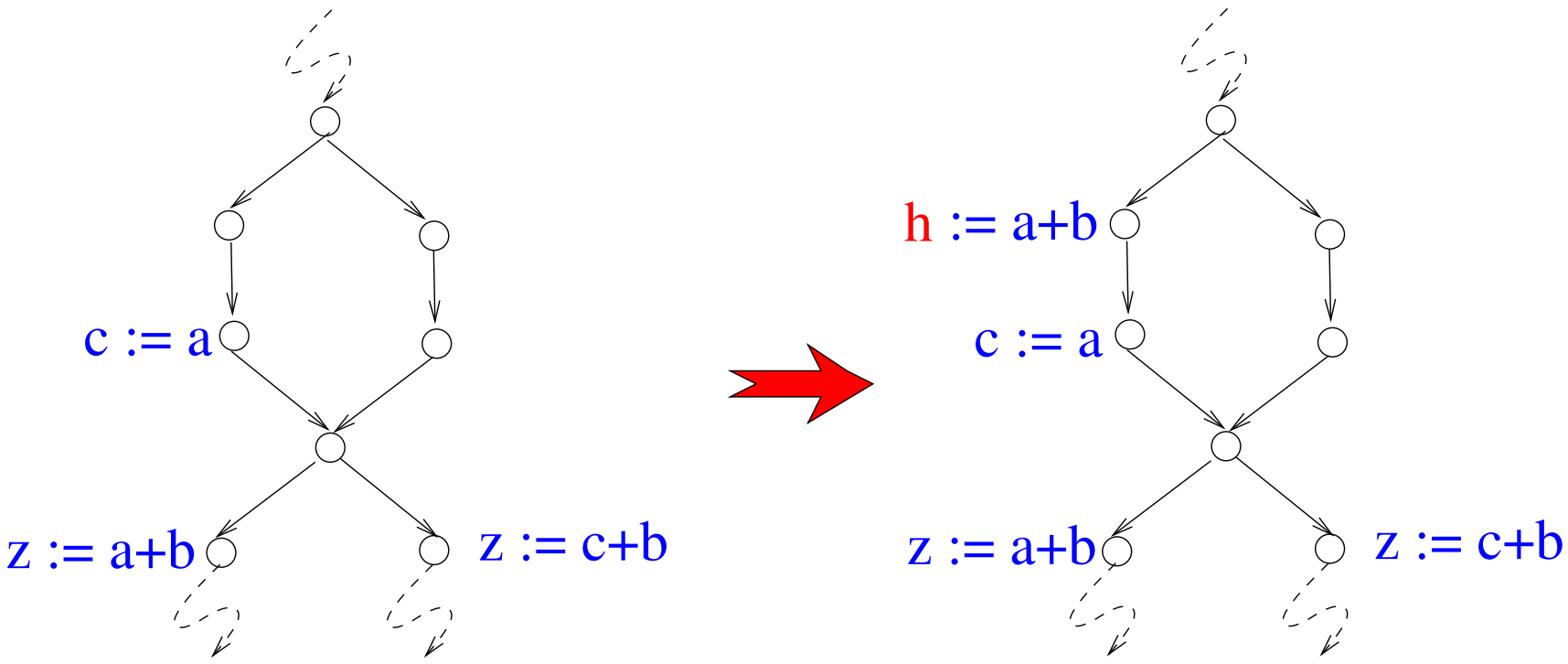


Incomparable!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Even more worse

The performance can be impaired, when applied naively!



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 12.2

- 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 Pro- ceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

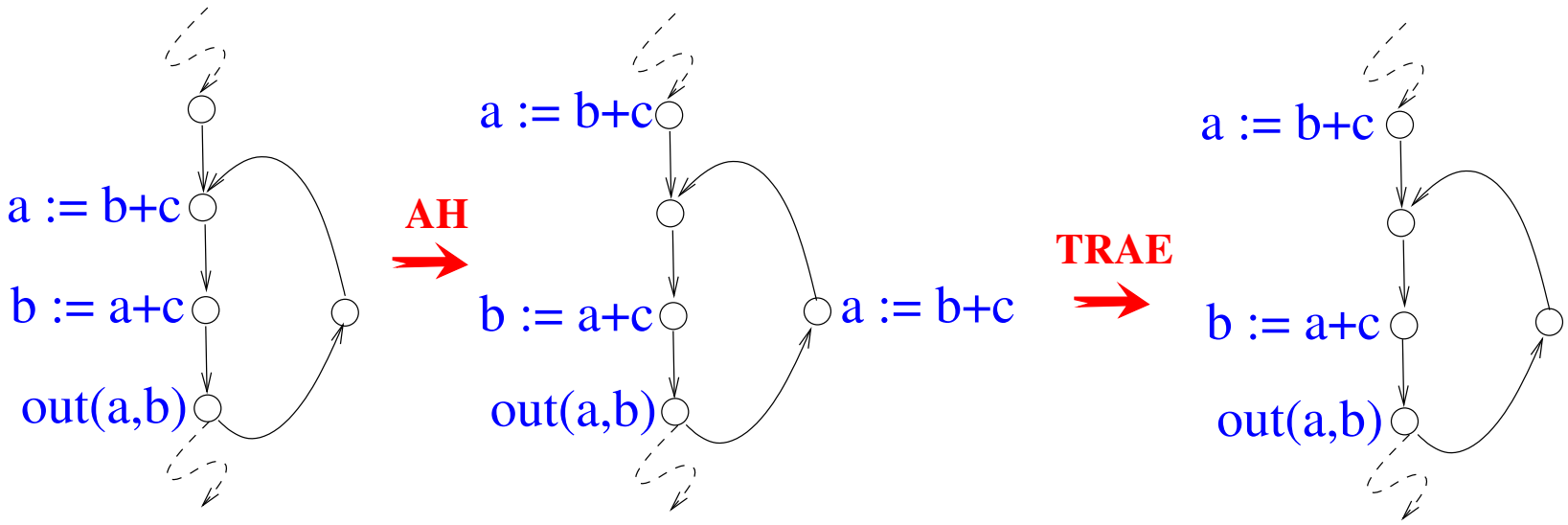
Chapter 12.3

Interactions of Elementary Transformations

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Assignment Hoisting (AH) plus Totally Redundant Assignment Elimination (TRAE)

...leads to Partially Redundant Assignment Elimination (PRAE):

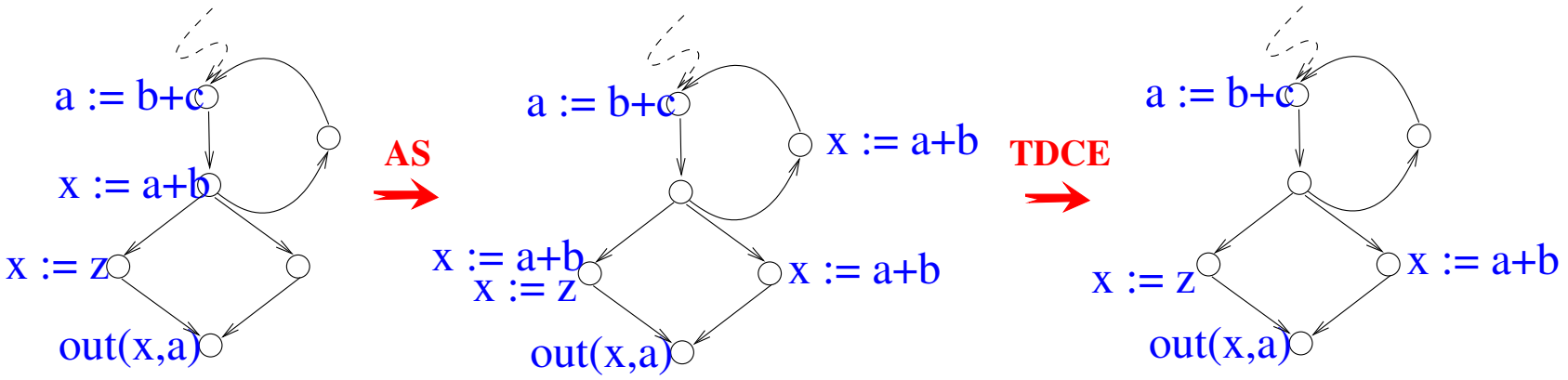


...2nd Order Effects!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Assignment Sinking (AS) plus Total Dead-Code Elimination (TDCE)

...leads to Partial Dead-Code Elimination (PDCE):



...2nd Order Effects!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Conceptually

...we can understand PREE, PRAE, and PDCE as follows:

- $\mathsf{PREE} = \mathsf{EH} \; ; \; \mathsf{TREE}$
- $\mathsf{PRAE} = (\mathsf{AH} + \mathsf{TRAE})^*$
- $\mathsf{PDCE} = (\mathsf{AS} + \mathsf{TDCE})^*$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Optimality Results for PREE

Theorem (12.2.1, Optimality)

1. *The BCM transformation yields computationally optimal results.*
2. *The LCM transformation yields computationally and lifetime optimal results.*
3. *The SpCM transformation yields optimal results wrt a given prioritization of the goals of redundancy avoidance, register pressure, and code size.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Optimality Results for (Pure) PRAE/PDCE

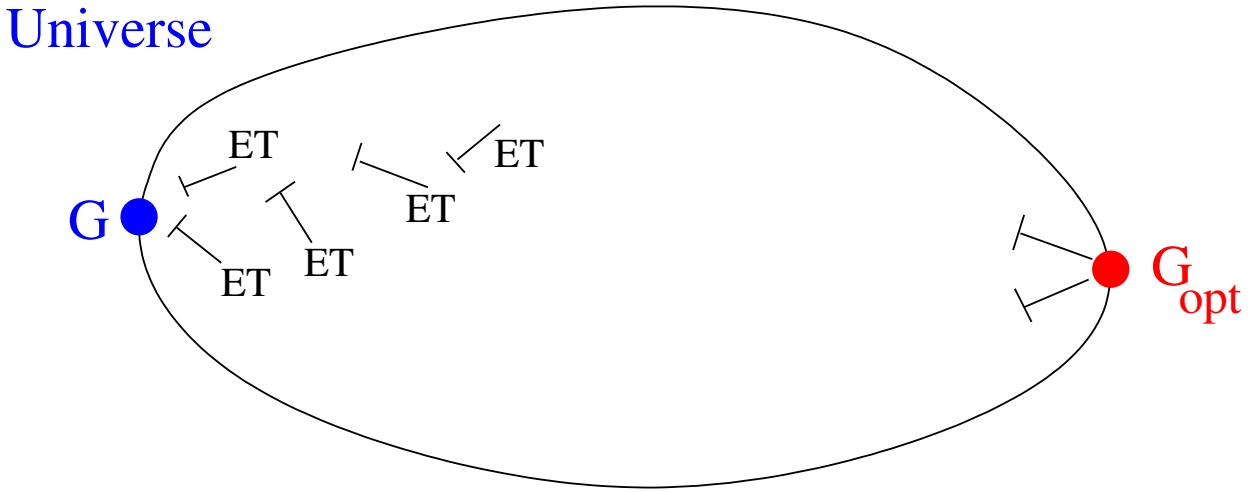
Deriving relation $\vdash$...

- PRAE... $G \vdash_{AH,TRAE} G'$ (ET={AH,TRAE})
- PDCE... $G \vdash_{AS,TDCE} G'$ (ET={AS,TDCE})

We can prove:

Theorem (12.2.2, Optimality)

For PRAE and PDCE the deriving relation $\vdash_{ET}$ is confluent and terminiating.



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

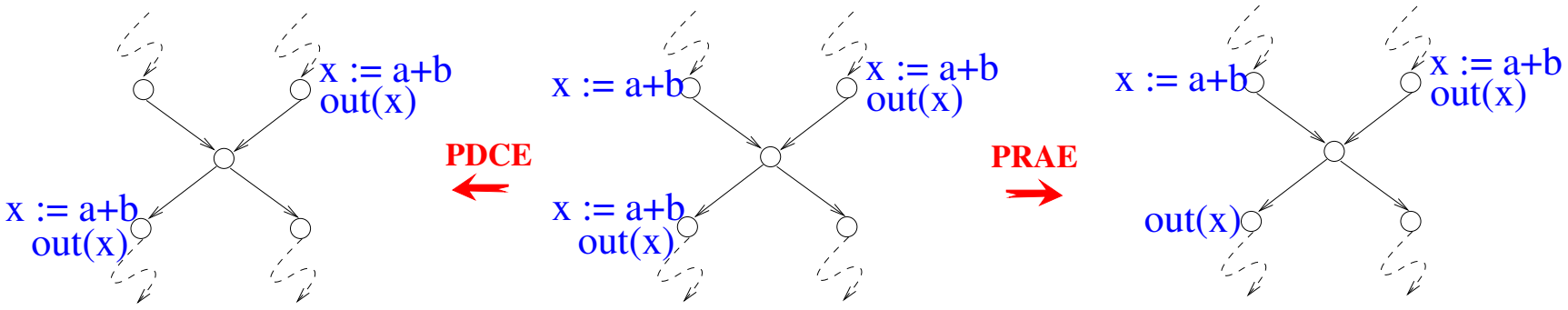
Now

...extend and amalgate PRAE and PDCE to Assignment Placement (AP):

- AP = (AH + TRAE + AS + TDCE)*

...AP should be more powerful than PRAE and PDCE alone!

Indeed, it is but:

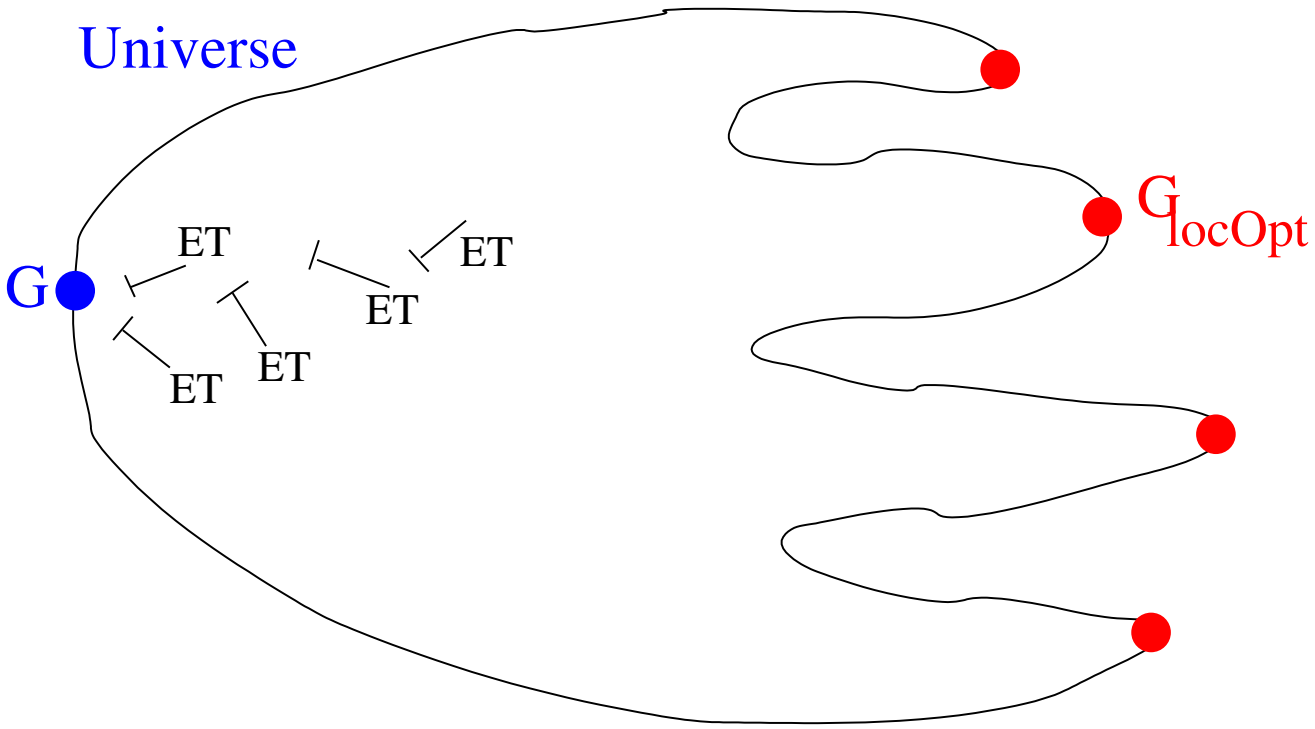


The resulting two programs are incomparable.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Confluence

...and hence (global) optimality is lost!

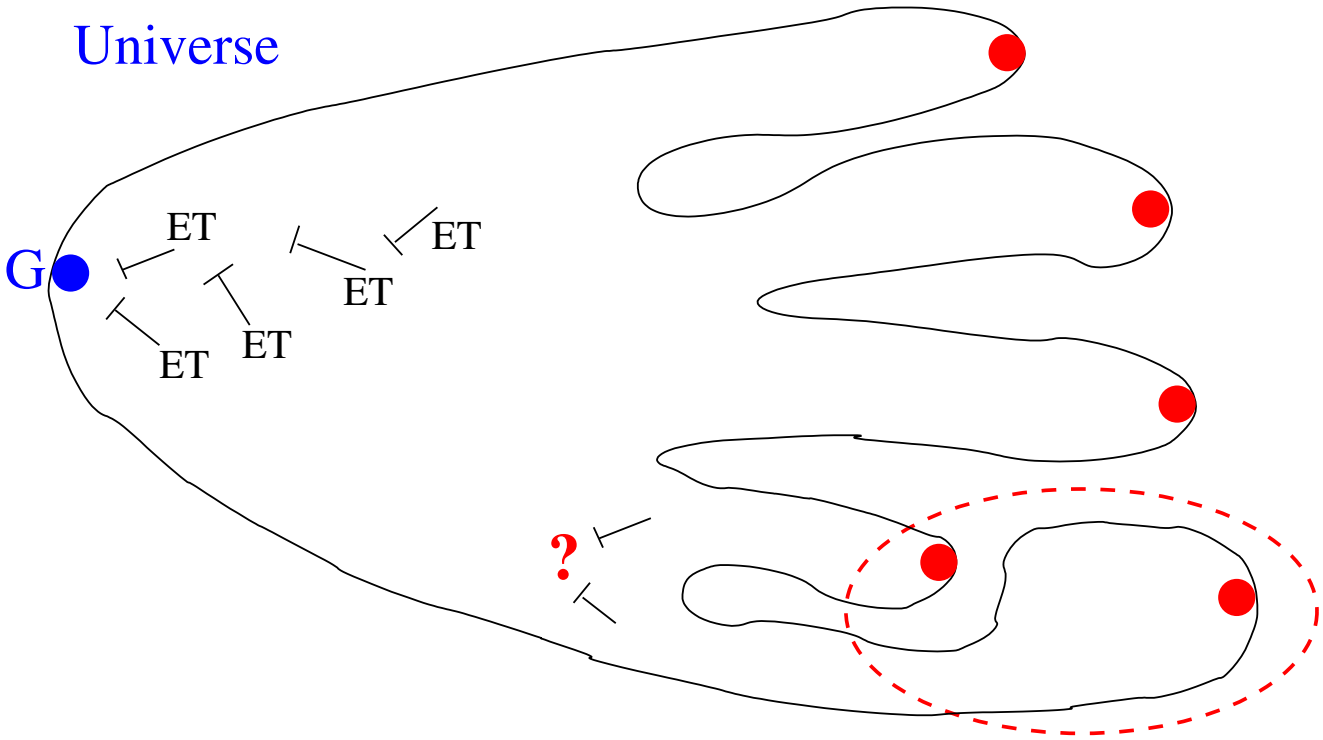


Fortunately, we retain local optimality!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

However

...there are settings, where we end up w/ universes like the following:



Here, even local optimality is lost!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 12.3

- 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.

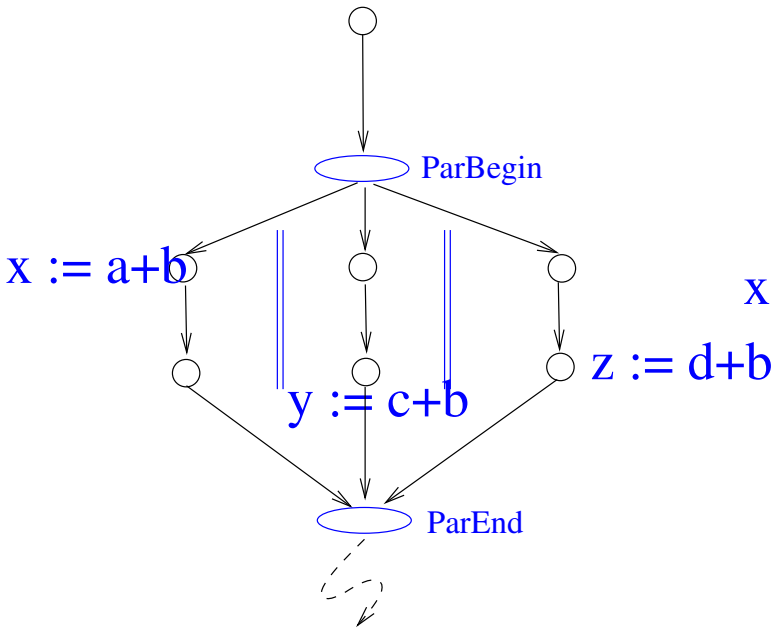
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Chapter 12.4

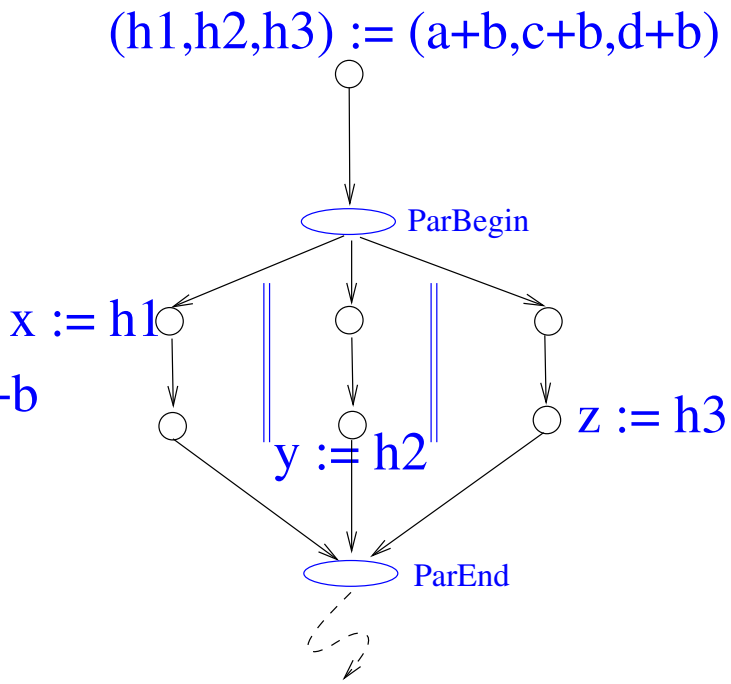
Paradigm Impacts

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Adding Parallelism



Original Program



After Earliestness Transformatio

...a naive transfer of the "place computations as early as possible" transformation strategy leads here to an essentially sequential program!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Adding Procedures

Similar phenomena are encountered when naively applying successful transformation strategies from the intraprocedural to the interprocedural setting.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 12.4

- Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998. (Chapter 10, Interprocedural Code Motion: The Transformations)
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Chapter 12.5

Further Code Motion Transformations

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Suggested Reading (1)

- Syntactic PRE
  - Knoop, J., Rüthing, O., and Steffen, B. 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 (2004), 460 - 461 & 462-472.
  - Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. ACM Transactions on Programming Languages and Systems 16, 4 (1994), 1117 - 1155.
  - Rüthing, O., Knoop, J., and Steffen, B. Sparse code motion. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000) (Boston, MA, Jan. 19 - 21, 2000), ACM New York, (2000), 170 - 183.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Suggested Reading (2)

- Eliminating partially dead code
  - Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94) (Orlando, FL, USA, June 20 - 24, 1994), ACM SIGPLAN Notices 29, 6 (1994), 147 - 158.
- Eliminating partially redundant assignments
  - Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95) (La Jolla, CA, USA, June 18 - 21, 1995), ACM SIGPLAN Notices 30, 6 (1995), 233 - 245.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Suggested Reading (3)

- BB-graphs vs. SI-graphs
  - Knoop, J., Koschützki, D., and Steffen, B. Basic-block graphs: Living dinosaurs? In Proceedings of the 7th International Conference on Compiler Construction (CC'98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1383 (1998), 65 - 79.
- Moving vs. placing code
  - Knoop, J., Rüthing, O., and Steffen, B. Code motion and code placement: Just synonyms? In Proceedings of the 7th European Symposium On Programming (ESOP'98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1381 (1998), 154 - 169.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Suggested Reading (4)

- Speculative vs. classical PRE
  - Scholz, B., Horspool, N. and Knoop, J. Optimizing for space and time usage with speculative partial redundancy elimination. In Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004) (Washington, DC, June 11 - 13, 2004), ACM SIGPLAN Notices 39, 7 (2004), 221 -230.
  - Xue, J., Knoop, J. A fresh look at PRE as a maximum flow problem. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006) (Vienna, Austria, March 25 - April 2, 2006), Springer-Verlag, Heidelberg, LNCS 3923 (2006), 139 - 154.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Suggested Reading (5)

- Further techniques
  - Geser, A., Knoop, J., Lüttgen, G., Rüthing, O., and Steffen, B. Non-monotone fixpoint iterations to resolve second order effects. In Proceedings of the 6th International Conference on Compiler Construction (CC'96) (Linköping, Sweden, April 24 - 26, 1996), Springer-V., Heidelberg, LNCS 1060 (1996), 106 - 120.
  - Knoop, J., and Mehofer, E. Optimal distribution assignment placement. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97) (Passau, Germany, August 26 - 29, 1997), Springer-V., Heidelberg, LNCS 1300 (1997), 364 - 373.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 12 (1)

- B. Alpern, Mark N. Wegman, F. Ken Zadeck. *Detecting Equality of Variables in Programs*. In Proceedings of POPL'88, 1-11, 1988.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 12 (2)

- 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. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998. (Chapter 10.1, Essential Differences to the Intraprocedural Setting)
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Expansion- based Removal of Semantic Partial Redundancies*. In Pro- ceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Further Reading for Chapter 12 (3)

- 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.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Part III

Interprocedural Data Flow Analysis

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Outline

We consider:

- The Context Information Approach (cf. Chapter 13)
  - Call Strings
  - Assumption Sets Strings
- The Functional Approach (cf. Chapter 14)
  - The Base Setting
    Adding Procedures but no parameters and local variables
  - The General Setting
    Adding value parameters and local variables
  - Extensions
    Adding reference parameters and procedural parameters

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
12.1
12.2
12.3
12.4
12.5
Chap. 13
Chap. 14
Chap. 15

Chapter 13

The Context Information Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Outline

From intraprocedural to interprocedural data-flow analysis...

To this end we extend our programming language WHILE by introducing programs with

- top-level declarations of global mutually recursive procedures and
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Syntax

Extended WHILE-Language WHILE$_\pi$:

$$
\begin{array}{lcl}
P_\star & ::= & \text{begin } D_\star \; S_\star \text{ end} \\
D & ::= & D; D \;\mid\; \text{proc } p(\text{val } x;\ \text{res } y) \text{ is}^{\ell_n} \; S \text{ end}^{\ell_x} \\
S & ::= & \ldots \;\mid\; [\text{call } p(a, z)]_{\ell_r}^{\ell_c}
\end{array}
$$

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Assumptions

We assume that

- WHILE_π 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.
- Programs are uniquely labelled.
- There are no procedures of the same name.
- Only procedures may be called by a program that have been declared in it.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

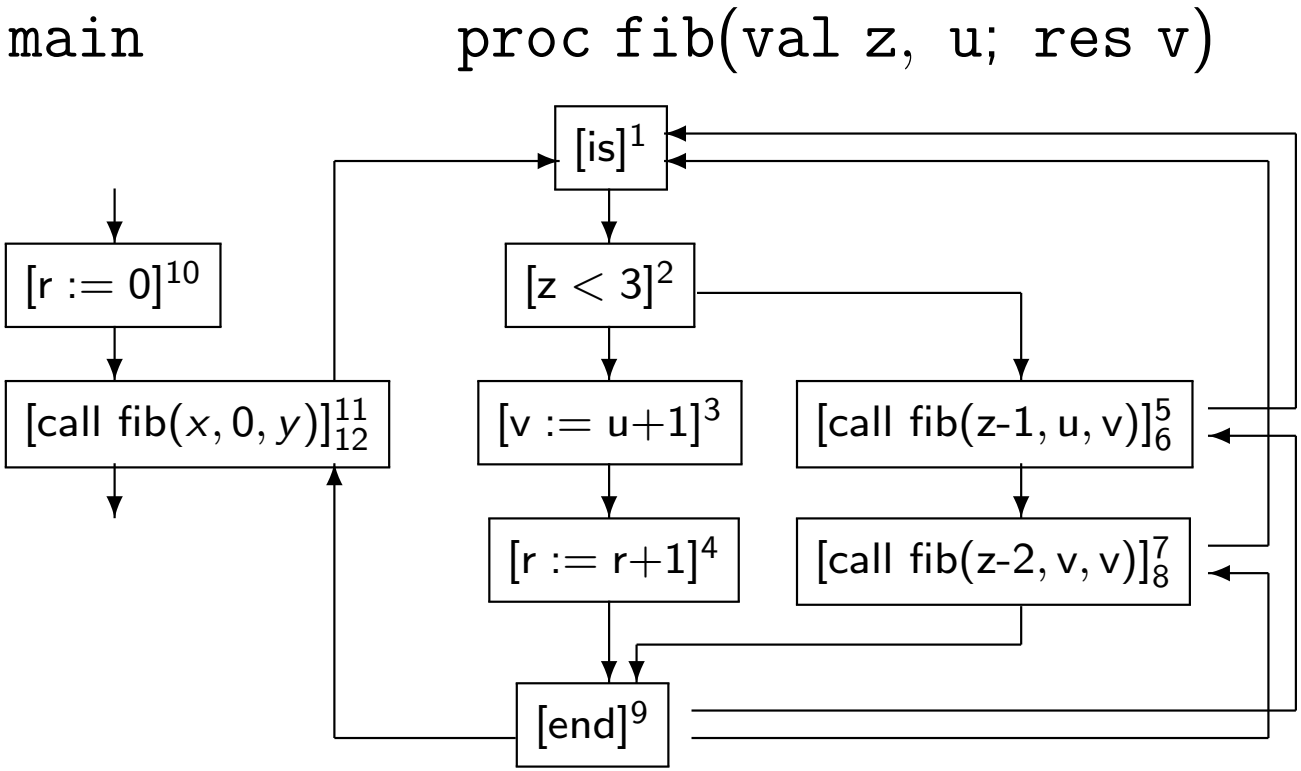
Example

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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The Flow Graph of Procedure `proc fib`



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Notions and Notations for Flow Graphs (1)

...for procedure calls and procedure declarations:

	$[\text{call } p(a,z)]^{\ell_c}_{\ell_r}$	$\text{proc } p(\text{val } x;\ \text{res } y)\ \text{is}^{\ell_n}\ S\ \text{end}^{\ell_x}$
init	ℓ_c	ℓ_n
final	$\{\ell_r\}$	$\{\ell_x\}$
blocks	$\{[\text{call } p(a,z)]^{\ell_c}_{\ell_r}\}$	$\{\text{is}^{\ell_n}\} \cup \text{blocks}(S) \cup \{\text{end}^{\ell_x}\}$
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 ';'.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Notions and Notations for Flow Graphs (2)

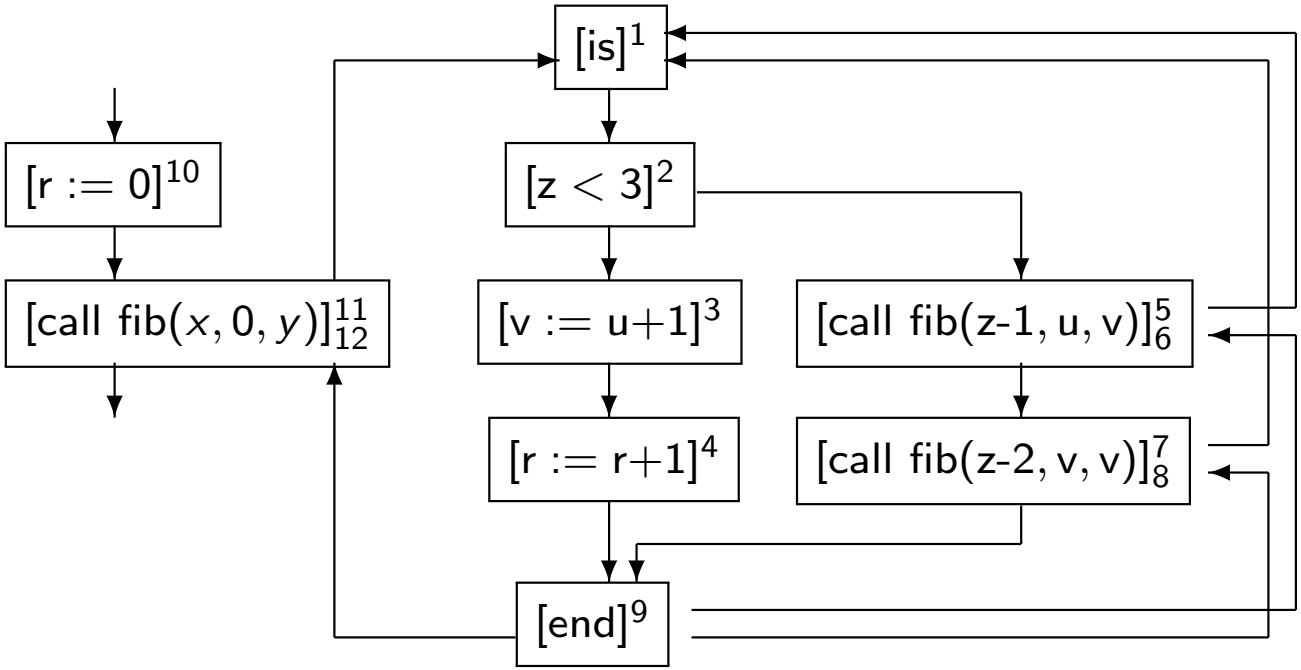
...for (whole) programs:

	$P_\star$
$\mathsf{init}_\star$	$\mathsf{init}(\mathsf{S}_\star)$
$\mathsf{final}_\star$	$\mathsf{final}(\mathsf{S}_\star)$
$\mathsf{blocks}_\star$	$\bigcup\{\mathsf{blocks}\ (p) \mid \mathsf{proc}\ p(\mathsf{val}\ x;\ \mathsf{res}\ y)\ \mathsf{is}^{\ell_n}\ S\ \mathsf{end}^{\ell_x}\ \textit{is in}\ D_\star\} \cup\ \mathsf{blocks}(S_\star)$
$\mathsf{labels}_\star$	$\bigcup\{\mathsf{labels}\ (p) \mid \mathsf{proc}\ p(\mathsf{val}\ x;\ \mathsf{res}\ y)\ \mathsf{is}^{\ell_n}\ S\ \mathsf{end}^{\ell_x}\ \textit{is in}\ D_\star\} \cup\ \mathsf{labels}(S_\star)$
$\mathsf{flow}_\star$	$\bigcup\{\mathsf{flow}\ (p) \mid \mathsf{proc}\ p(\mathsf{val}\ x;\ \mathsf{res}\ y)\ \mathsf{is}^{\ell_n}\ S\ \mathsf{end}^{\ell_x}\ \textit{is in}\ D_\star\} \cup\ \mathsf{flow}(S_\star)$
$\mathbf{Lab}_\star$	$\mathsf{labels}_\star$

$$\textit{inter-flow}_\star \quad = \quad \{(\ell_c, \ell_n, \ell_x, \ell_r) \mid P_\star\ \textit{contains}\ [\mathsf{call}\ p(a,z)]^{\ell_c}_{\ell_r}\ \textit{as well as}\ \mathsf{proc}\ p(\mathsf{val}\ x;\ \mathsf{res}\ y)\ \mathsf{is}^{\ell_n}\ S\ \mathsf{end}^{\ell_x}\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example



$$
\begin{aligned}
\mathit{flow}_\star &= \{(1,2),(2,3),(3,4),(4,9),\\
&\quad (2,5),(5;1),(9;6),(6,7),(7;1),(9;8),(8,9),\\
&\quad (11;1),(9;12),(10,11)\}\\
\mathit{inter\text{-}flow}_\star &= \{(11,1,9,12),(5,1,9,6),(7,1,9,8)\}
\end{aligned}
$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Metavariables for Forward/Backward Analyses

Forward Analyses:

- $F = flow_\star$
- $E = init_\star$
- $IF = inter\text{-}flow_\star$

Backward Analyses:

- $F = flow_\star^R$
- $E = final_\star$
- $IF = inter\text{-}flow_\star^R$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Towards Interprocedural DFA

New transfer functions dealing with interprocedural flow are required:

- For each procedure call $[\text{call } p(a, z)]^{\ell_c}_{\ell_r}$ 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 proc $p(\text{val } x;\ \text{res } y)$ is$^{\ell_n}$ S end$^{\ell_x}$ we require two transfer funcions
  - f_{l_n} and f_{l_x}

  corresponding to entering and exiting the procedure body.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Interprocedural DFA: Naive Formulation (1)

- Treat the three kinds of flow, (ℓ_1, ℓ_2), $(\ell_c; \ell_n)$, $(\ell_x; \ell_r)$ 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:

$$\begin{array}{lcl} A_\circ(\ell) & = & \bigsqcap\{A_\bullet(\ell') \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\} \sqcap \iota_E^\ell \\ A_\bullet(\ell) & = & f_\ell^A(A_\circ(\ell)) \end{array}$$

where

$$\iota_E^\ell =_{df} \begin{cases} \iota & \text{if } I \in E \\ \bot & \text{if } I \notin E \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Interprocedural DFA: Naive Formulation (2)

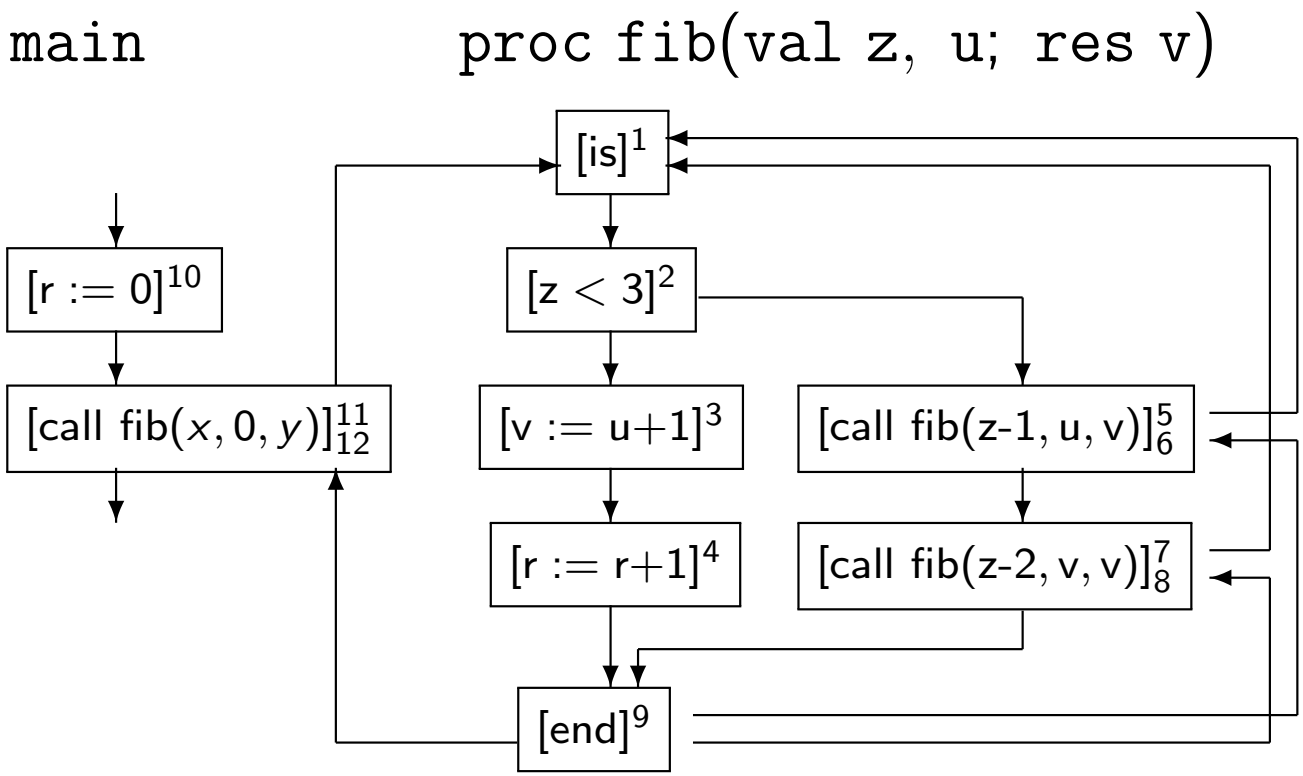
Given the previous assumptions we have:

- Both procedure calls $(\ell_c; \ell_n)$ and procedure returns $(\ell_x; \ell_r)$ are treated like "goto's".
- There is no mechanism for ensuring that information flowing along $(\ell_c; \ell_n)$ flows back along $(\ell_x; \ell_r)$ 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

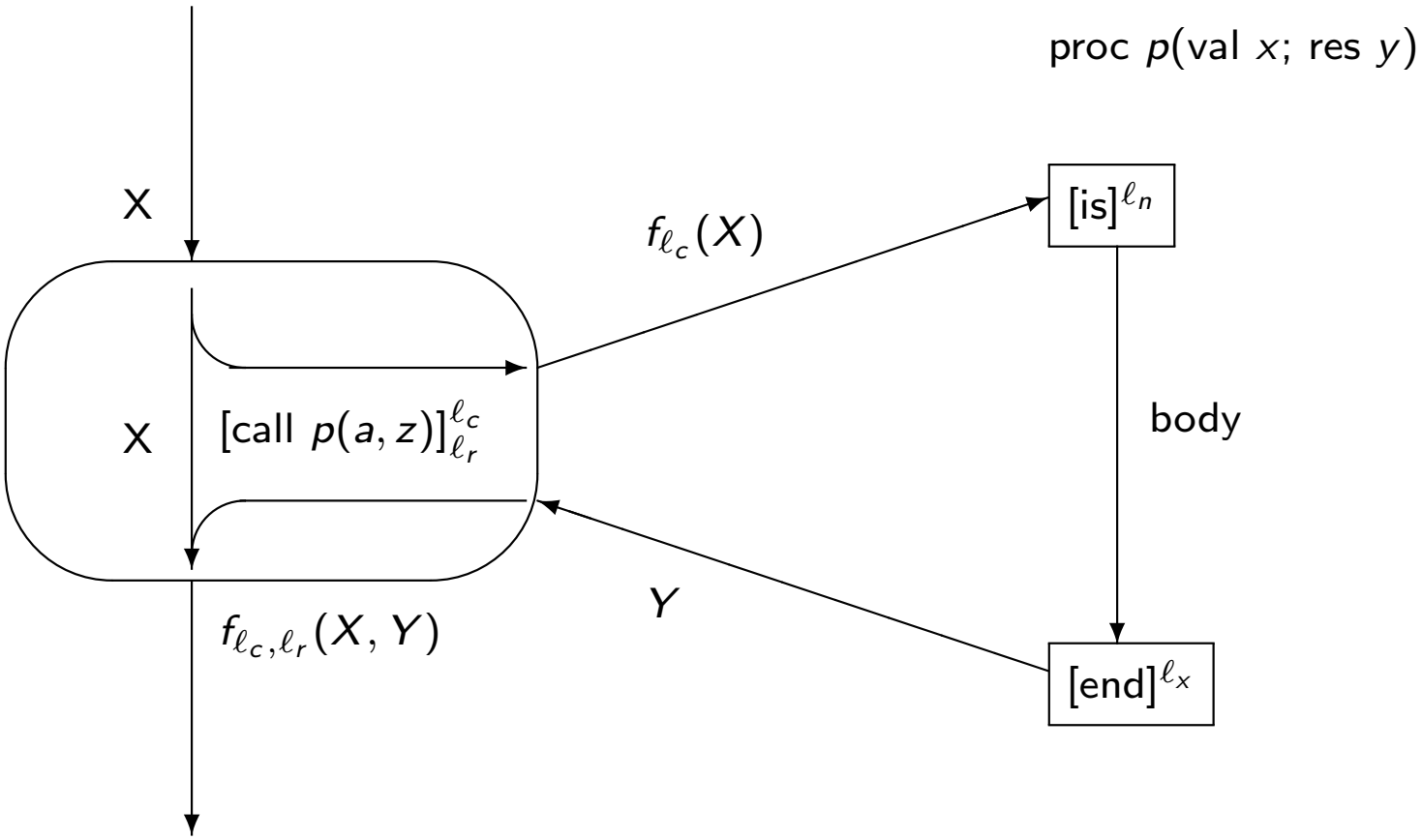
Interprocedural DFA: Naive Formulation (3)

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.



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Matching Procedure Entries and Exits



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Complete Paths

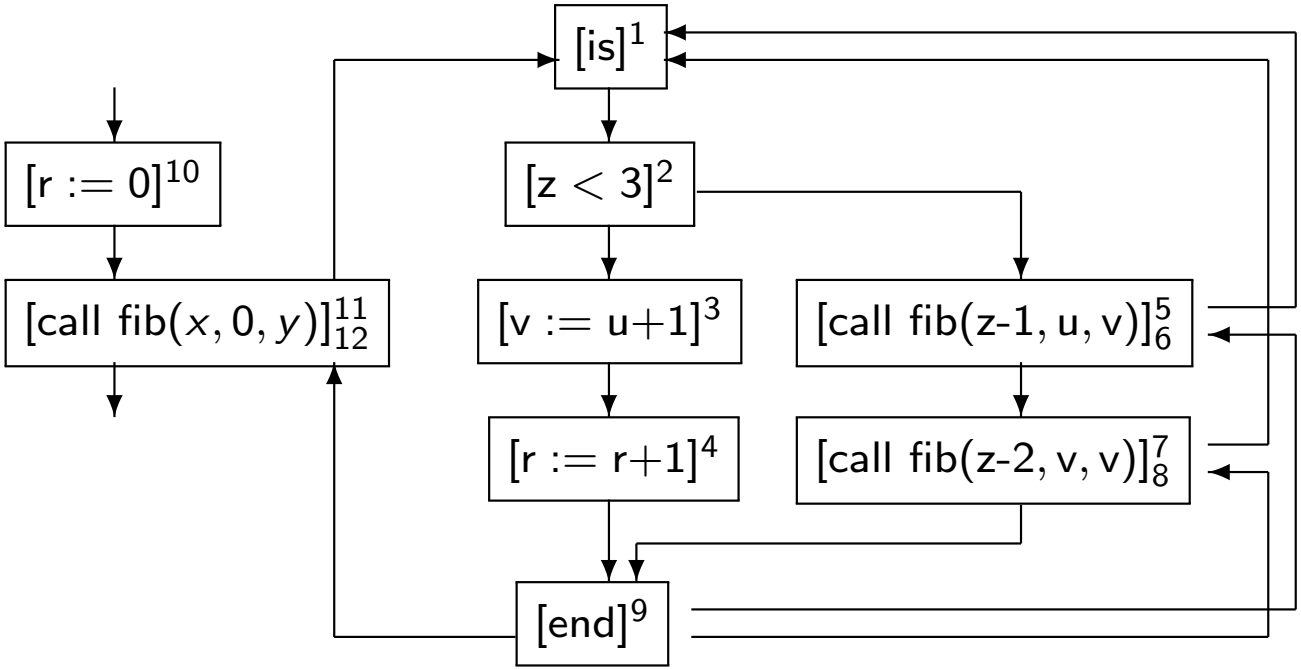
A complete path from ℓ_1 to ℓ_2 in $P_\star$ has proper nesting of procedure entries and exits; and a procedure returns to the point where it was called:

$$
\begin{array}{ll}
CP_{\ell_1,\ell_2} \longrightarrow \ell_1 & \text{whenever } \ell_1 = \ell_2 \\
CP_{\ell_1,\ell_3} \longrightarrow \ell_1, CP_{\ell_2,\ell_3} & \text{whenever } (\ell_1, \ell_2) \in \text{flow}_\star \\
CP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, CP_{\ell_r,\ell} & \text{whenever } (\ell_c, \ell_n, \ell_x, \ell_r) \in \text{inter-flow}_\star
\end{array}
$$

Recall:
$(\ell_c, \ell_n, \ell_r, \ell_x) \in \text{inter-flow}_\star$, if $P_\star$ contains $[\text{call } p(a, z)]_{\ell_r}^{\ell_c}$ as well as $\text{proc } p(\text{val } x;\ \text{res } y) \text{ is}^{\ell_n}\ S \text{ end}^{\ell_x}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: Complete Paths



$CP_{10,12} \rightarrow 10, CP_{11,12}$

$CP_{11,12} \rightarrow 11, CP_{1,9}, CP_{12,12}$

$CP_{1,9} \rightarrow 1, CP_{2,9}$

$CP_{2,9} \rightarrow 2, CP_{3,9}$

$CP_{2,9} \rightarrow 2, CP_{5,9}$

$CP_{3,9} \rightarrow 3, CP_{4,9}$

$CP_{4,9} \rightarrow 4, CP_{9,9}$

$CP_{5,9} \rightarrow 5, CP_{1,9}, CP_{6,9}$

$CP_{6,9} \rightarrow 6, CP_{7,9}$

$CP_{7,9} \rightarrow 7, CP_{1,9}, CP_{8,9}$

$CP_{8,9} \rightarrow 8, CP_{9,9}$

$CP_{12,12} \rightarrow 12$

$CP_{9,9} \rightarrow 9$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Valid Paths

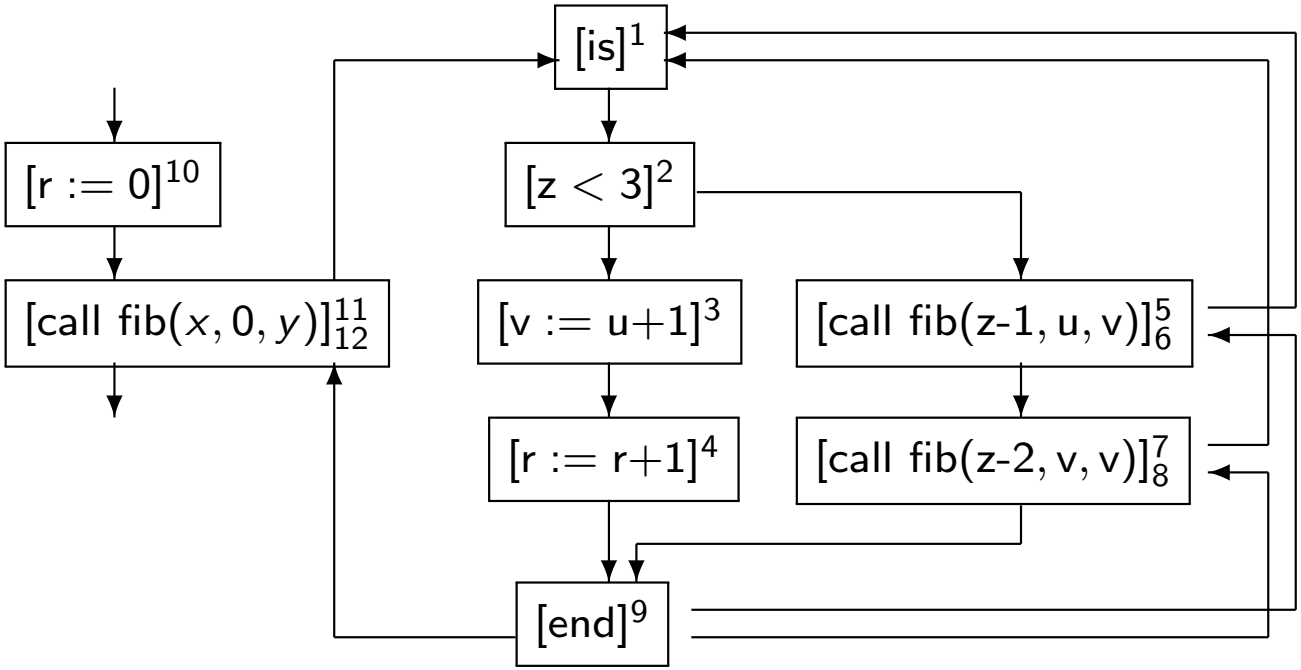
A valid path starts at the entry node $\mathsf{init}_\star$ of $P_\star$, all the procedure exits match the procedure entries but some procedures might be entered but not yet exited:

$$
\begin{array}{ll}
VP_\star \longrightarrow VP_{\mathsf{init}_\star,\ell} & \text{whenever } \ell \in \mathsf{Lab}_\star \\
VP_{\ell_1,\ell_2} \longrightarrow \ell_1 & \text{whenever } \ell_1 = \ell_2 \\
VP_{\ell_1,\ell_3} \longrightarrow \ell_1,\, VP_{\ell_2,\ell_3} & \text{whenever } (\ell_1,\ell_2) \in \mathsf{flow}_\star \\
VP_{\ell_c,\ell} \longrightarrow \ell_c,\, CP_{\ell_n,\ell_x},\, VP_{\ell_r,\ell} & \text{whenever } (\ell_c,\ell_n,\ell_x,\ell_r) \in \text{inter-flow}_\star \\
VP_{\ell_c,\ell} \longrightarrow \ell_c,\, VP_{\ell_n,\ell} & \text{whenever } (\ell_c,\ell_n,\ell_x,\ell_r) \in \text{inter-flow}_\star
\end{array}
$$

Note: The valid paths are generated by the non-terminal $VP_\star$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Example: Valid Paths



$CP_{10,12} \rightarrow 10, CP_{11,12}$

$CP_{11,12} \rightarrow 11, CP_{1,9}, CP_{12,12}$

$CP_{1,9} \rightarrow 1, CP_{2,9}$

$CP_{2,9} \rightarrow 2, CP_{3,9}$

$CP_{2,9} \rightarrow 2, CP_{5,9}$

$CP_{3,9} \rightarrow 3, CP_{4,9}$

$CP_{4,9} \rightarrow 4, CP_{9,9}$

$CP_{5,9} \rightarrow 5, CP_{1,9}, CP_{6,9}$

$CP_{6,9} \rightarrow 6, CP_{7,9}$

$CP_{7,9} \rightarrow 7, CP_{1,9}, CP_{8,9}$

$CP_{8,9} \rightarrow 8, CP_{9,9}$

$CP_{12,12} \rightarrow 12$

$CP_{9,9} \rightarrow 9$

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]

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Meet over Valid Paths: The MVP Solution

$$MVP_\circ(\ell) = \bigsqcap\{f_{\vec{\ell}}(\iota) | \vec{\ell} \in vpath_\circ(\ell)\}$$

$$MVP_\bullet(\ell) = \bigsqcap\{f_{\vec{\ell}}(\iota) | \vec{\ell} \in vpath_\bullet(\ell)\}$$

where

$$vpath_\circ(\ell) = \{[\ell_1, \ldots, \ell_{n-1}] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1, \ldots, \ell_n] \text{ is valid path}\}$$

$$vpath_\bullet(\ell) = \{[\ell_1, \ldots, \ell_n] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1, \ldots, \ell_n] \text{ is valid path}\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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 and adapt it to

- avoid considering too many paths
- taking call context information into account.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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 $\delta \in \Delta$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Towards the MFP Counterpart

- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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 $\Delta = \mathrm{Lab}_*^*$ 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 $\ell_c^1, \ell_c^2, \ldots, \ell_c^m$ is the call string leading to the current call which happened at ℓ_c^m; the previous calls where at $\ell_c^2 \ldots \ell_c^1$. If $m = 0$ then no calls have been performed so far.

For the example program the following call strings are of interest:

$$\Lambda, [11], [11, 5], [11, 7], [11, 5, 5], [11, 5, 7], [11, 7, 5], [11, 7, 7], \ldots$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Adapted MFP Equation System

The Adapted MFP-Equation System:

$$\begin{aligned} A_\circ(\ell) &= \bigsqcap\{A_\bullet(\ell') \mid (\ell',\ell) \in F \vee (\ell';\ell) \in F\} \sqcap \widehat{\iota_E^\ell} \\ A_\bullet(\ell) &= \widehat{f_\ell^A}(A_\circ(\ell)) \end{aligned}$$

where

- $\hat{L} = \Delta \to L$ maps a context to a data flow property (i.e., a data flow lattice element)
- each transfer function $\widehat{f_l}$ is given by $\widehat{f_l}(\hat{l})(\delta) = f_l(\hat{l}(\delta))$ (i.e., $\widehat{f_l}$ adapts resp. specializes f_l to the call context δ)
- $$\widehat{\iota_E^\ell} =_{df} \begin{cases} \iota_E^\ell & \text{if } \delta = \Lambda \\ \bot & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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:

- $\triangle = \mathrm{Lab}_*^{\leq k}$, i.e. call strings of bounded length k.
- $k = 0$: Context insensitive analysis
  - Λ (the 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
  - $\Lambda, [11], [11, 5], [11, 7], [5, 5], [5, 7], [7, 5], [7, 7]$
- ...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Assumption Sets δ as Context Information Δ

Instead of describing a path directly in terms of teh calls being performed (as a textcolorbluecall string does), information about the state in which a call was made can be stored (as an textcolorblueassumption set does).

For a more detailed account of the textcolorblueassumption 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Further Reading for Chapter 13

- 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)
- 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.
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Chapter 2.5, Interprocedural Analysis)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter 14

The Functional Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.1

The Base Setting

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Setting

We consider programs w/ procedures that will be represented in terms of

- Flow graph systems
- Interprocedural flow graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

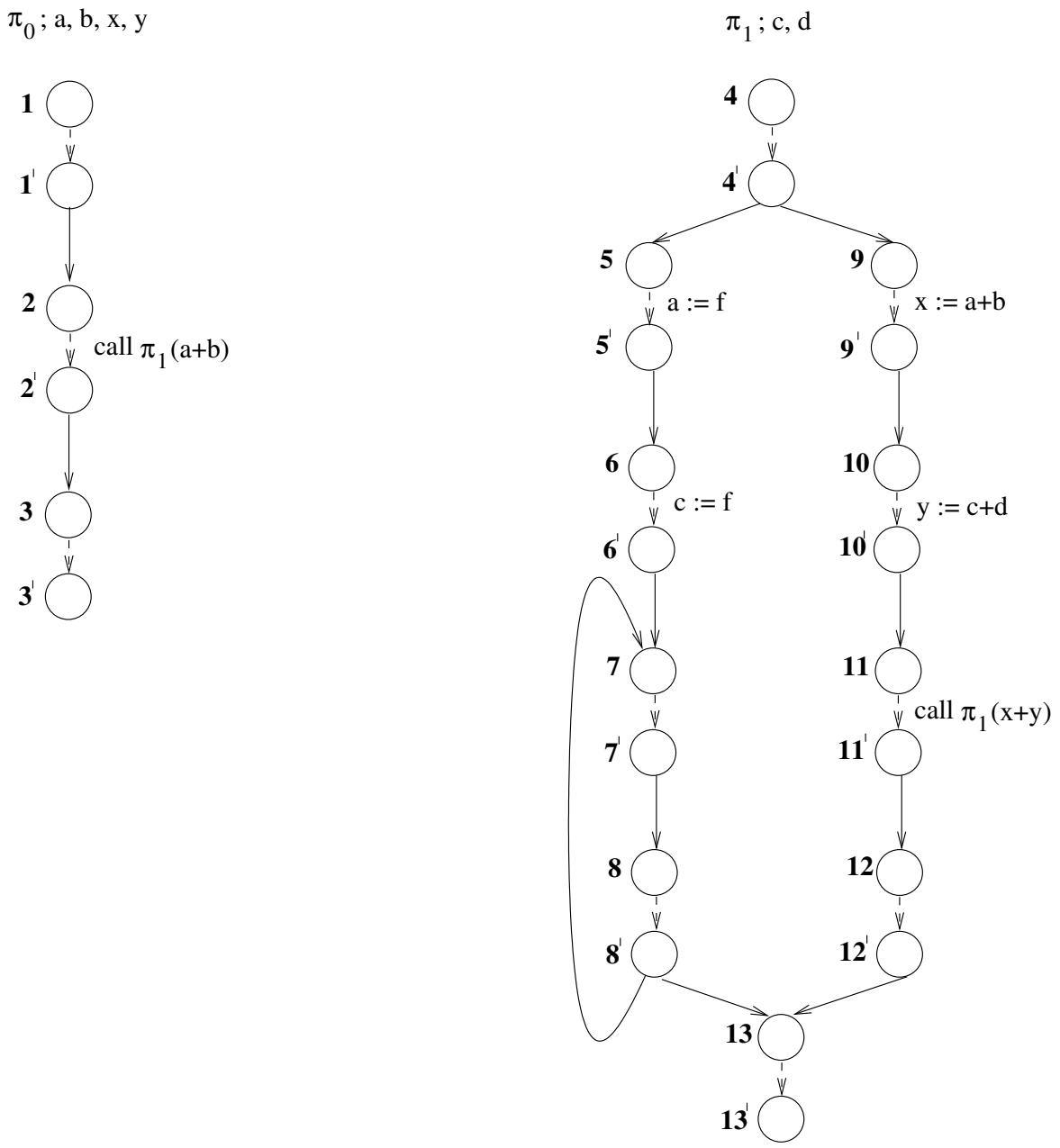
Flow Graph Systems

Definition (14.1.1, Flow Graph System)

A flow graph system $S =_{df} \langle G_0, \ldots, G_k \rangle$ is a system of (intra-procedural) flow graphs in the sense of Chapter 4, where each flow graph G_i represents a procedure of the underlying program Π. The flow graph G_0 of S represents the main procedure or the main program of Π.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Illustration: Flow Graph System



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Flow Graphs

Definition (14.1.2, Interprocedural Flow Graph)

A flow graph system S induces an interprocedural flow graph, where the flow graphs of S are melted to a single flow graph $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$.

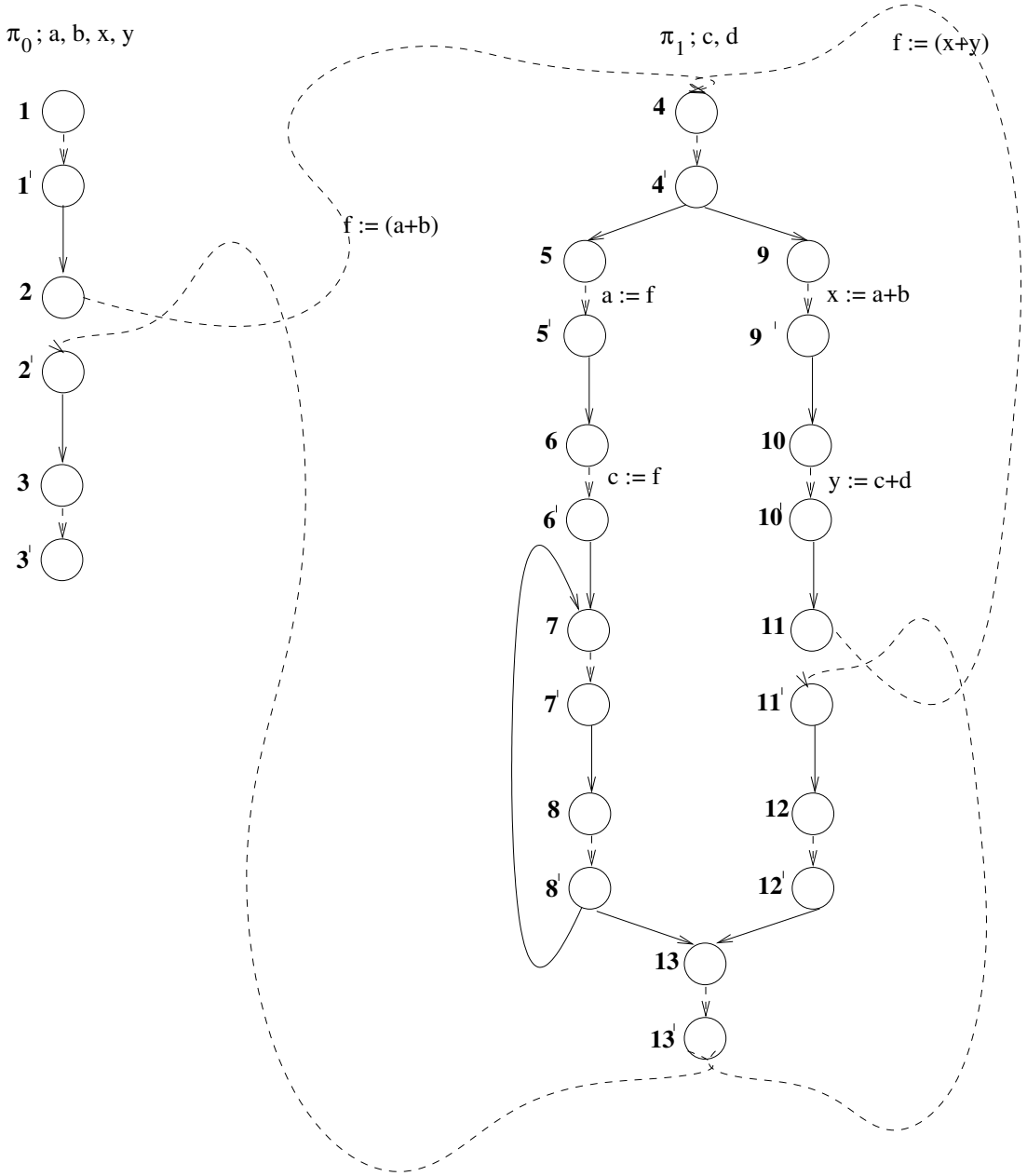
G^* evolves from S by replacing each call edge e of a procedure π by two new edges e_c and e_r.

The edge e_c connects the source node of e w/ the start node of the called procedure.

The edge e_r connects the end node of the called procedure w/ the final node of e.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Illustration: Interprocedural Flow Graph



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Notations for Flow Graph Systems

- G_0 represents the main procedure.
- The start node $\mathbf{s}_0$ of G_0 is often abbreviated by $\mathbf{s}$.
- The sets of nodes and edges N_i und E_i, $0 \leq i \leq k$, are assumed to be pairwise disjoint.
- $N =_{df} \bigcup\{N_i \mid i \in \{0, \ldots, k\}\}$ and $E =_{df} \bigcup\{E_i \mid i \in \{0, \ldots, k\}\}$ denote the set of all nodes and edges of a flow graph system.
- $E_{call} \subseteq E$ denotes the set of edges, which represent a procedure call, for short, the set of call edges.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

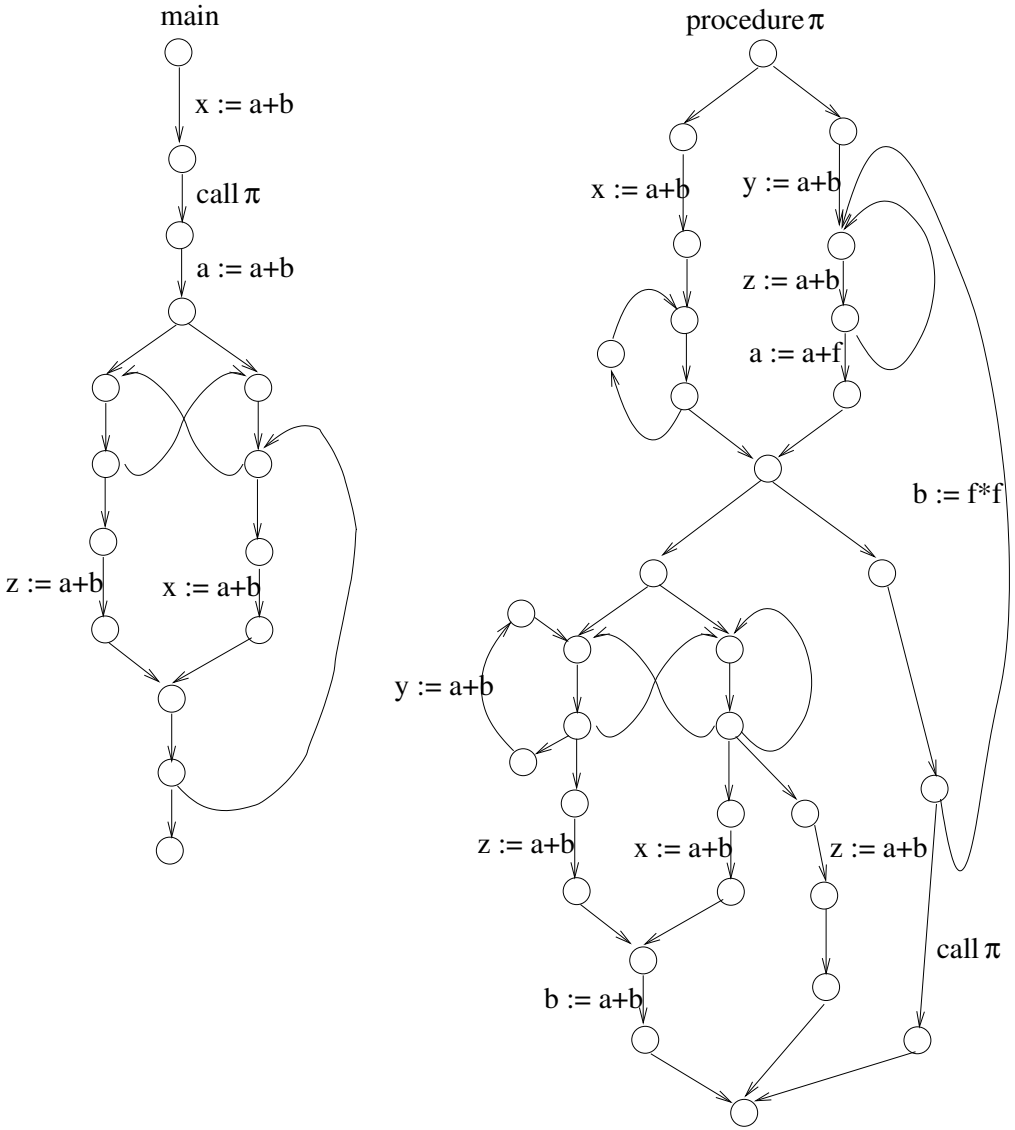
Notations for Interprocedural Call Graphs

- The set of new edges in an interprocedural flow graph are called the call edges and return edges of G^*, and are denoted by E_c^* und E_r^*.
- $E_{call}^* =_{df} E_c^* \cup E_r^*$ denotes the set of call and return edges of G^*.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Streamlining Flow Graph Systems

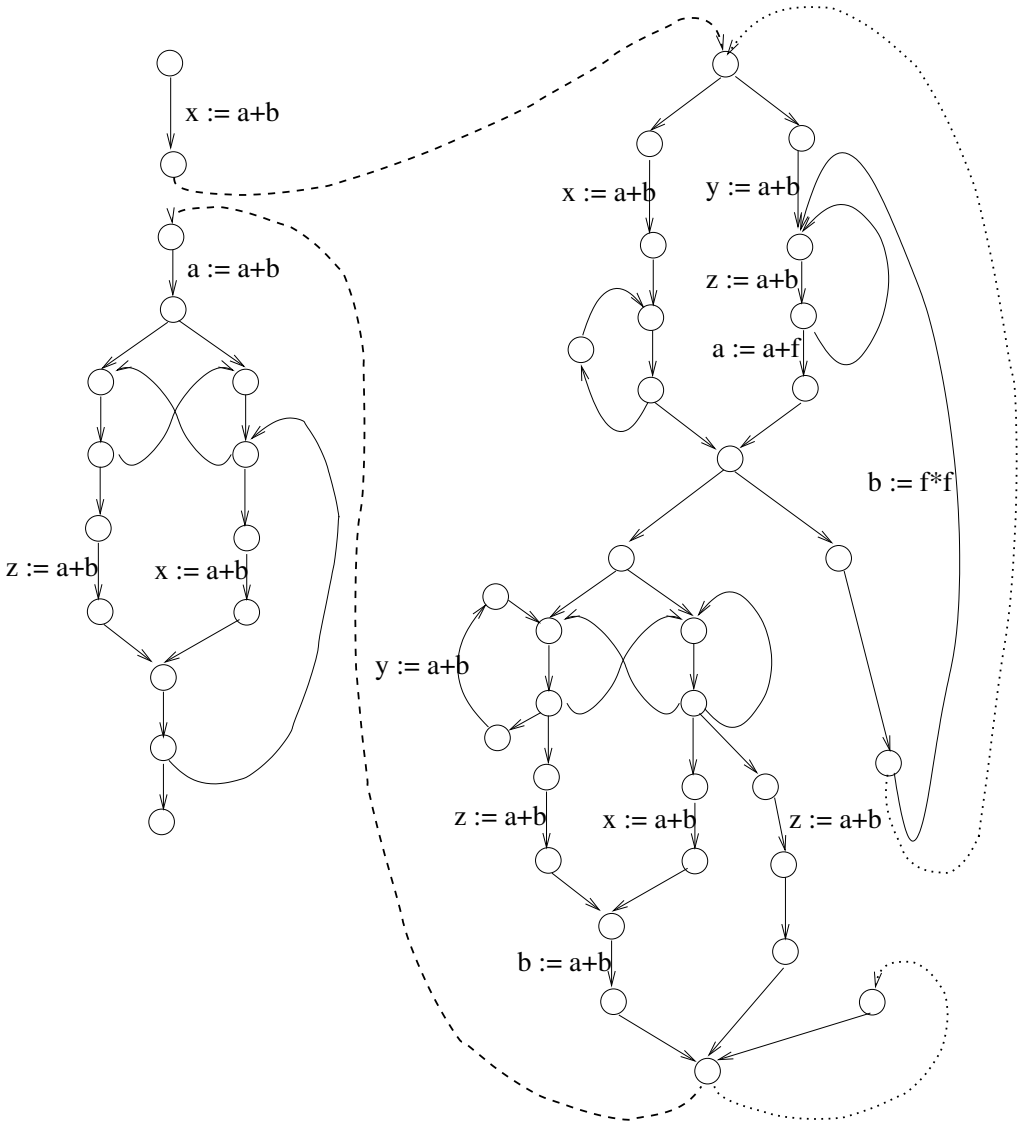
...by removing unnecessary nodes and edges in flow graph systems:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Streamlining Interprocedural Flow Graphs

...as well as in interprocedural flow graphs:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Key to Interprocedural DFA (1)

The Functional *MaxFP*-Approach:

The "functional" *MaxFP* results from lifting the analysis level from elements to functions. It is the pointwise extension of the *MaxFP* approach to all DFA-lattice elements:

The Functional *MaxFP* Equation System 14.1.3

$$[\![n]\!] = \begin{cases} Id_{\mathcal{C}} & \text{if } n = \mathbf{s} \\ \bigsqcap \{ [\![(n, m)]\!] \circ [\![m]\!] \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Let

$$[\![\;\;]\!]^* : N \to (\mathcal{C} \to \mathcal{C})$$

denote the greatest solution of the above equation system.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Key to Interprocedural DFA (2)

Intuitively

The functional *MaxFP* approach lifts the *MaxFP* approach to the level of functions, i.e.

- it computes the *MaxFP* solution not just for a specially selected single lattice element as start information but simultaneously for all.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Key to Interprocedural DFA (3)

MaxFP approach vs. functional *MaxFP* approach:

The Equivalence Theorem 14.1.4 characterizes the relationship of the *MaxFP* approach and the functional *MaxFP* approach:

Theorem (14.1.4, Equivalence)

$\forall\, n \in N\ \forall\, c_{\mathsf{s}} \in \mathcal{C}.\ MaxFP_{(G,[\![\ ]\!])}(n)(c_s) = [\![\, n \,]\!]^*(c_{\mathsf{s}})$

In the following we will overload the symbol $[\![\ ]\!]$ and use it to also denote the greatest fixed point $[\![\, n \,]\!]^*$ of the functional *MaxFP* equation system 14.1.3.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Outlook

The functional variant of the *MaxFP* approach is the key to

- interprocedural (i.e., of programs w/ procedures)
- object-oriented (i.e., of programs w/ classes, objects, and methods)
- parallel (i.e., of programs w/ parallelism)

data flow analysis.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.1.1

Local Abstract Semantics

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

(Local) Abstract Semantics

Two components:

- Data flow analysis lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$
- Data flow analysis functional $[\![\ ]\!]' : E^* \to (\mathcal{C} \to \mathcal{C})$

Note: In the parameterless base setting call edges and return edges of E^* are given the identity function on $\mathcal{C}$ as their semantics.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.1.2

The *IMOP* Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedurally Valid Paths (1)

Observations:

- The notion of a finite path for intraprocedural flow graphs extends naturally to interprocedural flow graphs.
- Unlike, however, as in intraprocedural flow graphs, where each path connecting two nodes represents (up to non-determinism) a possible execution of the program, this does not hold for interprocedural flow graphs.
- In interprocedural DFA this is taken care of by focusing on interprocedurally valid paths.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedurally Valid Paths (2)

Intuitively: Interprocedurally valid paths respect the call/return behaviour of procedures.

Definition (14.1.2.1, Interprocedurally Valid Path)

Identifying call and return edges of G^* with opening and closing brackets "(" and ")", 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 $\mathbf{IP}[m, n]$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedurally Valid Paths (3)

Observation: If we consider the sequences of edge labelings (we suppose that each edge is uniquely marked by some label) of a path as word of a formal language, then the set of intraprocedurally valid paths is given by a regular language, the one of interprocedurally valid paths by a context-free language.

Note:

- Sharir and Pnueli gave an algorithmic definition of interprocedurally valid paths in 1981.
- An immediate definition of interprocedurally valid paths in terms of a context-free language is possible, too.
- The definition of interprocedurally valid paths as in Definition 14.1.2.1 is due to Reps, Horwitz, and Sagiv, POPL'95.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The *IMOP* Approach

The *IMOP* Solution:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMOP_{c_{\mathbf{s}}}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!]'(c_{\mathbf{s}}) \mid p \in \mathbf{IP}[\mathbf{s}, n] \,\}$$

where $\mathrm{IP}[\mathbf{s}, n]$ denotes the set of interprocedurally valid paths from $\mathbf{s}$ to n.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.1.3

The *IMaxFP* Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The *IMaxFP* Approach

...is a two-stage approach:

- Stage 1: Preprocess – Computing the Semantics of Procedures
- Stage 2: Main Process – Computing the *IMaxFP* solution

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Notations

The definition of the *IMaxFP* approach requires the following mappings on a flow graph system S:

- $\mathit{flowGraph} : N \cup E \to S$ maps the nodes and edges of S to the flow graph containing them.
- $\mathit{callee} : E_{call} \to S$ maps every call edge to the flow graph of the called procedure.
- $\mathit{caller} : S \to \mathcal{P}(E_{call})$ maps every flow graph to the set of call edges calling it.
- $\mathit{start} : S \to \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ and $\mathit{end} : S \to \{\mathbf{e}_0, \ldots, \mathbf{e}_k\}$ map every flow graph of S to its start node and stop node.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The *IMaxFP* Approach (1)

Stage 1: Preprocess – Computing the Semantics of Procedures

The 2nd Order *IMaxFP* Equation System 14.1.3.1

$$[\![n]\!] =$$

$$\begin{cases} Id_{\mathcal{C}} & \text{if } n \in \{ \mathbf{s}_0, \ldots, \mathbf{s}_k \} \\ \bigsqcap \{ [\![(m, n)]\!] \circ [\![m]\!] \mid m \in pred_{flowGraph(n)}(n) \} & \text{otherwise} \end{cases}$$

and

$$[\![e]\!] = \begin{cases} [\![e]\!]' & \text{if } e \in E \backslash E_{call} \\ [\![end(caller(e))]\!] & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The *IMaxFP* Approach (2)

Stage 2: Main Process – The "Actual" Interprocedural DFA

The 1st Order *IMaxFP* Equation System 14.1.3.2

$$inf(n) =$$

$$\begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s}_0 \\ \bigsqcap \{\, inf(src(e)) \mid e \in caller(flowGraph(n)) \,\} & \text{if } n \in \{\mathbf{s}_1, \ldots, \mathbf{s}_k\} \\ \bigsqcap \{\, [\![(m,n)]\!](inf(m)) \mid m \in pred_{flowGraph(n)}(n) \,\} & \text{otherwise} \end{cases}$$

The *IMaxFP* Solution:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{c_{\mathbf{s}}}(n) =_{df} inf^*_{c_{\mathbf{s}}}(n)$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.1.4

Main Results

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Main Results – 1st Stage

Safety and coincidence results of the 2nd-order 1st-stage analysis:

Theorem (14.1.4.1, 2nd Order)

For all $e \in E_{call}$ *hold:*

1. $[\![e]\!] \sqsubseteq \bigsqcap\{[\![p]\!]' \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, *if the data flow analysis functional* $[\![\ ]\!]'$ *is monotonic.*
2. $[\![e]\!] = \bigsqcap\{[\![p]\!]' \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, *if the data flow analysis functional* $[\![\ ]\!]'$ *is distributive.*

where the mappings *src* and *dst* yield the start and final node of an edge.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Complete Interprocedural Paths

Definition (14.1.4.2, Complete Interproc. Path)

An interprocedural path p from the start node $\mathbf{s}_i$ of a procedure G_i, $i \in \{0, \ldots, k\}$, 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 $\mathbf{s}_i$ to n with $\mathbf{CIP}[\mathbf{s}_i, n]$.

Note:

- Intuitively, the completeness requirement states that the occurrences of $\mathbf{s}_i$ and n belong to the same incarnation of the procedure.
- We have that the subpaths of a complete interprocedural path that belong to a procedure call, are either disjoint or properly nested.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Main Results – 2nd Stage

Safety and coincidence results of the 1st-order 2nd-stage analysis:

Theorem (14.1.4.3, Interprocedural Safety)

The IMaxFP solution is a safe, i.e., a lower approximation of the IMOP solution, i.e.

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{c_s}(n) \sqsubseteq IMOP_{c_s}(n)$$

if the data flow analysis functional $[\![\ ]\!]'$ *is monotonic.*

Theorem (14.1.4.4, Interprocedural Coincidence)

The IMaxFP solution coincides with the IMOP solution, i.e.

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{c_s}(n) = IMOP_{c_s}(n)$$

if the data flow analysis functional $[\![\ ]\!]'$ *is distributive.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.1.5

Algorithms

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 2nd Order Alg. 14.1.5.1 – 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 $[\![\ ]\!]' : E^* \to (\mathcal{C} \to \mathcal{C})$.

Output: Under the assumption of termination (cf. Theorem 14.1.5.4), an annotation of S with functions $[\![\![\, n \,]\!]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *gtr* , which stands for *global transformation*), and $[\![\, e \,]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *ltr* , which stands for *local transformation*) representing the greatest solution of the 2nd order Equation System 14.1.3.1.

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 $[\![\![\ ]\!]\!]$ and $[\![\ ]\!]$ the iterative approximation of $[\![\![\ ]\!]\!]$ is superposed by an interprocedural iteration step, which updates the current approximation of the effect $[\![\ ]\!]$ of call edges. The temporary *meet* stores the result of the most recent meet operation.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 2nd Order Alg. 14.1.5.1 – Preprocess

(Prologue: Initializing the annotation arrays *gtr* and *ltr* and the variable *workset*)

FORALL $n \in N$ DO

IF $n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ THEN $gtr\ [n] := Id_{\mathcal{C}}$

ELSE $gtr\ [n] := \top_{[\mathcal{C} \to \mathcal{C}]}$ FI OD;

FORALL $e \in E$ DO

IF $e \in E_{call}$ THEN $ltr\ [e] := \top_{[\mathcal{C} \to \mathcal{C}]}$ ELSE $ltr\ [e] := [\![\ e\ ]\!]'$ FI OD;

$workset := \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 2nd Order Alg. 14.1.5.1 – Preprocess

```
( Main process: Iterative fixed point computation )
WHILE workset ≠ ∅ DO
    CHOOSE m ∈ workset ;
        workset := workset\{m};
        ( Update the successor-environment of node m )
        IF m ∈ {e_1, ..., e_k}
            THEN
                FORALL e ∈ caller(flowGraph(m)) DO
                    ltr[e] := gtr[m];
                    meet := ltr[e] ∘ gtr[src(e)] ⊓ gtr[dst(e)];
                    IF gtr[dst(e)] ⊐ meet
                        THEN
                            gtr[dst(e)] := meet;
                            workset := workset ∪ {dst(e)}
                    FI
                OD
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 2nd Order Alg. 14.1.5.1 – Preprocess

```
            ELSE ( i.e., m ∉ {e_1, ..., e_k} )
                FORALL n ∈ succ_flowGraph(m)(m) DO
                    meet := ltr[(m, n)] ∘ gtr[m] ⊓ gtr[n];
                    IF gtr[n] ⊐ meet
                        THEN
                            gtr[n] := meet;
                            workset := workset ∪ {n}
                    FI
                OD
        FI
    ESOOHC
OD.
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order Alg. 14.1.5.2 – 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 $[\![\ ]\!]$ computed by Algorithm 14.1.5.1, and (3) a context information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 14.5.1.4), the *IMaxFP*-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is *distributive*: variable *inf* stores for every node the strongest component information valid there wrt the context information $c_{\mathbf{s}}$.
(2) $[\![\ ]\!]$ is *monotonic*: variable *inf* stores for every node a valid component information wrt the context information $c_{\mathbf{s}}$, 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order Alg. 14.1.5.2 – Main Process

(Prologue: Initialization of the annotation array *inf* and the variable *workset*)

FORALL $n \in N \backslash \{\mathbf{s}_0\}$ DO $inf[n] := \top$ OD;

$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;

$workset := \{\, \mathbf{s}_0 \,\}$;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order Alg. 14.1.5.2 – Main Process

```
( Main process: Iterative fixed point computation )
WHILE workset ≠ ∅ DO
    CHOOSE m ∈ workset;
        workset := workset \ { m };
        ( Update the successor-environment of node m )
        FORALL n ∈ succ_flowGraph(m)(m) DO
            meet := ⟦ (m, n) ⟧(inf[m]) ⊓ inf[n];
            IF inf[n] ⊐ meet
                THEN
                    inf[n] := meet;
                    workset := workset ∪ {n}
            FI;
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order Alg. 14.1.5.2 – Main Process

```
            IF (m, n) ∈ E_call
                THEN
                    meet := inf[m] ⊓ inf[start(callee((m, n)))];
                    IF inf[start(callee((m, n)))] ⊐ meet
                        THEN
                            inf[start(callee((m, n)))] := meet;
                            workset := workset ∪ { start(callee((m, n))) }
                    FI
            FI
        OD
    ESOOHC
OD.
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

A 1st Variant of the *IMaxFP*-Algorithm

- Algorithm 14.1.5.3 uses the semantics functions computed by Algorithm 14.1.5.1 more efficiently.
- Algorithm 14.1.5.1 and 14.1.5.3 constitute a pair of algorithms computing the *IMaxFP* solution, too.
- Replacing Algorithm 14.5.1.2 by Algorithm 14.1.5.3 has no impact on Algorithm 14.1.5.1.
- Unlike Algorithm 14.1.5.2, Algorithm 14.1.5.3 does not iterate over all nodes but only over procedure start nodes. After stabilization of the solution for the start nodes, a single run over all other nodes in the epilogue suffices to compute the *IMaxFP* solution at every node.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order Algorithm 14.1.5.3 – The "Functional" 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 $[\![\ ]\!]=_{df} gtr$ and $[\![\ ]\!]=_{df} ltr$ with respect to $\mathcal{C}$ (computed by Algorithm 14.1.5.1), and (4) a context information $c_{\mathsf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 14.1.5.4), the *IMaxFP*-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is *distributive*: variable *inf* stores for every node the strongest component information valid there wrt the context information c_{s}.
(2) $[\![\ ]\!]$ is *monotonic*: variable *inf* stores for every node a valid component information wrt 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order Algorithm 14.1.5.3 – The "Functional" Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)
FORALL $\mathbf{s} \in \{\mathbf{s}_i \mid i \in \{1, \ldots, k\}\}$ DO $inf[\mathbf{s}] := \top$ OD;
$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;
$workset := \{\mathbf{s}_i \mid i \in \{1, 2, \ldots, k\}\}$;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order Algorithm 14.1.5.3 – The "Functional" Main Process

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

  CHOOSE $\mathbf{s} \in workset$;

    $workset := workset \backslash \{\mathbf{s}\}$;

    $meet := inf[\mathbf{s}] \sqcap$

      $\bigsqcap \{ [\![src(e)]\!] (inf[start(flowGraph(e))]) \mid e \in caller(flowGraph(\mathbf{s})) \}$;

    IF $inf[\mathbf{s}] \sqsupset meet$

      THEN

        $inf[\mathbf{s}] := meet$;

        $workset := workset \cup \{start(callee(e)) \mid e \in E_{call}. flowGraph(e) = flowGraph(\mathbf{s}) \}$

    FI

  ESOOHC

OD;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The 1st Order *IMaxFP*-Algorithm 14.1.5.3 – The "Functional" Main Process

(Epilogue)

FORALL $n \in N \backslash \{\mathbf{s}_i \mid i \in \{0, \ldots, k\}\}$ DO

$$inf[n] := [\![\, n \,]\!](inf[start(flowGraph(n))])$$ OD.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Termination

Theorem (14.1.5.4, Termination)

The sequential composition of Algorithm 14.1.5.1 and Algorithm 14.1.5.2 resp. Algorithm 14.1.5.3 terminates with the IMaxFP solution, if the data flow analysis functional $[\![\ ]\!]'$ *is monotonic and the function lattice* $[\mathcal{C} \to \mathcal{C}]$ *satisfies the descending chain condition.*

Note: The descending chain condition on $[\mathcal{C} \to \mathcal{C}]$ implies the descending chain condition on $\mathcal{C}$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

A 2nd Variant of the *IMaxFP*-Algorithm (1)

Partial instead of total computation of the semantics of the procedures:

- Unlike to the previous two algorithm variants, the new variant allows an interleaving of preprocess and main process.
- The computation starts with the main process algorithm.
- If a procedure call is encountered during the iterative process, the 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 main process algorithm is continued with the computed result.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

A 2nd Variant of the *IMaxFP*-Algorithm (2)

Note:

- The computation of the semantics of the procedures is performed demand-drivenly.
- The semantics of procedures are only computed as as far as necessary.
- Overall, this results in some efficiency gain in practice, which, however, is difficult to quantify.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.1.6

Applications

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Applications

- For the parameterless base setting the specifications of intraprocedural DFA problems can be reused unmodified.
- In order to be effective, the descending chain condition must hold both for the data flow analysis lattice and its corresponding function lattice.
- This condition holds in particular for all bitvector problems (availability of expressions, lifeness of variables, reaching definitions, etc.) but not for simple constants. Therefore, weaker and simpler classes of constants are used interprocedurally, e.g., the set of linear constants.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.2

The General Setting

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6
14.2

Outline

We extend our setting by adding

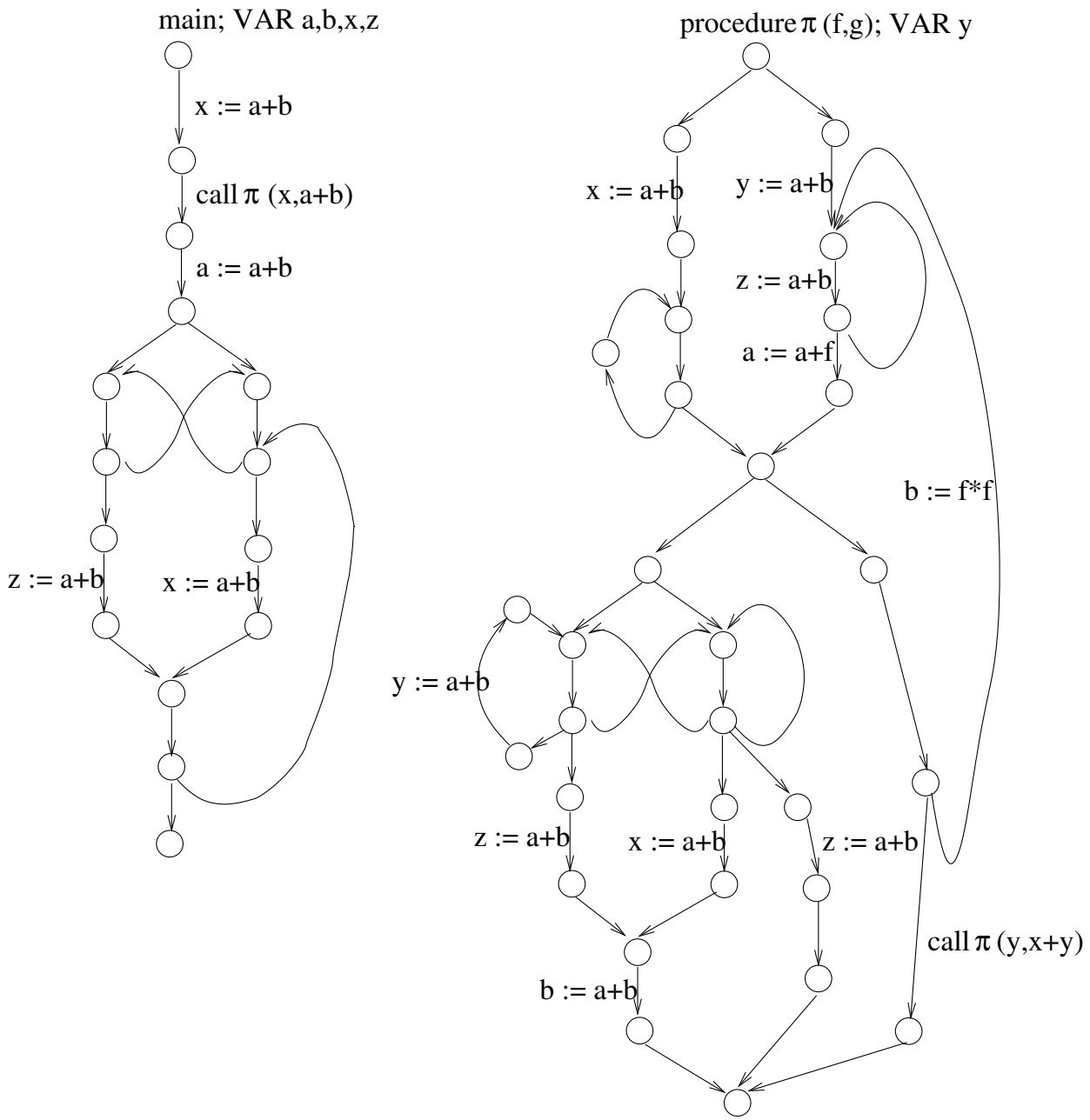
- Value parameters
- Local variables

This requires to adjust our program representations towards

- Flow graph systems (FGS) w/ value parameters and local variables
- Interprocedural flow graphs (IFG) w/ value parameters and local variables

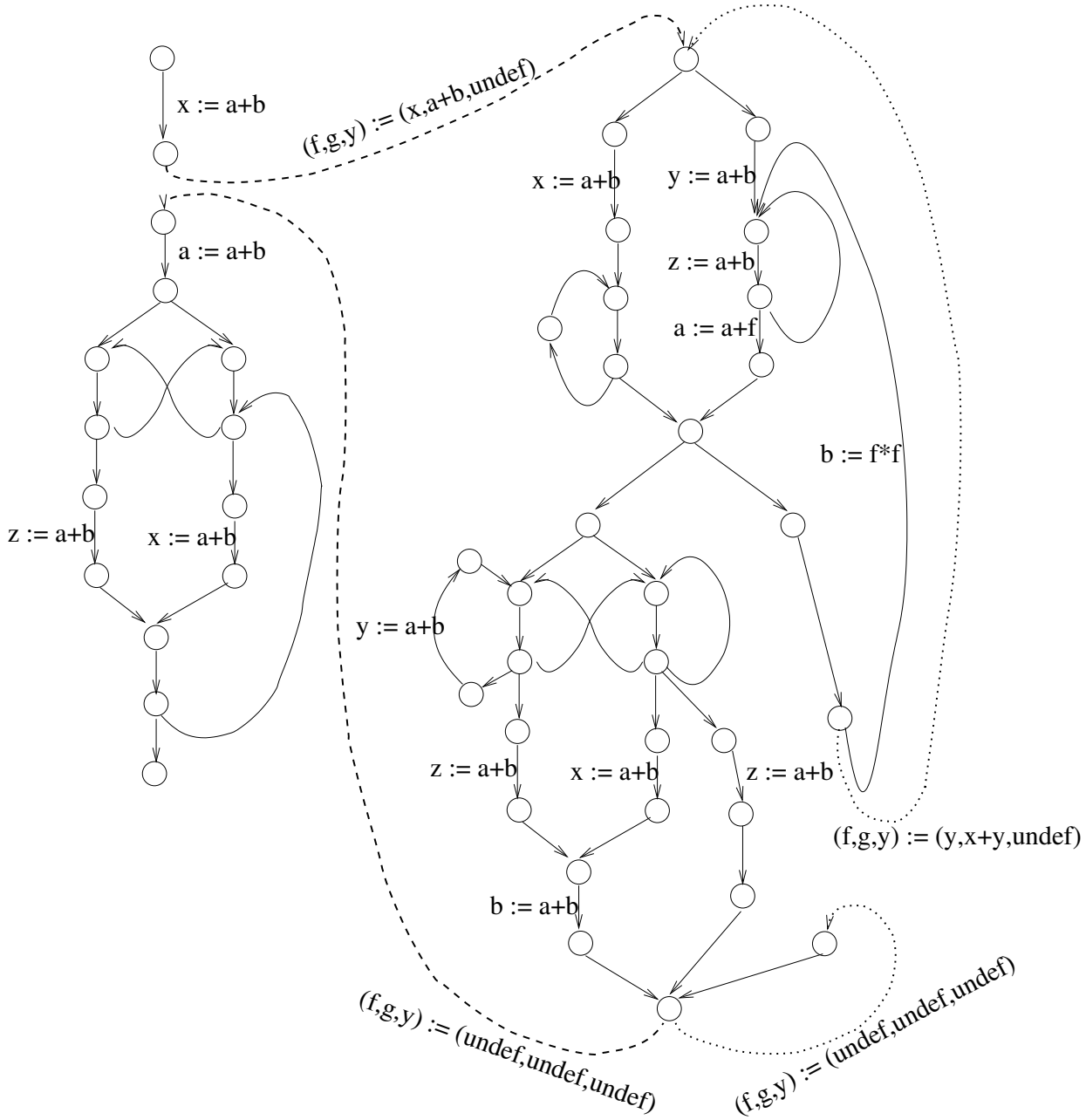
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6
14.2

FGS w/ Value Parameters and Local Variables



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6
14.2

IFG w/ Value Parameters and Local Variables



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6
14.2

New Phenomena

...related to procedures, value parameters, and local variables.

Conceptually most important:

- Existence of an unlimited number of copies (incarnations) of local variables and value parameters at run-time due to recursive procedures.
- After termination of a recursive procedure call the local variables and value parameters of the proceding not yet finished procedure call become valid again.
- The run-time system handles this phenomena by means of of a run-time stack which stores the activation records of the various procedure incarnations.

For program analysis, we have to take these phenomena into account and to model them properly.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6
14.2

Data Flow Analysis Stacks

Intuitively:

- DFA stacks are a compile-time equivalent of run-time stacks.
- Entries in DFA stacks are data flow facts of an underlying DFA lattics $\mathcal{C}$.
- We denote the set of all non-empty DFA stacks by $STACK$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6
14.2

Manipulating DFA Stacks

DFA stacks can be 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6
14.2

Remarks (1)

- The usual stack function emptystack : $\rightarrow STACK$ is replaced by newstack. Empty DFA stacks are not considered since they do not occur in interprocedural DFA.
- push and pop allow to manipulate the top-most entries of a DFA stack. This is different to and less flexible as for run-time stacks but suffices for interprocedural DFA.
- In fact, DFA stacks are only conceptually relevant, i.e., for the specifying, i.e., for the specifying *IMOP* approach but not for the algorithmic *IMaxFP* approach.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Remarks (2)

- Like run-time stacks DFA stacks store that part of the history of a program path that is relevant after finishing a procedure call.
- DFA stack entries can be considered abstractions of the activation records of procedure calls.
- The top-most entry of a DFA stack represents always the currently valid activation record (therefore, DFA stacks are never empty).
- Other than the top-most DFA stack entries represent the activation records of already started but not yet finished procedure calls.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.2.1

Local Abstract Semantics

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Basic Local Abstract Semantics

Basic Local Abstract Semantics on DFA Lattice

1. DFA lattics $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$
2. DFA functional $[\![\ ]\!]' : E^* \to (\mathcal{C} \to \mathcal{C})$
3. Return functional $\mathcal{R} : E_{call} \to (\mathcal{C} \times \mathcal{C} \to \mathcal{C})$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Induced Local Abstract Semantics

Induced Local Abstract Semantics on DFA Stacks

- DFA functional $[\![\ ]\!]^* : E^* \to (STACK \to STACK)$ on DFA stacks induced by a basic local abstract semantics that is defined by

$$\forall\, e \in E^*\ \forall\, stk \in STACK.\ [\![\, e \,]\!]^*(stk) =_{df}$$

$$\begin{cases} \text{push}(\,\text{pop}(stk),\ [\![\, e \,]\!]'(\text{top}(stk))\,) & \text{if } e \in E^* \backslash E^*_{call} \\ \text{push}(\,stk,\ [\![\, e \,]\!]'(\text{top}(stk))\,) & \text{if } e \in E^*_c \\ \text{push}(\,\text{pop}(\text{pop}(stk)),\ \mathcal{R}_e(\,\text{top}(\text{pop}(stk)),\ [\![\, e \,]\!]'(\text{top}(stk))\,)\,) & \\ & \text{if } e \in E^*_r \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Notations related to DFA Stacks

- $STACK_{\geq i}$ ($STACK_{\leq i}$, etc.), $i \in \mathbb{N}$ denotes the set of all DFA Stacks w/ at last (at most, etc.) i entries (hence $STACK$ equals $STACK_{\geq 1}$.
- $STACK_i$, $i \in \mathbb{N}$, denotes the set of all DFA Stacks w/ exactly i entries.
- ϑ_{stk} denotes the number of entries of the DFA stack stk.
- stk_i, $1 \leq i \leq \vartheta_{stk}$, denotes the ith entry of the DFA stack stk.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Properties

Lemma (14.2.1.1)

Let $e \in E^*$ *and* $stk \in STK$. *Then we have:*

1. $[\![e]\!]^*(stk) \in \begin{cases} STK_{\vartheta_{stk}} & \text{if } e \in E^* \backslash E^*_{call} \\ STK_{\vartheta_{stk}+1} & \text{if } e \in E^*_c \\ STK_{\vartheta_{stk}-1} & \text{if } e \in E^*_r \wedge \vartheta_{stk} \geq 2 \end{cases}$
2. $pop([\![e]\!]^*(stk)) = pop(stk)$, *if* $e \in E^* \backslash E^*_{call}$
3. $pop([\![e]\!]^*(stk)) = stk$, *if* $e \in E^*_c$
4. $pop([\![e]\!]^*_R(stk)) = pop(pop(stk))$, *if* $e \in E^*_r \wedge \vartheta_{stk} \geq 2$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Structure of the Semantic Functions

All semantic functions occurring in interprocedural DFA are an element of the following subsets of the set of all functions $\mathcal{F} =_{df} [\, STACK \to STACK \,]$ on DFA stacks:

- $\mathcal{F}_{ord}$
- $\mathcal{F}_{psh}$
- $\mathcal{F}_{pop}$

These sets of functions are given by:

$$\mathcal{F}_{ord} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK.\ \mathrm{pop}(f(stk)) = pop(stk) \,\}$$

$$\mathcal{F}_{psh} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK.\ \mathrm{pop}(f(stk)) = stk \,\}$$

$$\mathcal{F}_{pop} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK_{\geq 2}.\ \mathrm{pop}(f(stk)) = \mathrm{pop}(\mathrm{pop}(stk)) \,\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Properties

Lemma (14.2.1.2)

$\forall f_{pp} \in \mathcal{F}_{pop} \;\; \forall f_o, f'_o \in \mathcal{F}_{ord} \;\; \forall f_{ph} \in \mathcal{F}_{psh}.$

1. $f_o \circ f'_o \in \mathcal{F}_{ord}$
2. $f_{pp} \circ f_o \circ f_{ph} \in \mathcal{F}_{ord}$

Lemma (14.2.1.3)

1. $\forall e \in E^* \backslash E^*_{call}. \; [\![e]\!]^* \in \mathcal{F}_{ord}$
2. $\forall e \in E^*_c. \; [\![e]\!]^* \in \mathcal{F}_{psh}$
3. $\forall e \in E^*_r. \; [\![e]\!]^* \in \mathcal{F}_{pop}$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Significant Part of DFA Functions

Only the two top-most entries of DFA stacks are modified by DFA functions. This gives rise to the following definition:

Definition (14.2.1.4, Significant Part)

- $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh}$: Then $f_s : \mathcal{C} \to \mathcal{C}$ is defined by:
  $f_s(c) =_{df} \mathrm{top}(f(\mathrm{newstack}(c)))$
- $f \in \mathcal{F}_{pop}$: Then $f_s : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined by:
  $f_s(c_1, c_2) =_{df} \mathrm{top}(f(\mathrm{push}(\mathrm{newstack}(c_1), c_2)))$
  (Note that $\mathcal{C} \times \mathcal{C}$ is a lattice, if $\mathcal{C}$ is a lattice.)

We have:

Lemma (14.2.1.5)

1. $\forall\, e \in E^* \backslash E_r^*.\ [\![\, e \,]\!]_s^* = [\![\, e \,]\!]'$
2. $\forall\, e \in E_r^*\ \forall\, c_1, c_2 \in \mathcal{C} \times \mathcal{C}.\ [\![\, e \,]\!]_s^* = \mathcal{R}_e(c_1, [\![\, e \,]\!]'(c_2))$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Properties

Lemma (14.2.1.6)

1. $\forall\, e \in E^* \backslash E^*_{call}\ \forall\, stk \in STK.\ [\![\, e \,]\!]^*(stk) = stk'$ *with* $\vartheta_{stk'} = \vartheta_{stk}$ *and*

$$\forall\, 1 \leq i \leq \vartheta_{stk'}.\ stk_i^{'} =_{df} \begin{cases} stk_i & \text{if } i < \vartheta_{stk} \\ [\![\, e \,]\!]^*_s(stk_{\vartheta_{stk}}) & \text{if } i = \vartheta_{stk} \end{cases}$$

2. $\forall\, e \in E^*_c\ \forall\, stk \in STK.\ [\![\, e \,]\!]^*(stk) = stk'$ *with* $\vartheta_{stk'} = \vartheta_{stk} + 1$ *and*

$$\forall\, 1 \leq i \leq \vartheta_{stk'}.\ stk_i^{'} =_{df} \begin{cases} stk_i & \text{if } i < \vartheta_{stk} + 1 \\ [\![\, e \,]\!]^*_s(stk_{\vartheta_{stk}}) & \text{if } i = \vartheta_{stk} + 1 \end{cases}$$

3. $\forall\, e \in E^*_r\ \forall\, stk \in STK_{\geq 2}.\ [\![\, e \,]\!]^*(stk) = stk'$ *with* $\vartheta_{stk'} = \vartheta_{stk} - 1$ *and*

$$\forall\, 1 \leq i \leq \vartheta_{stk'}.\ stk_i^{'} =_{df} \begin{cases} stk_i & \text{if } i < \vartheta_{stk} - 1 \\ [\![\, e \,]\!]^*_s(stk_{\vartheta_{stk}-1}, stk_{\vartheta_{stk}}) & \text{if } i = \vartheta_{stk} - 1 \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

S-Monotonicity, S-Distributivity

Definition (14.2.1.7, S-Mon., S-Distrib.)

A DFA function $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh} \cup \mathcal{F}_{pop}$ is

1. s-monotonic iff f_s is monotonic
2. s-distributive iff f_s is distributive

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Properties

Lemma (14.2.1.8)

For all $e \in E^*$ *the function* $[\![e]\!]^*$ *is s-monotonic* (*s-distributive*), *if*

- $e \in E^* \backslash E_r^*$: $[\![e]\!]'$ *is monotonic* (*distributive*)
- $e \in E_r^*$: $[\![e]\!]'$ *and* $\mathcal{R}_e$ *are monotonic* (*distributive*)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Conventions

In the following

- we consider s-monotonicity and s-distributivity as generalizations of the usual monotonicity and distributivity.

To this end, we

- identify lattice elements with their representation as a DFA stack with just a single entry.
- extend the meet and join operation $\sqcap$ and $\sqcup$ as follows to (the top most entries of) DFA stacks:

$$\bigsqcap STK =_{df} \text{newstack}(\bigsqcap\{top(stk) \mid stk \in STK\})$$

$$\bigsqcup STK =_{df} \text{newstack}(\bigsqcup\{top(stk) \mid stk \in STK\})$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.2.2

The $IMOP_{Stk}$ Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The $IMOP_{Stk}$ Approach

The $IMOP_{Stk}$ Solution:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMOP_{Stk\,c_{\mathbf{s}}}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!]^{*}(\mathrm{newstack}(c_{\mathbf{s}})) \mid p \in \mathbf{IP}[\mathbf{s}, n] \,\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.2.3

The $IMaxFP_{Stk}$ Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The $IMaxFP_{Stk}$ Approach

...is a two-stage approach:

- Stage 1: Preprocess – Computing the Semantics of Procedures
- Stage 2: Main Process – Computing the *IMaxFP* solution

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Preliminaries

Let

- Id_{STACK} denote the identity on $STACK$, and
- $\sqcap$ 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 $\forall\, stk \in STACK.\ \ topl(f''(stk)) = \mathrm{top}(f(stk)) \sqcap \mathrm{top}(f'(stk))$.
- "$\sqcap$" induces an inclusion relation "$\sqsubseteq$" on $\mathcal{F}_{ord}$ by: $f \sqsubseteq f'$ gdw. $f \sqcap f' = f$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The $IMaxFP_{Stk}$ Approach (1)

Stage 1: Preprocess – Computing the Semantics of Procedures:

The 2nd Order Equation System 14.2.3.1

$$[\![n]\!] = \begin{cases} Id_{STACK} & \text{if } n \in start(S) \\ \bigsqcap\{[\![(m,n)]\!] \circ [\![m]\!] \mid m \in pred_{flowGraph(n)}(n)\} & \\ & \text{otherwise} \end{cases}$$

and

$$[\![e]\!] = \begin{cases} [\![e]\!]^* & \text{if } e \in E \backslash E_{call} \\ [\![e_r]\!]^* \circ [\![end(callee(e))]\!] \circ [\![e_c]\!]^* & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The $IMaxFP_{Stk}$ Approach (2)

The 1st Order $IMaxFP_{Stk}$ Equation System 14.2.3.2:

$$inf(n) = \begin{cases} \text{newstack}(c_{\mathbf{s}}) & \text{if } n = \mathbf{s}_0 \\ \bigsqcap \{ [\![e_c]\!]^*(inf(src(e))) \mid e \in caller(flowGraph(n)) \} & \\ & \text{falls } n \in start(S) \setminus \{\mathbf{s}_0\} \\ \bigsqcap \{ [\![(m,n)]\!](inf(m)) \mid m \in pred_{flowGraph(n)}(n) \} & \\ & \text{otherwise} \end{cases}$$

The $IMaxFP_{Stk}$ Solution:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{Stk\,c_{\mathbf{s}}}(n) =_{df} inf^*_{c_{\mathbf{s}}}(n)$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.2.4

Main Results

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Towards the Main Results

Important:

Lemma (14.2.4.1)

For all $n \in N$ the semantic functions $[\![e]\!]^*$, $e \in E^*$, are

1. *s-monotonic:* $[\![n]\!] \sqsubseteq imop_n$
2. *s-distributive:* $[\![n]\!] = imop_n$

where $imop_n : N \to (STACK \to STACK)$ denotes a functional that is defined by:

$$\forall\, n \in N.\ imop_n =_{df} \begin{cases} Id_{STACK} & \text{if } n \in start(S) \\ \bigsqcap\{[\![p]\!]^* \mid p \in \mathbf{CIP}[start(flowGraph(n)), n]\} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Main Results – 1st Stage

Safety and coincidence results of the 2nd order 1st stage analysis:

Theorem (14.2.4.2, 2nd-Order)

For all $e \in E_{call}$ *we have:*

1. $[\![e]\!] \sqsubseteq \bigsqcap\{[\![p]\!]^* \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, *if* $[\![\ ]\!]^*$ *is s-monotonic.*
2. $[\![e]\!] = \bigsqcap\{[\![p]\!]^* \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, *if* $[\![\ ]\!]^*$ *is s-distributive.*

where the mappings *src* and *dst* yield the start and final node of an edge.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Main Results – 2nd Stage

Safety and coincidence results of the 1st order 2nd stage analysis:

Theorem (14.2.4.3, Interprocedural Safety)

The $IMaxFP_{Stk}$ *solution is a lower (i.e., safe) approximation of the* $IMOP_{Stk}$ *solution, i.e.*

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{Stk\,c_s}(n) \sqsubseteq IMOP_{Stk\,c_s}(n)$$

if $[\![\ ]\!]^*$ *is s-monotonic.*

Theorem (14.2.4.4, Interprocedural Coincidence)

The $IMaxFP_{Stk}$ *solution coincides with the* $IMOP_{Stk}$ *solution, i.e.*

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{Stk\,c_s}(n) = IMOP_{Stk\,c_s}(n)$$

if $[\![\ ]\!]^*$ *is s-distributive.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.2.5

Algorithms

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Algorithms

- The algorithms of Chapter 14.1.5 can straightforwardly be extended to stack-based functions.
- This way we receive
  - the standard variant of pre- and post-process
  - the more efficient variant of pre- and functional main process.
  - a demand-driven "by-need" variant.
- In the following we present another stackless variant. 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 is sufficient for the implementation.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 2nd Order Algorithm 14.2.5.1 – 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 $[\![\ ]\!]' : E^* \to (\mathcal{C} \to \mathcal{C})$.

Output: Under the assumption of termination (cf. Theorem 14.2.5.4), an annotation of S with functions $[\![\![\, n \,]\!]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *gtr*, which stands for *global transformation*), and $[\![\, e \,]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *ltr*, which stands for *local transformation*) representing the greatest solution of Equation System 14.2.3.1.

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 $[\![\![\ ]\!]\!]$ and $[\![\ ]\!]$ the iterative approximation of $[\![\![\ ]\!]\!]$ is superposed by an interprocedural iteration step, which updates the current approximation of the effect $[\![\ ]\!]$ of call edges. The temporary *meet* stores the result of the most recent meet operation.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 2nd Order Algorithm 14.2.5.1 – Preprocess

(Prologue: Initialization of the annotation arrays *gtr* and *ltr* and the variable *workset*)

FORALL $n \in N$ DO
    IF $n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ THEN $gtr[n] := Id_{\mathcal{C}}$
                ELSE $gtr[n] := \top_{[\mathcal{C} \to \mathcal{C}]}$ FI OD;

FORALL $e \in E$ DO
    IF $e \in E_{call}$ THEN $ltr[e] := [\![\, e_r \,]\!]' \circ \top_{[\mathcal{C} \to \mathcal{C}]} \circ [\![\, e_c \,]\!]'$
                ELSE $ltr[e] := [\![\, e \,]\!]'$ FI OD; $\langle\star\rangle$

$workset := \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 2nd Order Algorithm 14.2.5.1 – Preprocess

```
( Main process: Iterative fixed point computation )
WHILE workset ≠ ∅ DO
    CHOOSE m ∈ workset;
        workset := workset \ { m };
        ( Update the successor-environment of node m )
        IF m ∈ {e_1, ..., e_k}
            THEN
                FORALL e ∈ caller(flowGraph(m)) DO
                    ltr[e] := R_e ∘ (Id_C, [[ e_r ]]' ∘ gtr[m] ∘ [[ e_c ]]');     ⟨⋆⟩
                    meet := ltr[e] ∘ gtr[src(e)] ⊓ gtr[dst(e)];
                    IF gtr[dst(e)] ⊐ meet
                        THEN
                            gtr[dst(e)] := meet;
                            workset := workset ∪ {dst(e)}
                    FI
                OD
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 2nd Order Algorithm 14.2.5.1 – Preprocess

```
            ELSE ( i.e., m ∉ {e_1, ..., e_k} )
                FORALL n ∈ succ_flowGraph(m)(m) DO
                    meet := ltr[(m, n)] ∘ gtr[m] ⊓ gtr[n];
                    IF gtr[n] ⊐ meet
                        THEN
                            gtr[n] := meet;
                            workset := workset ∪ {n}
                    FI
                OD
        FI
    ESOOHC
OD.
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 1st Order Algorithm 14.2.5.2 – 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 $[\![\ ]\!]$ computed by Algorithm 14.5.2.1, and (3) a context information $c_{\mathsf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 14.5.2.4), the $IMaxFP_{StkLss}$-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is *distributive*: variable *inf* stores for every node the strongest component information valid there with respect to the context information c_{s}.
(2) $[\![\ ]\!]$ 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 1st Order Algorithm 14.2.5.2 – Main Process

(Prologue: Initialization of the annotation array *inf* and the variable *workset*)

FORALL $n \in N \backslash \{\mathbf{s}_0\}$ DO $inf[n] := \top$ OD;

$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;

$workset := \{\, \mathbf{s}_0 \,\}$;

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

  CHOOSE $m \in workset$;

    $workset := workset \backslash \{\, m \,\}$;

    (Update the successor-environment of node m)

    FORALL $n \in succ_{flowGraph(m)}(m)$ DO

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

      IF $inf[n] \sqsupset meet$

        THEN

          $inf[n] := meet$;

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 1st Order Algorithm 14.2.5.2 – Main Process

IF $(m, n) \in E_{call}$
THEN
$meet := [\![(m, n)_c]\!]'(inf[m]) \sqcap$
$inf[start(callee((m, n)))]$; $\langle\star\rangle$
IF $(m, n) \in E_{call}$
THEN
$meet := [\![(m, n)_c]\!]'(inf[m]) \sqcap$
$inf[start(callee((m, n)))]$; $\langle\star\rangle$
IF $inf[start(callee((m, n)))] \sqsupset meet$
THEN
$inf[start(callee((m, n)))] := meet$;
$workset := workset \cup \{ start(callee((m, n))) \}$
FI
FI
OD
ESOOHC OD.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 1st Order Algorithm 14.2.5.3 – "Functional" 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 $[\![\ ]\!]=_{df} gtr$ and $[\![\ ]\!]=_{df} ltr$ with respect to $\mathcal{C}$ (computed by Algorithm 14.5.2.1), and (4) a context information $c_{\mathsf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 14.5.2.4), the $IMaxFP_{StkLss}$-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is distributive: variable *inf* stores for every node the strongest component information valid there with respect to the context information c_{s}.
(2) $[\![\ ]\!]$ 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 1st Order Algorithm 14.2.5.3 – "Functional" Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)
FORALL $\mathbf{s} \in \{\mathbf{s}_i \mid i \in \{1, \ldots, k\}\}$ DO $\mathit{inf}[\mathbf{s}] := \top$ OD;
$\mathit{inf}[\mathbf{s}_0] := c_{\mathbf{s}}$;
$\mathit{workset} := \{\mathbf{s}_i \mid i \in \{1, 2, \ldots, k\}\}$;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 1st Order Algorithm 14.2.5.3 – "Functional" Main Process

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

  CHOOSE $\mathbf{s} \in workset$;

    $workset := workset \setminus \{\mathbf{s}\}$;

    $meet := inf[\mathbf{s}] \sqcap$

      $\bigsqcap \{ [\![e_c]\!]' \circ [\![src(e)]\!](inf[start(flowGraph(e))]) \mid$

        $e \in caller(flowGraph(\mathbf{s})) \}$; $\langle \star \rangle$

    IF $inf[\mathbf{s}] \sqsupset meet$

      THEN

        $inf[\mathbf{s}] := meet$;

        $workset := workset \cup$

          $\{start(callee(e)) \mid e \in E_{call}.$

            $flowGraph(e) = flowGraph(\mathbf{s}) \}$

    FI

  ESOOHC

OD;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

The Stackless 1st Order Algorithm 14.2.5.3 – "Functional" Main Process

(Epilogue)

FORALL $n \in N \backslash \{\mathbf{s}_i \mid i \in \{0, \ldots, k\}\}$ DO

$\quad inf[n] := [\![n]\!](inf[start(flowGraph(n))])$

OD.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Termination

Theorem (14.2.5.4, Termination)

The sequential composition of Algorithm 14.2.5.1 and Algorithmus 14.2.5.2 resp. Algorithm 14.2.5.3 terminates with the IMaxFP$_{Stk}$ solution, if the DFA functional $[\![\ ]\!]'$ and the return functional $\mathcal{R}$ are monotonic and the lattice of functions $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition.

Note: If $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition, then $\mathcal{C}$ does so as well.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.3

Extensions

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Extensions

- Further parameter transfer mechanisms
  - Reference parameters
  - Procedural parameters, for short: procedure parameters
- Static nesting of procedures

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Reference Parameters

Intuitively:

- The effect of reference parameters is encoded in the local semantic functionals of the application problems.
- Reference parameters can thus be handled and computed by suitable preprocess computing may and must aliases of variables and parameters.
- The computed alias information is then fed into the definitions of the local semantics functions of the application problems (cf. Chapter 14.4)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Procedure Parameter

Intuitively:

- A formal procedure call is replaced by the set of all ordinary procedure calls that it may call.
- This set of procedures can be computed by a suitable preprocess; dependingy on the program or programming language class this can be either a safe approximation or an exact solution.
- The computed calling information for formal procedure call reduces then the analysis of programs w/ formal procedure calls to the analysis of programs w/out formal procedure calls.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Static Procedure Nesting

Various variants are possible:

- De-nesting of procedures by a suitable preprocess; this way the analysis of programs w/ static procedure nesting is reduced to analysing programs w/out static procedure nesting.
- Taking into account the effect of relatively global variables in the definition of the local semantics functions of the application problems.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 14.4

Applications

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Preliminaries

In the following we assume:

- No static procedure nesting, no procedure 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Notations (1)

- $GlobVar(S)$: 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 $Var(t) \setminus GlobVar(S)$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Notations (2)

- $NoGlobalChanges : E^* \to \mathbb{B}$: indicates that if a variable $v \in Var(t)$ is modified by e, then this modification will not be visible after finishing the call as the relevant memory location of v is local for the currently active call.
- $PotAccessible : S \to \mathbb{B}$: indicates that the memory locations of all variables $v \in Var(t)$, 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Local Predicates

The definition of the preceding functions relies on the predicates $Transp_{LocId}$ and $Transp_{GlobId}$ that are defined as follows:

$$Transp_{LocId}(e) =_{df} LocId(t) \cap MayAliases_{flowGraph(e)}(LhsVar(e)) = \emptyset$$

$$Transp_{GlobId}(e) =_{df} GlobId(t) \cap (LhsVar(e) \cup MayAliases_{flowGraph(e)}(LhsVar(e))) = \emptyset$$

This allows us to define:

$$\forall\, e \in E^*.\ NoGlobalChanges(e) =_{df} \begin{cases} \textbf{true} & \text{if } e \in E_c^* \cup E_r^* \\ Transp_{LocId}(n) \wedge Transp_{GlobId}(n) & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Parameterized Local Predicates (1)

...parameterized wrt alias information:

$$\forall\, e \in E^*.\ A\text{-}Comp_{\,e} =_{df} Comp_{\,e} \ \vee\ Comp_{\,e}^{MstAl}$$

$$\forall\, e \in E^*.\ A\text{-}Transp_{\,e} =_{df} Transp_{\,e} \ \wedge \begin{cases} \textbf{true} & \text{if } e \in E^*_{call} \\ Transp_{\,e}^{MayAl} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Parameterized Local Predicates (2)

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., $Comp_e^{MstAl}$).
- $A\text{-}Transp_e$, $e \in E^* \backslash E^*_{call}$, is true, 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., $Transp_e^{MayAl}$).
- For call and return edges $e \in E^*_{call}$, $A\text{-}Transp_e$ is true, 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Remark

- $\mathcal{B}_X =_{df} \{\textbf{false}, \textbf{true}, \textit{failure}\}$

Note: The element *failure* is introduced as an artifical $\top$-element in $\mathbb{B}$ in order to be prepared for reverse data flow analysis as required for demand-driven data flow analysis (cf. LVA 185.276 Analysis and Verification).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Availability (1)

Local Abstract Semantics:

1. Data flow lattice:
$$(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\, \mathcal{B}_X^2, \wedge, \vee, \leq, (\mathbf{false}, \mathbf{false}), (failure, failure)\,)$$
2. Data flow functional: $[\![\ ]\!]'_{av} : E^* \rightarrow (\, \mathcal{B}_X^2 \rightarrow \mathcal{B}_X^2 \,)$ defined by
$$\forall\, e \in E^*\ \forall\, (b_1, b_2) \in \mathcal{B}_X^2.\ [\![\, e \,]\!]'_{av}(b_1, b_2) =_{df} (b_1', b_2')$$
where
$$b_1' =_{df} A\text{-}Transp_e \wedge (A\text{-}Comp_e \vee b_1)$$
$$b_2' =_{df} \begin{cases} b_2 \wedge NoGlobalChanges_e & \text{if } e \in E^* \backslash E_c^* \\ \mathbf{true} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Availability (2)

3. Return functional: $\mathcal{R}_{av} : E_{call} \rightarrow (\, \mathcal{B}_X^2 \times \mathcal{B}_X^2 \rightarrow \mathcal{B}_X^2 \,)$
defined by $\forall\, e \in E_{call}\ \forall\, (\,(b_1, b_2),\, (b_3, b_4)\,) \in \mathcal{B}_X^2 \times \mathcal{B}_X^2.$
$\mathcal{R}_{av}(e)(\,(b_1, b_2),\, (b_3, b_4)\,) =_{df} (b_5, b_6)$ where

$$b_5 =_{df} \begin{cases} b_3 & \text{if } PotAccessible(callee(e)) \\ (b_1 \vee A\text{-}Comp_e) \wedge b_4 & \text{otherwise} \end{cases}$$

$$b_6 =_{df} b_2 \wedge b_4$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Availability (3)

Lemma (14.4.1)

1. *The lattice $\mathcal{B}_X^2$ and the induced lattice of functions satisfy the descending chain condition.*
2. *The functionals $[\![\ ]\!]'_{av}$ and $\mathcal{R}_{av}$ are distributive.*

$\leadsto$ Hence, the preconditions of the Interprocedural Coincidence Theorem 14.2.4.4 and the Termination Theorem 14.2.5.4 are satisfied.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Simple Constants

Local Abstract Semantics:

1. Data flow lattice:
   $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\Sigma_X, \sqcap, \sqcup, \sqsubseteq, \sigma_\bot, \sigma_{failure})$
2. Data flow functional: $[\![\ ]\!]'_{sc} : E \to (\Sigma_X \to \Sigma_X)$ defined by
   $\forall\, e \in E.\ [\![\, e \,]\!]'_{sc} =_{df} \theta_e$
3. Return functional: $\mathcal{R}_{sc} : E_{call} \to (\Sigma_X \times \Sigma_X \to \Sigma_X)$
   defined by

   $$\forall\, e \in E_{call}\ \forall\, (\sigma_1, \sigma_2)) \in \Sigma_X \times \Sigma_X.\ \mathcal{R}_{sc}(e)(\sigma_1, \sigma_2) =_{df} \sigma_3$$

   where

   $$\forall\, x \in Var.\ \sigma_3(x) =_{df} \begin{cases} \sigma_2(x) & \text{if } x \in GlobVar(S) \\ \sigma_1(x) & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Problems and Solutions/Work-Arounds

In practice

- the preceding analysis specification for simple interprocedural constants does not induce a terminating analysis since the lattice of functions does not satisfy the descending chain condition
- thus simpler constant propagation problems are considered like copy constants and linear constants.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Copy Constants and Linear Constants

A term is a

- copy constant at a program point, if it is a source-code constant or an operator-less term that is itself a copy constant
- linear constant at a program point, if it is a source-code constant or of the form $a * x + b$ w/ a, b source-code constants and x a linear constant.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Copy Constants (1)

The specification of copy constants is based on the following simpler evaluation function of terms:

$$\mathcal{E}_{cc} : \mathbf{T} \to (\Sigma_X \to \mathbf{D})$$

$\mathcal{E}_{cc}$ is undefined for the failure state $\sigma_{failure}$; otherwise it is defined as follows:

$$\forall\, t \in \mathbf{T}\ \forall\, \sigma \in \Sigma.\ \mathcal{E}_{cc}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t = x \in \mathbf{V} \\ I_0(c) & \text{if } t = c \in \mathbf{C} \\ \bot & \text{otherwise} \end{cases}$$

Note that Σ_X is analogously to $\mathcal{B}_X$ extended by an artificial top-element.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Copy Constants (2)

- Replacing θ_ι in $\mathcal{E}$ by $\mathcal{E}_{cc}$ yields the data flow analysis functional $[\![\ ]\!]'_{cc}$.
- Replacing of $[\![\ ]\!]'_{sc}$ by $[\![\ ]\!]'_{cc}$ yields the definition of the local abstract semantics of interprocedural copy constants.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Copy Constants (3)

Note:

- The number of source-code constants is finite.
- Hence, the lattice of functions that belongs to the relevant sublattice Σ_{cc_X} of Σ_X satisfies the descending chain condition.
- Thus, the *IMaxFP* algorithm terminates.
- Unlike as interprocedural simple constants are copy constants distributive; thus, the $IMaxFP_{Stk}$ solution and the $IMOP_{Stk}$ solution coincide.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural Copy Constants (4)

Lemma (14.4.2)

1. *The lattice Σ_{cc_X} and the induced lattice of functions satisfy the descending chain condition.*
2. *The functionals $[\![\ ]\!]'_{cc}$ and $\mathcal{R}_{cc}$ are distributive.*

Therefore, the preconditions of the Interprocedural Coincidence Theorem 14.2.4.4 and the Termination Theorem 14.2.5.4 are satisfied.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

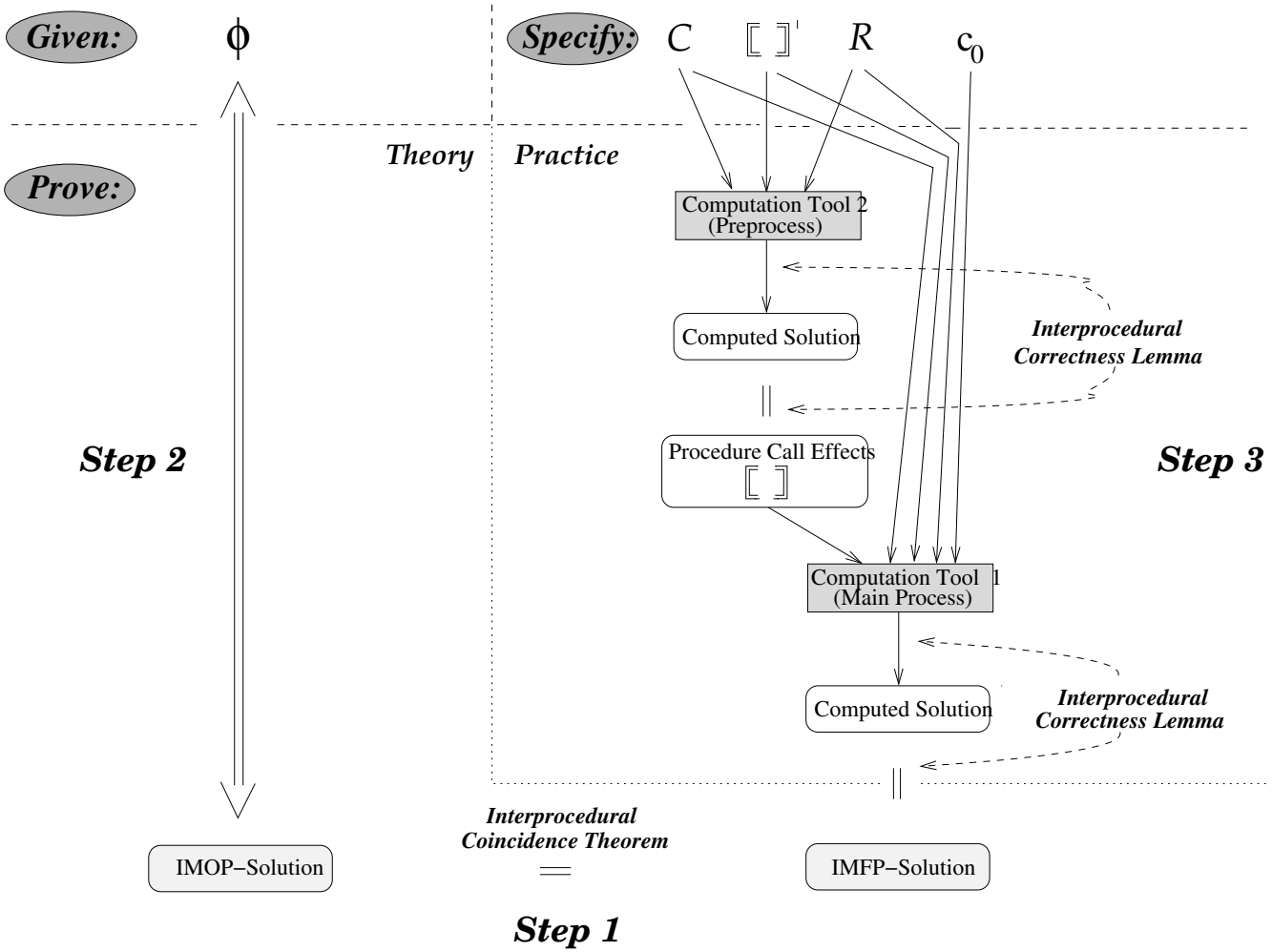
Chapter 14.5

Interprocedural DFA: Framework and Toolkit

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural DFA: The Framework View

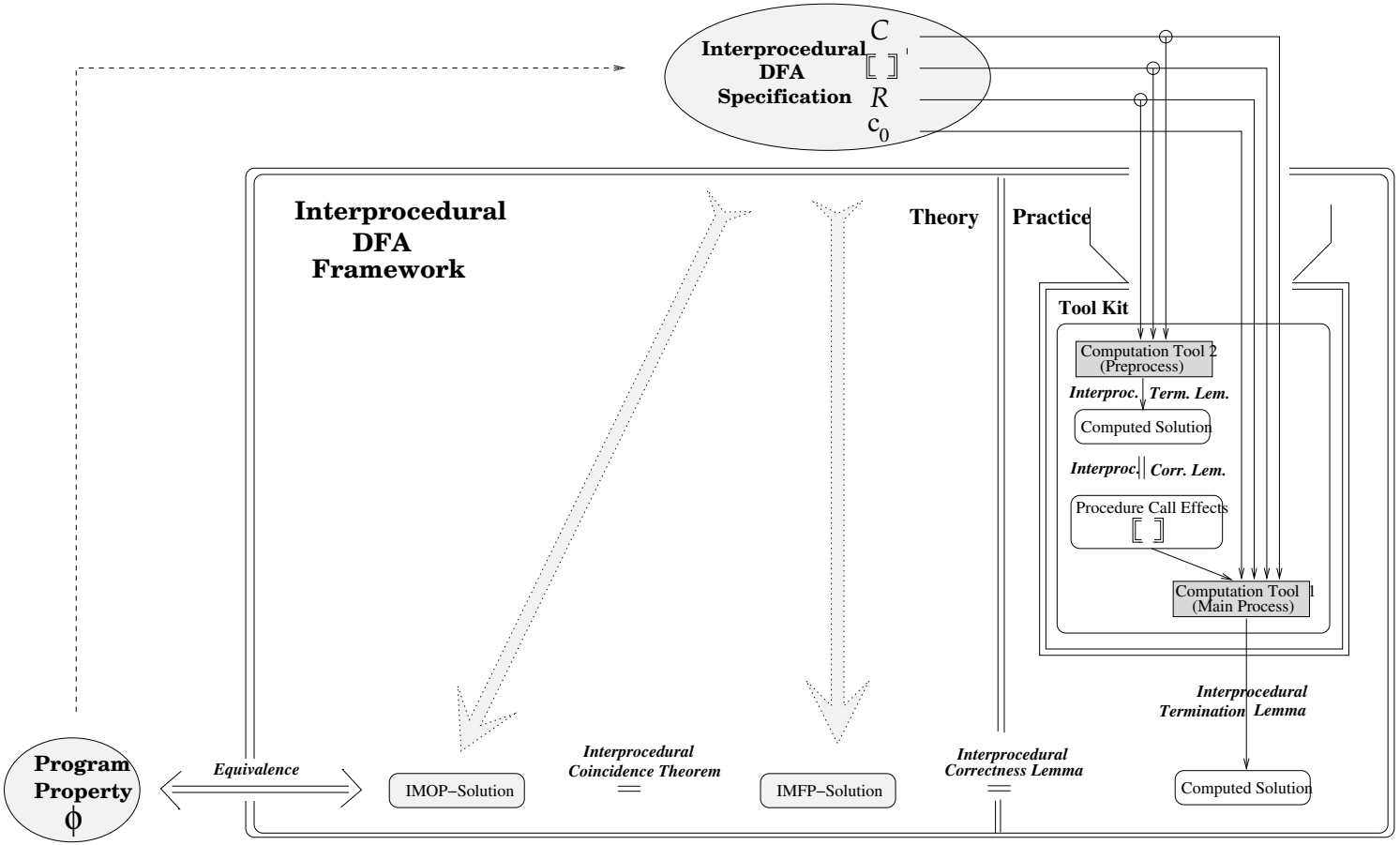
The interprocedural DFA Framework at a glance:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Interprocedural DFA: The Toolkit View

The Toolkit View of the interprocedural DFA Framework:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Further Reading for Chapter 14 (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)
- 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Further Reading for Chapter 14 (2)

- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Further Reading for Chapter 14 (3)

- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 19, Interprocedural Analysis and Optimization)
- Mooly Sagiv, Tom Reps, Susan Horwitz. *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation*. In Proceedings TAPSOFT'95, Springer-V., LNCS 915, 651-665, 1995.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Part IV

Extensions, Other Settings

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
14.1
14.1.1
14.1.2
14.1.3
14.1.4
14.1.5
14.1.6

Chapter 15

Aliasing

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Chapter 15.1

Sources of Aliasing

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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;
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Chapter 15.2

Relevance of Aliasing for Program Optimization

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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.

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Characterization of Aliasing

Flow-insensitive information

Binary relation on the variables in a procedure, $alias \in Var \times Var$ such that x *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 $|Loc| \leq 1$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

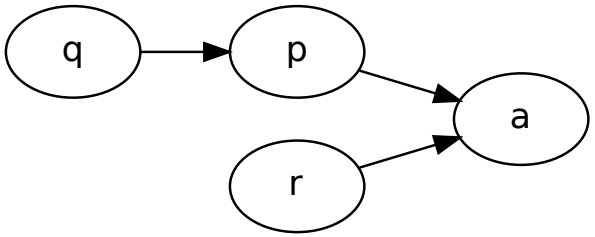
Representation of Alias Information

Representation of aliasing with pairs

	`q=&p; p=&a; r=&a;`
complete alias pairs	`<*q,p>, <*p,a>, <*r,a>,<**q,*p>, <**q,a>,<*p,*r>,<**q,*r>`
compact alias pairs	`<*q,p>, <*p,a>, <*r,a>`
points-to relations	`(q,p),(p,a),(r,a)`

Representation of aliases and shapes of data structures

- graphs
- regular expressions
- 3-valued logic



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Chapter 15.3

Shape Analysis

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Questions about Heap Contents (2)

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.

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Questions about Heap Contents (3)

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.

Reachability (Question 5)

Is a heap cell reachable from a specific variable or from any pointer variable?

- Yes (for every state). Information for program verification.
- No (for every state). Insert code at compile time that collects unreachable cells at run-time.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Questions about Heap Contents (4)

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Shape Analysis

Aim of Shape Analysis

The aim of shape analysis is to determine a finite representation of heap allocated data structures which can grow arbitrarily large.

It 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 PoPA Ch 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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:

- $[x := nil]^\ell$
- $[x.sel_1 := y.sel_2]^\ell$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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:

$$sel \in \mathsf{Sel} \qquad \text{selector names}$$

- we add a new syntactic category

$$p \in \mathsf{PExp} \qquad \text{pointer expressions}$$

- op_r is extended to allow for testing of equality of pointers
- unary operations op_p on pointers (e.g. is-null) are added

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Abstract Syntax of Pointer Language

The syntax of the while language is extended to have:

$$
\begin{array}{lcl}
p & ::= & x \mid x.sel \mid \text{null} \\
a & ::= & x \mid n \mid a_1 \; op_a \; a_2 \\
b & ::= & \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \; op_b \; b_2 \mid a_1 \; op_r \; a_2 \\
S & ::= & [\text{p:=a}]^\ell \mid [\text{skip}]^\ell \\
 & & \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \\
 & & \mid \text{while}[b]^\ell \text{ do } S \text{ od} \\
 & & \mid [\text{new } (p)]^\ell \\
 & & \mid S_1; S_2
\end{array}
$$

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Shape Graphs

We shall introduce a method for combining the locations of the semantics into a finite number of *abstract locations*.

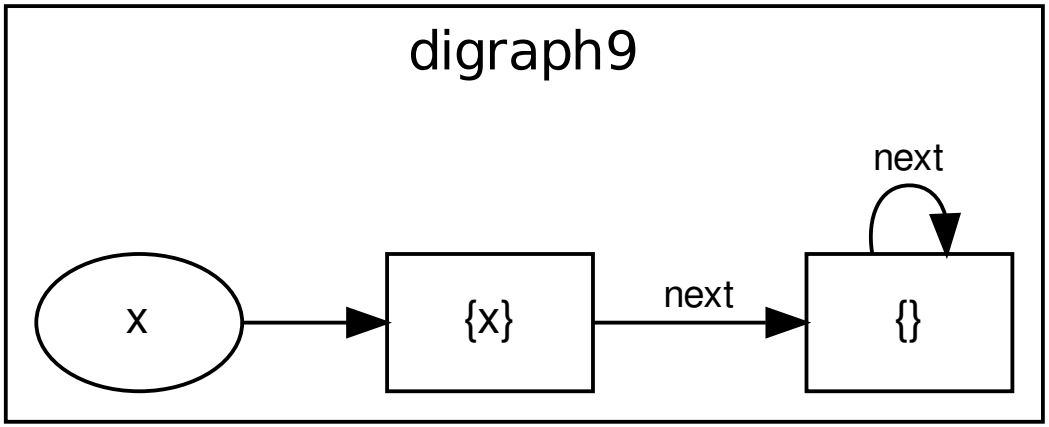
The analysis operates on shape graphs (S, H, is) consisting of:

- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Example



$g_9 = (\mathsf{S}, \mathsf{H}, \mathsf{is})$ where

$$\mathsf{S} = \{(\mathtt{x}, n_{\{\mathtt{x}\}})\}$$
$$\mathsf{H} = \{(n_{\{\mathtt{x}\}}, \mathtt{next}, n_{\emptyset}), (n_{\emptyset}, \mathtt{next}, n_{\emptyset})\}$$
$$\mathsf{is} = \emptyset$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Abstract Locations

The *abstract locations* have the form n_X where X is a subset of the variables of $\mathsf{Var}_\star$:

$$\mathsf{ALoc} = \{n_X \mid X \subseteq \mathsf{Var}_\star\}$$

A shape graph contains a subset of the abstract 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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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.

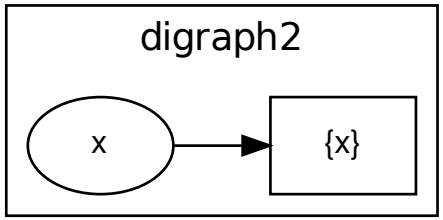
We shall only be interested in the shape of heap so we shall not distinguish between integer values, nil-pointers, and uninitialized fields; hence we can view the abstract state as an element of

$$\mathsf{S} \in \mathsf{AState} = \mathcal{P}\mathsf{Var}_\star \times \mathsf{ALoc}$$

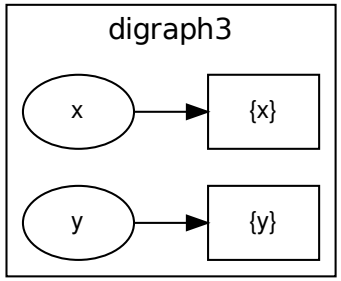
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Example: Creating Linked Data Structures

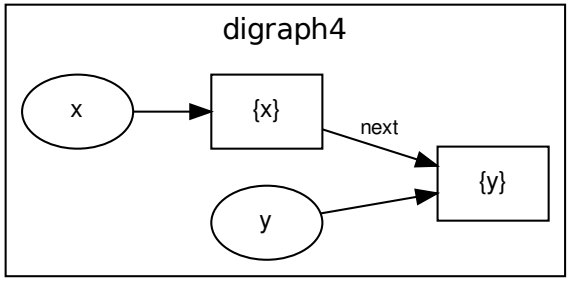
$[new(x)]^2$



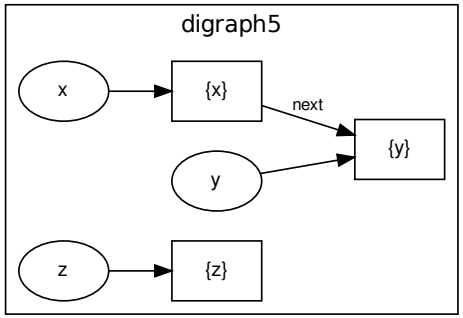
$[new(y)]^3$



$[\text{x.next} := \text{y}]^4$



$[new(z)]^5$



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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

$$\mathsf{H} \in \mathsf{AHeap} = \mathcal{P}\mathsf{ALoc} \times \mathsf{Sel} \times \mathsf{ALoc}$$

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 $n_\emptyset$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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.

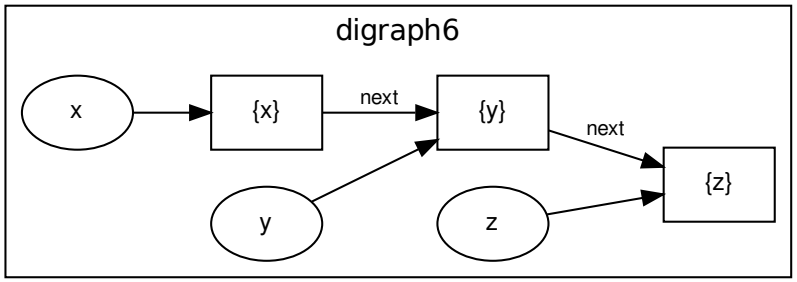
In the case of the abstract summary location, $n_\emptyset$, the explicit sharing information clearly gives extra information:

- if $n_\emptyset \in$ is then there might be a location represented by $n_\emptyset$ that is the target of two or more heap pointers.
- if $n_\emptyset \notin$ is then all the locations of represented by $n_\emptyset$ will be the target of at most one heap pointer.

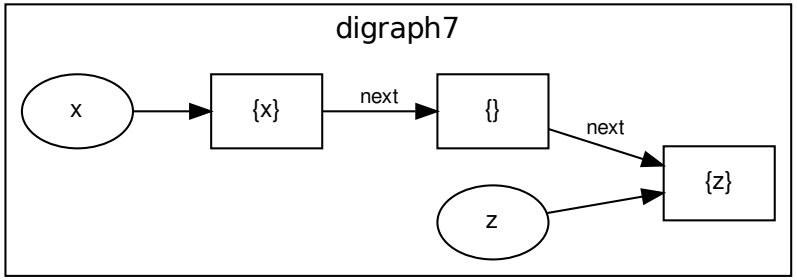
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Maintaining Sharing Information

$[y.next := z]^6$



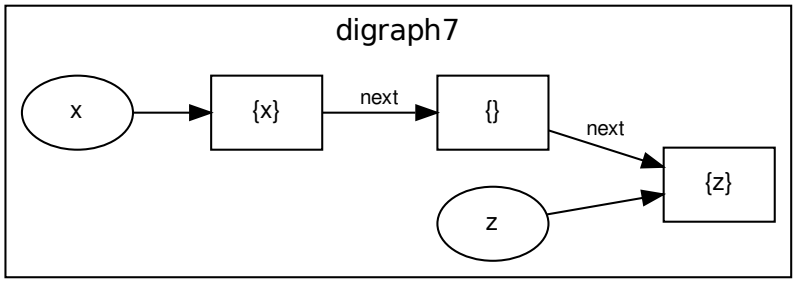
$[y := null]^7$



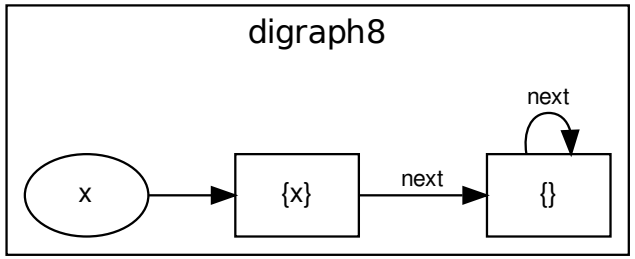
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Maintaining Sharing Information

$[y := \text{null}]^7$



$[z := \text{null}]^8$



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Sharing Information Invariants (1)

We shall impose two invariants to ensure that information in the sharing component is also reflected in the abstract heap.

The first 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Sharing Information Invariants (2)

The second invariant 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.

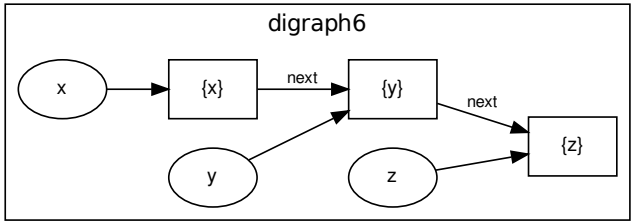
Note that invariant 5 is the "inverse" of invariant 4(b).

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.

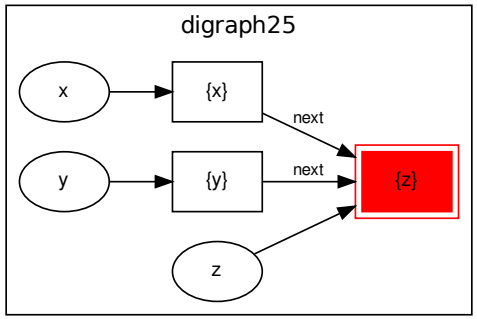
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Sharing Component Example 1

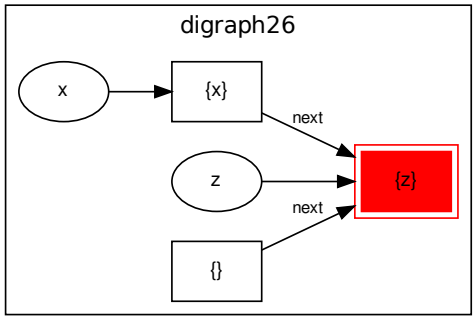
$[y.next := z]^6$



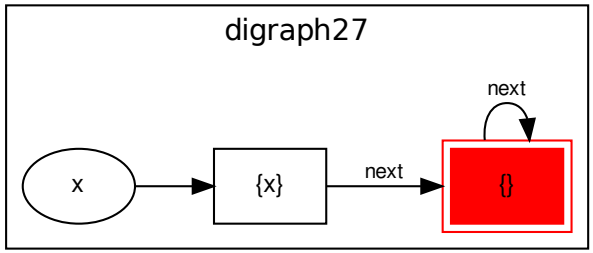
$[x.next := z]^{7'}$



$[y := null]^{8'}$



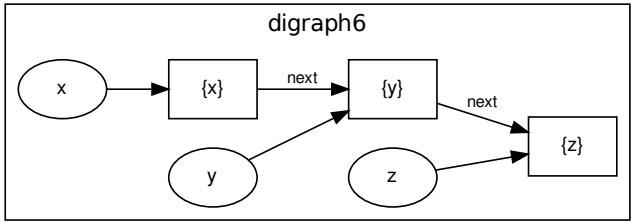
$[z := null]^{9'}$



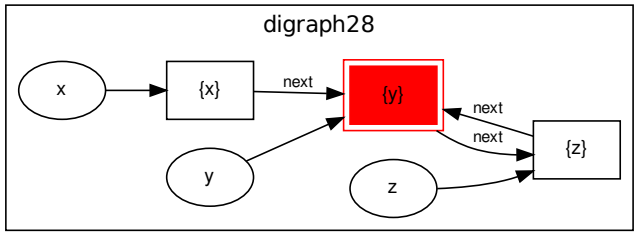
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Sharing Component Example 2

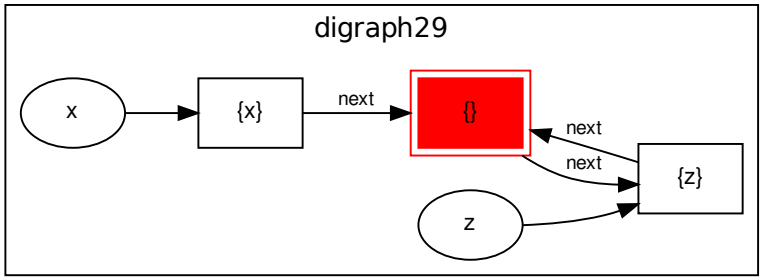
$[y.next := z]^{6}$



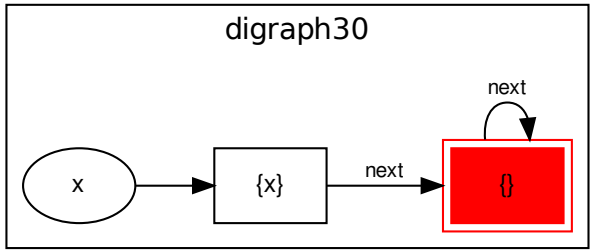
$[z.next := y]^{7''}$



$[y := null]^{8''}$



$[z := null]^{9''}$



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Compatible Shape Graphs

A shape graph is a triple $(\mathsf{S}, \mathsf{H}, \mathsf{is})$:

$$\begin{aligned} \mathsf{S} \in \mathsf{AState} &= \mathcal{P}\mathsf{Var}_\star \times \mathsf{ALoc} \\ \mathsf{H} \in \mathsf{AHeap} &= \mathcal{P}\mathsf{ALoc} \times \mathsf{Sel} \times \mathsf{ALoc} \\ \mathsf{is} \in \mathsf{IsShared} &= \mathcal{P}\mathsf{ALoc} \end{aligned}$$

where $\mathsf{ALoc} = \{n_X \mid X \subseteq \mathsf{Var}_\star\}$.

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

$$\mathsf{SG} = \{(\mathsf{S}, \mathsf{H}, \mathsf{is}) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \text{ is compatible}\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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

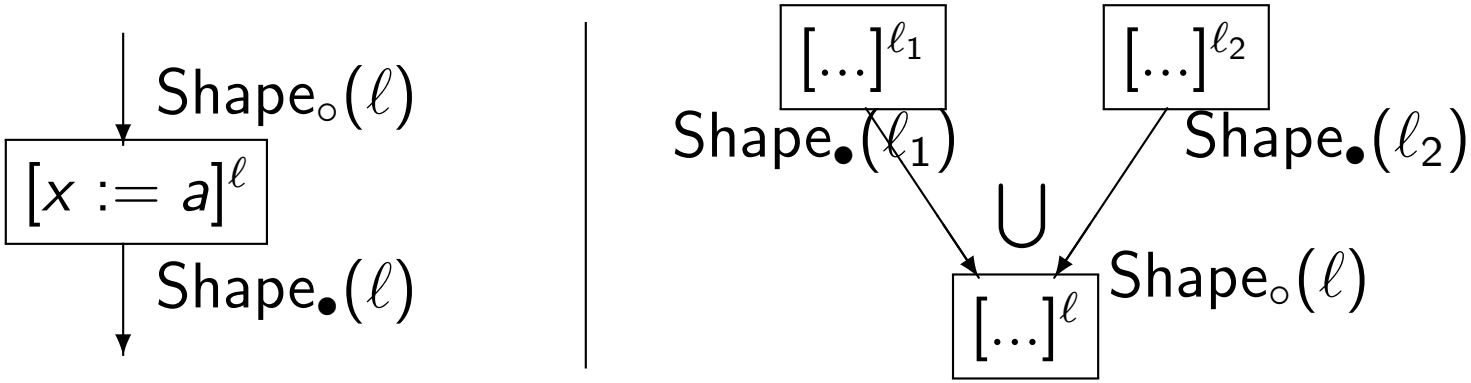
- ordering relation $\sqsubseteq$ being $\subseteq$
- combination operator $\sqcap$ being $\cup$ (may analysis)

$\mathcal{P}$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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Analysis



$$
\begin{array}{rcl}
\mathsf{Shape}_\circ(\ell) & = & \begin{cases} \iota & : \text{ if } \ell = \mathsf{init}(S_\star) \\ \bigcup\{\mathsf{Shape}_\bullet(\ell') \mid (\ell', \ell) \in \mathsf{flow}(S_\star)\} & : \text{ otherwise} \end{cases} \\
\mathsf{Shape}_\bullet(\ell) & = & f_\ell^{SA}(\mathsf{Shape}_\circ(\ell))
\end{array}
$$

where $\iota \in \mathcal{P}SG$ is the extremal value holding at entry to $S_\star$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Transfer Functions

The transfer function $f_\ell^{\mathsf{SA}} : \mathcal{P}\mathsf{SG} \to \mathcal{P}\mathsf{SG}$ has the form

$$f_\ell^{SA}(\mathsf{SG}) = \bigcup \{\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \in \mathsf{SG}\}$$

where ϕ_ℓ^{SA} specifies how a *single* shape graph (in $\mathsf{Shape}_\circ(\ell)$) may be transformed into a *set* of shape graphs (in $\mathsf{Shape}_\bullet(\ell)$.

The functions ϕ_ℓ^{SA} for the statements

x := a	x := y	x := y.sel
x.sel := a	x.sel := y	x.sel := y.sel

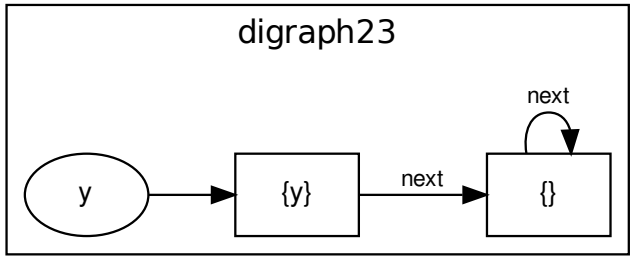
(illustrated by example)

transform a shape graph into a set of different shape graphs.

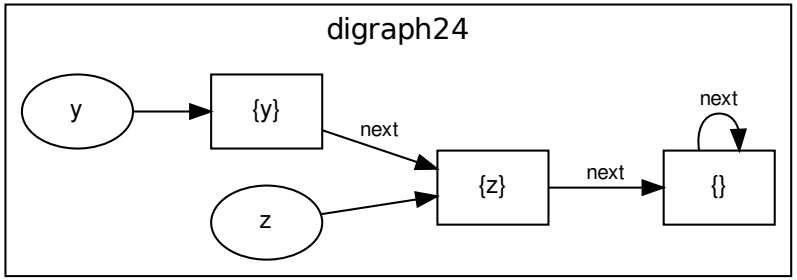
The transfer functions for other statements and expressions are specified by the identity function.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Example: Materialization



$[z := y.next]^7$



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

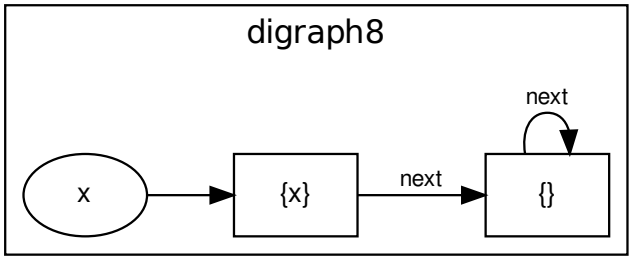
Example: Reverse List

$$
\begin{array}{l}
[\mathsf{y} := \mathsf{null}]^1; \\
\texttt{while } [\texttt{not isnull}(\texttt{x})]^2 \texttt{ do} \\
\quad [\mathsf{t} := \mathsf{y}]^3; \\
\quad [\mathsf{y} := \mathsf{x}]^4; \\
\quad [\mathsf{x} := \mathsf{x.next}]^5; \\
\quad [\mathsf{y.next} := \mathsf{t}]^6; \\
\mathsf{od} \\
[\mathsf{t} := \mathsf{null}]^7
\end{array}
$$

The program reverses the list pointed to by x and leaves the result in y.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Reverse List: Extremal Value

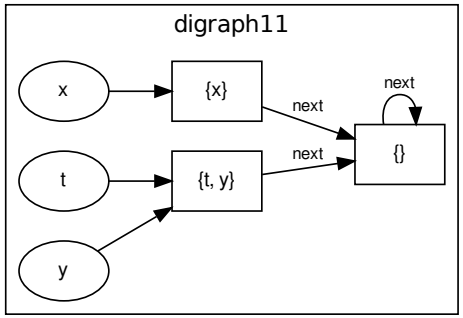


The extremal value ι is a set of graphs. The above graph is an element of this set for our example analysis of the list reversal program.

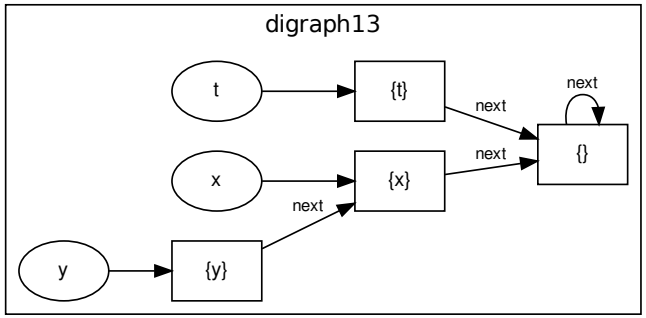
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Shape Graphs in Shape$_\bullet(\ell)$

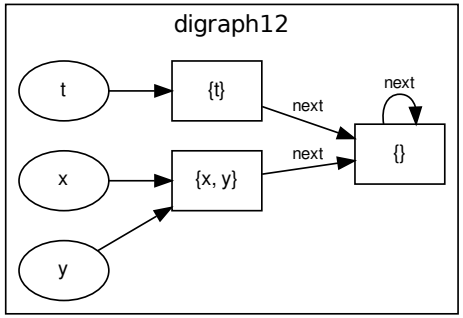
$[t := y]^3$



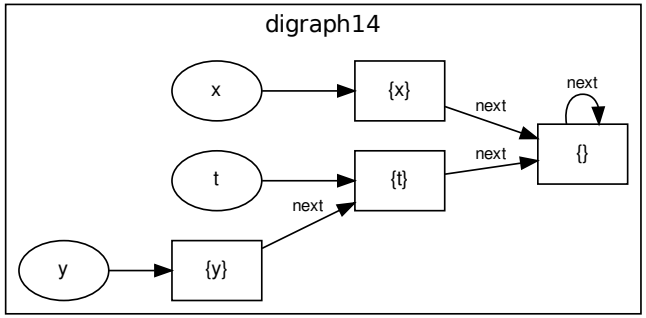
$[x := x.next]^5$



$[y := x]^4$



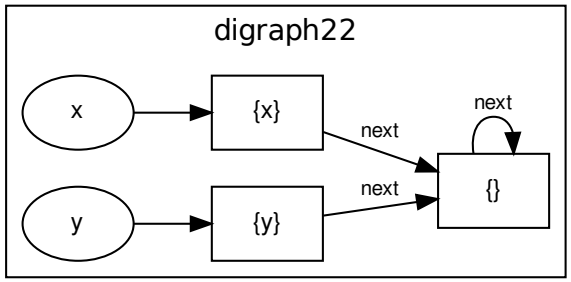
$[y.next := t]^6$



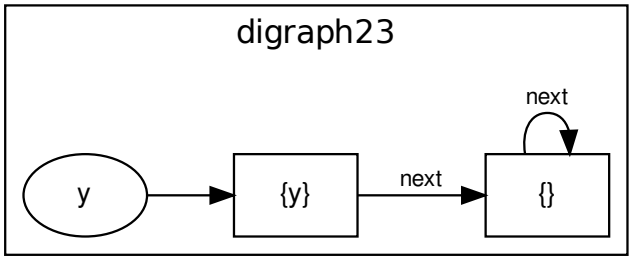
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Shape Graphs in $\text{Shape}_\bullet(\ell)$

$[\text{t} := \text{null}]^7$



$[\text{x} := \text{null}]^7$



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

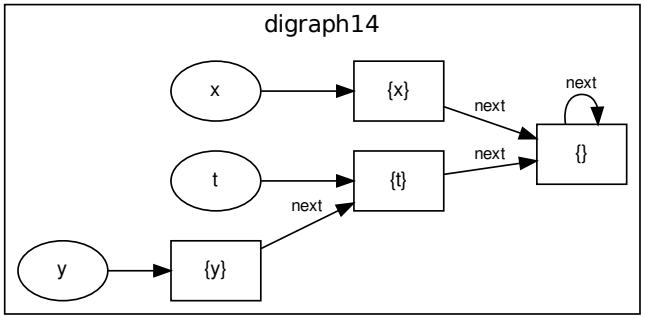
Reverse List: Established Properties

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.



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

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, CDH Ch 5]

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Further Reading for Chapter 15 (1)

- D. Chase, Mark N. Wegmann, F. Ken Zadeck. *Analysis of Pointers and Structures*. In Proceedings PLDI'90, 296-310, 1990.
- M. Emami, R. Ghiya, Laurie J. Hendren. *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*. In Proceedings PLDI'94, 1-14, 1994.
- R. Ghiya, Laurie J. Hendren. *Is it a Tree, a DAG, or a Cyclic Graph?* In Proceedings POPL'96, 1-15, 1996.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 10, Alias Analysis)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Further Reading for Chapter 15 (2)

- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 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), 2011. [Best Paper Award SCAM 2011]
- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-Analysis Problems in Languages with Destructive Updating*. ACM Transactions on Programming 20(1):1-50, 1998.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Further Reading for Chapter 15 (3)

- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. In Proceedings 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), 2002.
- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations & Machine Code Generation*. CRC Press, 2002. (Chapter 5, Shape Analysis and Application)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
15.1
15.2
15.3
Chap. 16

Chapter 16

Optimizations for Object-Oriented Languages

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Overview

- Object layout and method invocation
  - Single inheritance
  - Multiple Inheritance
- Devirtualization
  - Class hierarchy analysis
  - Rapid type analysis
  - Inlining
- Escape Analysis
  - Connection graphs
  - Intra-procedural
  - Inter-procedural

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.1

Object Layout and Method Invocation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

Chapter 16.1.1

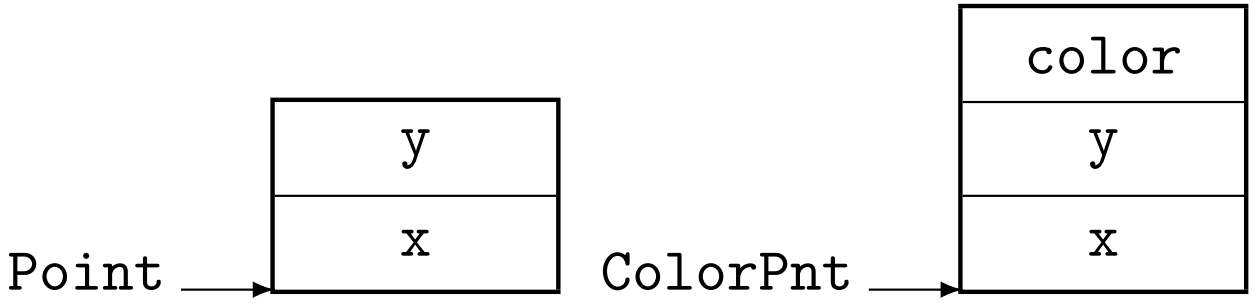
Single Inheritance

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

Single Inheritance Layout

```
class Point {
    int x, y;
}
```

```
class ColorPnt extends Point {
    int color;
}
```



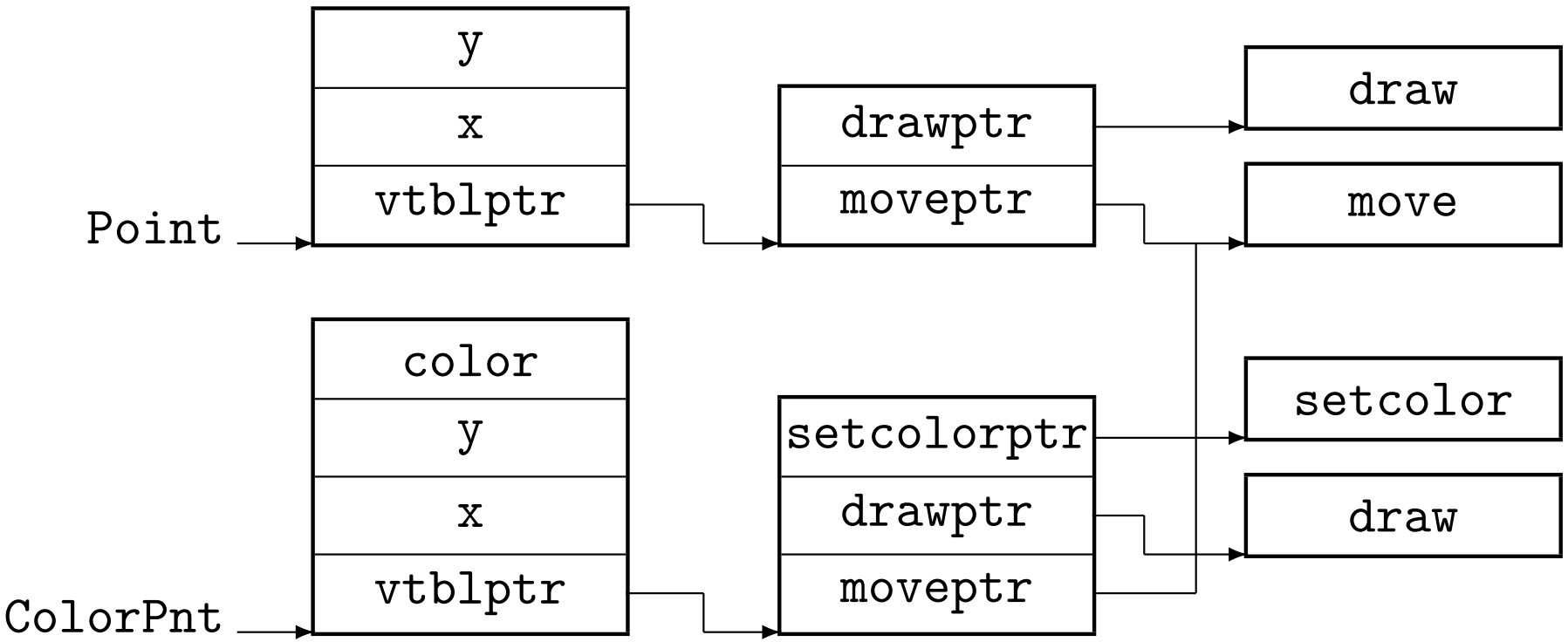
- Memory layout of an object of a superclass is a prefix of the memory layout of an object of the subclass.
- Instance variables access requires just one load or store instruction.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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) {...}
}
```



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

Invocation of Virtual Methods with vtbl

- Dynamic dispatching using a 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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.1.2

Multiple Inheritance

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Multiple Inheritance

- Extending the superclasses as in single inheritance does not work anymore.
- 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.

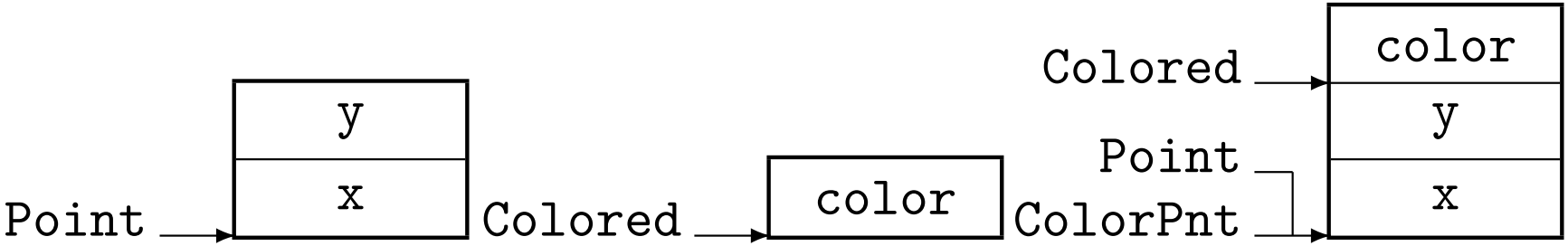
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Object Memory Layout (without vtbl)

```
class Point {
    int x, y;
}
```

```
class Colored {
    int color;
}
```

```
class ColorPnt extends Point, Colored {
}
```



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

Dynamic Dispatching for Embedding

- Allows fast access to instance variables exactly as with single inheritance.
- For every superclass
  - virtual method tables have to be created.
  - multiple *vtbl* 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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

Multiple Inheritance with vtbl (1)

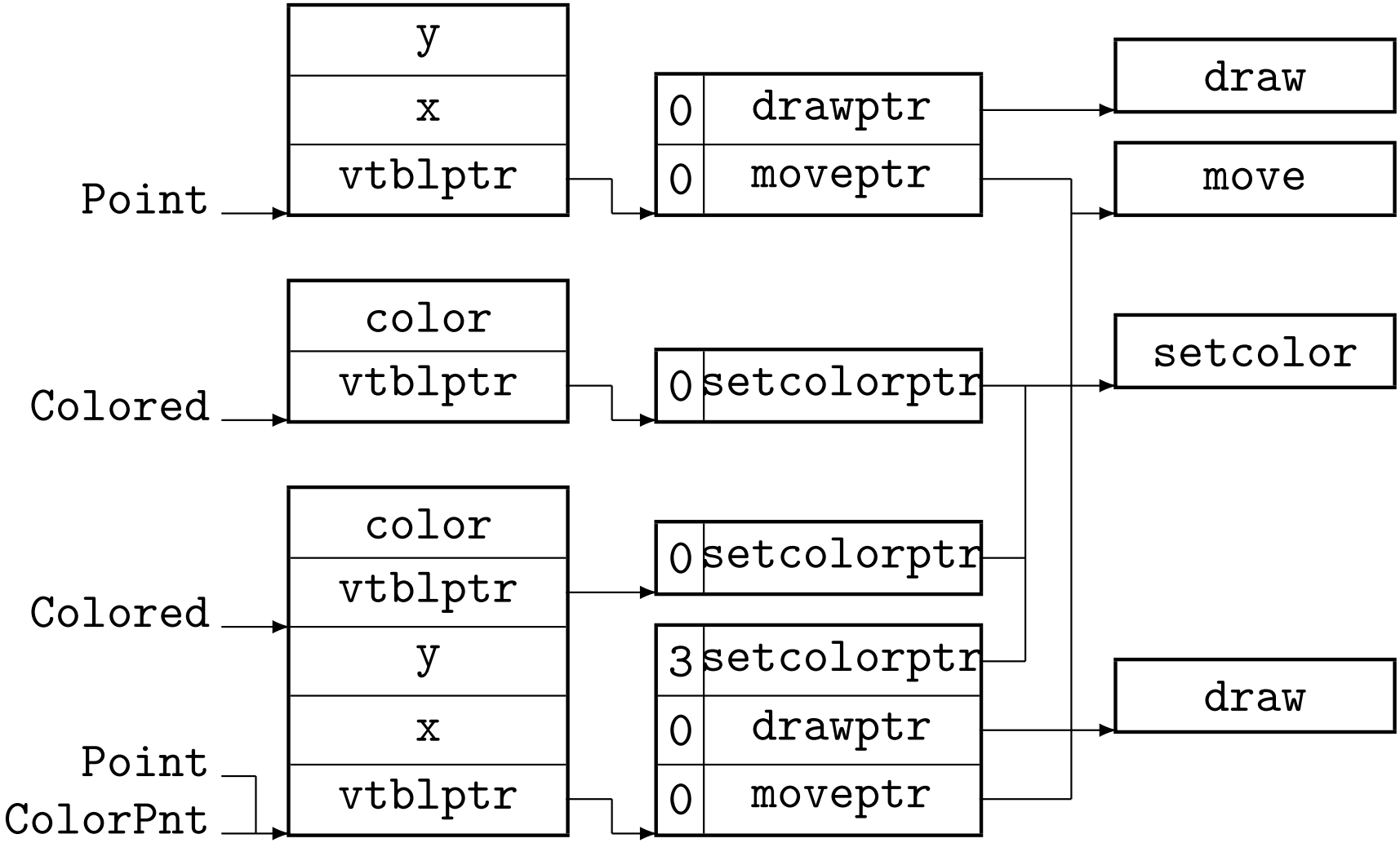
```
class Point {
    int x, y;
    void move(int x, int y) {...}
    void draw() {...}
    }

class Colored {
    int color;
    void setcolor(int c) {...}
    }

class ColorPnt extends Point, Colored {
    void draw() {...}
    }
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Multiple Inheritance with vtbl (2)



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

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 with 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.2
Devirtualization of Method Invocations

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.2.1

Class Hierarchy Analysis

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

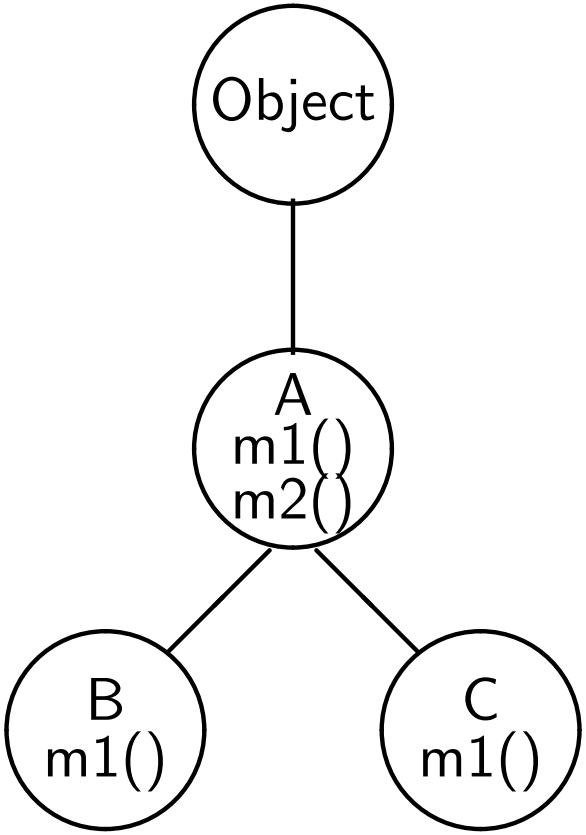
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

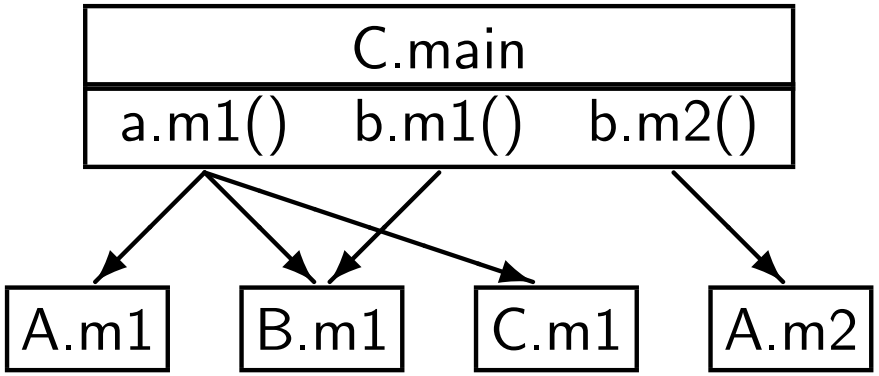
Example: Class Hierarchy Analysis

```
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
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Example: Class Hierarchy and Call Graph





Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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 $m \in callgraph$ **do**
  for each $m_{stat}()$ occuring in m **do**
    if $m_{stat} \notin callgraph$ **then**
      add m_{stat} to $callgraph$
  for each $e.m_{vir}()$ occuring in m **do**
    for each $c \in subtype(type(e))$ **do**
    $m_{def} := method(c, m_{vir})$
    if $m_{def} \notin callgraph$ **then**
      add m_{def} to $callgraph$
      add c to $hierarchy$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.2.2

Rapid Type Analysis

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Rapid Type Analysis (1)

Rapid type analysis 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.

- It 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.1.3

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.2.3

Inlining

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.3

Escape Analysis

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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 theron whorthwhile.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Escape States

The analysis uses a simple lattice to represent different escape states:

$$\begin{array}{c} \textit{NoEscape}\ (\top) \\ | \\ \textit{ArgEscape} \\ | \\ \textit{GlobalEscape}\ (\bot) \end{array}$$

State	Escapes the method	Escapes the thread
NoEscape	no	no
ArgEscape	may (via args)	no
GlobalEscape	may	may

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.3.1

Connection Graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Example: Connection Graphs

```
A a = new A();    // line L1
a.b1 = new B();   // line L2
a.b2 = a.b1;      // line L3
```

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.3.2

Intraprocedural Setting

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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 $byPass(p)$:

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Analyzing Statements (1)

p = new C(); // line L The operation *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 *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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 16.3.3

Interprocedural Setting

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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 $a_1, \ldots, a_n$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Further Reading for Chapter 16 (1)

- J-G Choi, M. Gupta, M. Serrano, V.C Sreedar, and Sam Midkiff. *Escape Analysis for Java*. In Proceedings of OOPSLA'99, ACM Press, 1-19, 1999.
- 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)
- H. D. Pande, Barbara Ryder. *Data-flow-based Virtual Function Resolution*. In Proceedings SAS'96, Springer-V., LNCS 1145, 238-254, 1996.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Further Reading for Chapter 16 (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)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
16.1
16.1.1
16.1.2
16.2

Chapter 17
Slicing

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Overview

Collection of Examples

- Definition of executable slice (liveness analysis)
- Example with scalar variables
- Example with pointers to stack-allocated variables
- Example with strong and weak update in alias analysis
- Example with non-cyclic dynamic data structures
- Example with cyclic data structures
- Comparison of slice size

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Reaching Defs with/out Pointers

$$
\begin{array}{l}
ReachingDefs(var, n, F) = \qquad\qquad\qquad\qquad\qquad\qquad [without\ pointers] \\
\quad let \\
\qquad F = (V, A, En, Ex),\ A' = A \backslash \{(x, n)\} \\
\quad return \bigcup_{(x,n) \in A} \ if\ var \notin def(x)\ then\ ReachingDefs(var, x, (V, A', En, Ex)) \\
\qquad\qquad\qquad else\ \{x\}
\end{array}
$$

$$
\begin{array}{l}
ReachingDefs(var, n, F) = \qquad\qquad\qquad\qquad\qquad\qquad [with\ pointers] \\
\quad let \\
\qquad F = (V, A, En, Ex),\ A' = A \backslash \{(x, n)\} \\
\quad return \bigcup_{(x,n) \in A} \ if\ var \notin def(x)\ then\ ReachingDefs(var, x, (V, A', En, Ex)) \\
\qquad\qquad\qquad else\ (if\ \#def(x) = 1\ then\ \{x\} \\
\qquad\qquad\qquad\qquad else\ \{x\} \cup ReachingDefs(var, x, (V, A', En, Ex)))
\end{array}
$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Program Dependence Graph

$$
\begin{aligned}
&DataDep(P) = \\
&\quad let \;\; CFG(P) = (V, A, En, Ex) \\
&\qquad\quad D = \bigcup_{n \in V, var \in use(n), x \in ReachingDefs(var, n, CFG(P))} \{(n, x)\} \\
&\quad return \; (V,D)
\end{aligned}
$$

$$
\begin{aligned}
&ProgramDep(P) = \\
&\quad let \;\; DataDep(P) = (V, D), \\
&\qquad\quad ControlDep(P) = (V, C, In), \\
&\quad return \; (V, D \cup C)
\end{aligned}
$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Static Slice

$$
\begin{aligned}
&ReachableNodes(v, G) = \\
&\quad let \;\; G = (V, A) \\
&\quad return \; \{v\} \cup \bigcup_{(v,x) \in A} ReachableNodes(x, (V, A \backslash \{(v, x)\}))
\end{aligned}
$$

$$
\begin{aligned}
&StaticSlice(P, n, Vars) = \\
&\quad let \;\; F = CFG(P) \\
&\qquad\;\; D = ProgramDep(P) \\
&\quad return \bigcup_{var \in Vars} \bigcup_{x \in ReachingDefs(var,n,F)} ReachableNodes(x, D)
\end{aligned}
$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: Artificial Sum (only scalar vars)

```
   main() {
0    int a,b,i,j,n,y;
1    n=read();
6    a=0;
7    b=0;
8    i=n;
9    j=n;
10   while (i>0) {
11     a=a+1;
12     i=i-1;
13     j=i;
14     while (j>0) {
15       b=b+1;
16       j=j-1;
       }
     };
17   y=a+b;
18   write(y);
   }
```

$$y = n + \sum_{i=1}^{n-1} i = \sum_{i=1}^{n} i$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: With Pointers

```
   main() {
0    int a,b,i,j,n, *ap,*bp,**cp;
1    n=read();
2    cp=&bp
3    ap=&a;
4    bp=ap;
5    *cp=&b;
6    a=0;
7    b=0;
8    i=n;
9    j=n;
10   while (i>0)
11     *ap=*ap+1;
12     i=i-1;
13     j=i;
14     while (j>0)
15       *bp=*bp+1;
16       j=j-1;

17   y=*ap + *bp;
18   write(y);
```

Slicing criterion 17: `*ap`

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Example: Strong vs. Weak Update

```
0     int a,b,i,j,*ap,*bp,**cp;
2     cp=&bp
3     ap=&a;
4     bp=ap;
5     *cp=&b;
```

slicing/gfx/flowsensitive.eps

strong update

update

slicing/gfx/anders

weak

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Example: Reaching Definitions

slicing/gfx/prog_pointer_cfgs.eps

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: Computation of Slice

slicing/gfx/prog_pointer_graphs_colored.eps

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Example: With Pointers (Slices)

```
   main()
     int a,b,i,j,n, *ap,*bp,**cp;

1    n=read();
2    cp=&bp;
3    ap=&a;
4    bp=ap;
5    *cp=&b;
6    a=0;
7    b=0;
8    i=n;
9    j=n;
10   while (i>0) }
11       *ap=*ap+1;
12       i=i-1;
13       j=i;
14       while (j>0) {
15           *bp=*bp+1;
16           j=j-1;
           }
         }
17   y=*ap+*bp;
18   write(y);
   }
```

Slicing criterion 17: `*ap`

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: Comparison

Program	Alias Analysis	Slice	Size
Sum	None	$\{1, 6, 8, 10, 11, 12, 17\}$	7
Sum (ptrs)	Flow insensitive	{1,2,3,5,6,7,8,10,11, 12,13,14,15,16,17}	15
Sum (ptrs)	Flow sensitive	$\{1, 3, 6, 8, 10, 11, 12, 17\}$	8

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: With Dynamic Data Structures

```
    main() {
    List *a,*b,*ap,*bp;
    int i,j,n;
1   n=read();
2   a=new List();
3   b=new List();
4   ap=a;
5   bp=b;
8   i=n;
9   j=n;
10      while (i>0) {
11         ap->next=new List();
11b        ap=ap->next;
12         i=i-1;
13         j=i;
14         while (j>0) {
15            bp->next=new List();
15b           bp=bp->next;
16            j=j-1;
           }
        }
17      ap->next = b; // conc
17b     y=a;
18      write(length(y)-2)
    }
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: Shape Analysis

17:

main_handout_jkms_ws1314-pics.pdf

Result for 17: `ap->next;`

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: W/ Dynamic Data Structures (Slice)

```
    main() {
    List *a,*b,*ap,*bp;
    int i,j,n;
1   n=read();
2   a=new List();
3   b=new List();
4   ap=a;
5   bp=b;
8   i=n;
9   j=n;
10      while (i>0) {
11          ap->next=new List();
11b         ap=ap->next;
12          i=i-1;
13          j=i;
14          while (j>0) {
15              bp->next=new List();
15b             bp=bp->next;
16              j=j-1;
            }
        }
17      ap->next = b; // conc
17b     y=a;
18      write(length(y)-2)
    }
```

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: Shape Analysis with Cycle

17:

slicing/srw17cyclic.eps

ADD: 6:

cp=cyclicList();

Result for 17:

ap->next;

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Example: Comparison

Program	Alias Analysis	Inner Loop in Slice
Sum	None	No
Sum (ptrs)	Flow insensitive (weak update)	Yes
Sum (ptrs)	Flow sensitive (strong update)	No
Sum (dyn)	Heap represented by 1 node only	Yes
Sum (dyn)	Shape analysis (strong update)	No
Sum (dyn+cycle)	Shape analysis	Yes

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Further Reading for Chapter 17

Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer- Verlag, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Part V
Conclusions and Prospectives

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter 18
Summary and Outlook

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

A Conclusion

...a question for the sense of life, or for what we did achieve resp.

- What did we consider?

  The least!

Or vice versa...

- What did we not consider?

  The most!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Especially not (or not in detail) (1)

- *Extensions of syntactic PRE beyond PDCE/PRAE*
  - Lazy Strength Reduction
  - ...
- *Semantic Extensions*
  - Semantic Code Motion/Code Placement
  - Semantic Strength Reduction
  - ...
- *Language Extensions*
  - Parallelität
  - ...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Especially not (or not in detail) (2)

- *Dynamic, profile-guided extensions*
  - Speculative PRE
  - ...
- ...

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Hints to Further Reading (1)

- *Syntactic PRE*
  - Knoop, J., Rüthing, O., and Steffen, B. 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 (2004), 460 - 461 & 462-472.
  - Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. ACM Transactions on Programming Languages and Systems 16, 4 (1994), 1117 - 1155.
  - Rüthing, O., Knoop, J., and Steffen, B. Sparse code motion. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000) (Boston, MA, Jan. 19 - 21, 2000), ACM New York, (2000), 170 - 183.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Hints to Further Reading (2)

- *Eliminating partially dead code*
  - Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94) (Orlando, FL, USA, June 20 - 24, 1994), ACM SIGPLAN Notices 29, 6 (1994), 147 - 158.
- *Eliminating partially redundant assignments*
  - Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95) (La Jolla, CA, USA, June 18 - 21, 1995), ACM SIGPLAN Notices 30, 6 (1995), 233 - 245.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Hints to Further Reading (3)

- *BB- vs. SI-Graphs*
  - Knoop, J., Koschützki, D., and Steffen, B. Basic-block graphs: Living dinosaurs? In Proceedings of the 7th International Conference on Compiler Construction (CC'98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1383 (1998), 65 - 79.
- *Moving vs. Placing*
  - Knoop, J., Rüthing, O., and Steffen, B. Code motion and code placement: Just synonyms? In Proceedings of the 7th European Symposium On Programming (ESOP'98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1381 (1998), 154 - 169.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Hints to Further Reading (4)

- *Speculative vs. classical PRE*
  - Scholz, B., Horspool, N. and Knoop, J. Optimizing for space and time usage with speculative partial redundancy elimination. In Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004) (Washington, DC, June 11 - 13, 2004), ACM SIGPLAN Notices 39, 7 (2004), 221 -230.
  - Xue, J., Knoop, J. A fresh look at PRE as a maximum flow problem. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006) (Vienna, Austria, March 25 - April 2, 2006), Springer-Verlag, Heidelberg, LNCS 3923 (2006), 139 - 154.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Hints to Further Reading (5)

- *Further Techniques and algorithms*
  - Geser, A., Knoop, J., Lüttgen, G., Rüthing, O., and Steffen, B. Non-monotone fixpoint iterations to resolve second order effects. In Proceedings of the 6th International Conference on Compiler Construction (CC'96) (Linköping, Sweden, April 24 - 26, 1996), Springer-V., Heidelberg, LNCS 1060 (1996), 106 - 120.
  - Knoop, J., and Mehofer, E. Optimal distribution assignment placement. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97) (Passau, Germany, August 26 - 29, 1997), Springer-V., Heidelberg, LNCS 1300 (1997), 364 - 373.
  - Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. Journal of Programming Languages 1, 1 (1993), 71 - 91.
  - ...: siehe auch `www.complang.tuwien.ac.at/knoop`

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Outlook

Emerging applications of (static) program analysis beyond optimization:

- Security Analysis
- Program Understanding
- Refactoring
- ...

...topics for master and PhD theses to come!

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Bibliography

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Recommended Reading

...for deepened and independent studies.

- I Textbooks
- II Monographs
- III Volumes
- III Articles

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

I Textbooks (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002.
- A. Arnold, I. Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004.
- B. A. Davey, H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 1990.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

I Textbooks (2)

- S. Even. *Graph Algorithms*. Pitman, 1979.
- Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
- Janusz Laski, William Stanley. *Software Verification and Analysis*. Springer-V., 2009.
- Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

I Textbooks (3)

- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007.
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005.
- Peter Pepper, Petra Hofstedt. *Funktionale Programmie- rung: Sprachdesign und Programmiertechnik*. Springer-V., 2006.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

II Monographs

Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

III Volumes

- Stephen S. Muchnick, Neil D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.
- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 1st edition, CRC Press, 2002.
- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 2nd edition, CRC Press, 2008.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (1)

- F. E. Allen, John A. Cocke. *A Program Data Flow Analysis Procedure*. Communications of the ACM 19(3):137-147, 1976.
- F. 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.
- B. Alpern, Mark N. Wegman, F. Ken Zadeck. *Detecting Equality of Variables in Programs*. In Proceedings of POPL'88, 1-11, 1988.
- D. Chase, Mark N. Wegmann, F. Ken Zadeck. *Analysis of Pointers and Structures*. In Proceedings PLDI'90, 296-310, 1990.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (2)

- J-G Choi, M. Gupta, M. Serrano, V.C Sreedar, and Sam Midkiff. *Escape Analysis for Java*. In Proceedings of OOPSLA'99, ACM Press, 1-19, 1999.
- Melvin E. Conway. *Proposal for an UNCOL*. Communications of the ACM 1(3):5, 1958.
- D. M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. International Journal of Computer Mathematics 27:1-14,31-32, 1989.
- D. 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (3)

- D. M. Dhamdhere. *E-path_pre: Partial Redundancy Elimination Made Easy*. ACM SIGPLAN Notices 37(8):53-65, 2002.
- D. 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.
- K.-H. Drechsler, M. 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (4)

- K.-H. Drechsler, M. P. Stadel. *A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION*. ACM SIGPLAN Notices 28(5):29-38, 1993.
- M. Emami, R. Ghiya, Laurie J. Hendren. *Context-Sen- sitive Interprocedural Points-to Analysis in the Presence of Function Pointers*. In Proceedings PLDI'94, 1-14, 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).
- C. Fecht, Helmut Seidl. *An Even Faster Solver for General Systems of Equations*. In Proceedings SAS'96, LNCS 1145, 189-204, 1996.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (5)

- C. Fecht, Helmut Seidl. *Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems*. In Proceedings ESOP'98, LNCS 1381, 90-104, 1998.
- C. Fecht, Helmut Seidl. *A Faster Solver for General Systems of Equations*. Science of Computer Programming 35(2):137-161, 1999.
- R. Ghiya, Laurie J. Hendren. *Is it a Tree, a DAG, or a Cyclic Graph?* In Proceedings POPL'96, 1-15, 1996.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (6)

- Susan Horwitz, A. Demers, T. Teitelbaum. *An Efficient General Iterative Algorithm for Dataflow Analysis*. Acta Informatica 24:679-694, 1987.
- S. M. Joshi, D. 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.
- John B. Kam, Jeffrey D. Ullman. *Global Data Flow Analysis and Iterative Algorithms*. Journal of the ACM 23:158-171, 1976.
- John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. Acta Informatica 7:305-317, 1977.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (7)

- 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.
- Marion Klein, Jens Knoop, Dirk Koschützki, Bernhard Steffen. *DFA&OPT-METAFrame: A Toolkit for Program Analysis and Optimization*. In Proceedings TACAS'96, Springer-V., LNCS 1055, 422-426, 1996.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (8)

- 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.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Imple- mentation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

III Articles (9)

- 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.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Code Motion and Code Placement: Just Synonyms?* In Procee- dings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 154-169, 1998.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (10)

- 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.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (11)

- 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.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (12)

- Donald E. Knuth. *An Empirical Study of Fortran Pro- grams*. Software – Practice and Experience 1:105-13, 1971.
- Thomas J. Marlowe, Barbara G. Ryder. *Properties of Data Flow Frameworks*. Acta Informatica 28(2):121-163, 1990.
- Florian Martin. *PAG - An Efficient Program Analyzer Generator*. Journal of Software Tools for Technology Transfer 2(1):46-67, 1998.
- Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (13)

- Flemming Nielson. *Semantics-directed Program Analysis: A Tool-maker's Perspective*. In Proceedings SAS'96, Springer-V., LNCS 1145, 2-21, 1996.
- Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings LFP'92, ACM Press, 96-108, 1992.
- H. D. Pande, Barbara Ryder. *Data-flow-based Virtual Function Resolution*. In Proceedings SAS'96, Springer-V., LNCS 1145, 238-254, 1996.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

III Articles (14)

- 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), 2011. [Best Paper Award SCAM 2011]
- Barry K. Rosen. *High-level Data Flow Analysis*. Communications of the ACM 20(10):141-156, 1977.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (15)

- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-Analysis Problems in Languages with Destructive Updating*. ACM Transactions on Programming 20(1):1-50, 1998.
- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. In Proceedings 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), 2002.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (16)

- 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.
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (17)

- 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.
- Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Pro- ceedings 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (18)

- 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.
- Mooly Sagiv, Tom Reps, Susan Horwitz. *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation*. In Proceedings TAPSOFT'95, Springer-V., LNCS 915, 651-665, 1995.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

III Articles (19)

Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. In Proceedings POPL'99, 105-118, 1999.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Appendix

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

A
Mathematical Foundations

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

A.1
Sets and Relations

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Sets and Relations (1)

Definition (A.1.1)

Let M be a set and R 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 $\forall\, m, n \in M.\ m\, R\, n \ \wedge\ n\, R\, m \ \Rightarrow\ m = n$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Sets and Relations (2)

Related notions (though less important for us here):

Definition (A.1.2)

Let M be a set and $R \subseteq M \times M$ a relation on M. Then R is called

- symmetric iff $\forall\, m, n \in M.\ m\, R\, n \iff n\, R\, m$
- total iff $\forall\, m, n \in M.\ m\, R\, n \ \vee\ n\, R\, m$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

A.2
Partially Ordered Sets

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Partially Ordered Sets

Definition (A.2.1, Quasi-Order, Partial Order)

A relation R on M is called a

- quasi-order iff R is reflexive and transitive
- partial order iff R is reflexive, transitive, and anti-symmetric

For the sake of completeness we recall:

Definition (A.2.2, Equivalence Relation)

A relation R on M is called an

- equivalence relation iff R is reflexive, transitive, and symmetric

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Remark

...a partial order is an anti-symmetric quasi-order, an equivalence relation a symmetric quasi-order.

Note: We here use terms like "partial order" as a short hand for the more accurate term "partially ordered set."

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Bounds, least and greatest Elements

Definition (A.2.3, Bounds, least/greatest Elements)

Let $(Q, \sqsubseteq)$ be a quasi-order, let $q \in Q$ and $Q' \subseteq Q$.

Then q is called

- upper (lower) bound of Q', in signs: $Q' \sqsubseteq q$ ($q \sqsubseteq Q'$), if for all $q' \in Q'$ holds: $q' \sqsubseteq q$ ($q \sqsubseteq q'$)
- least upper (greatest lower) bound of Q', if q is an upper (lower) bound of Q' and for every other upper (lower) bound $\hat{q}$ of Q' holds: $q \sqsubseteq \hat{q}$ ($\hat{q} \sqsubseteq q$)
- greatest (least) element of Q, if holds: $Q \sqsubseteq q$ ($q \sqsubseteq Q$)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Existence and Uniqueness of Bounds

We have:

- Given a partial order, least upper and greatest lower bounds are uniquely determined, if they exist.
- Given existence (and thus uniqueness), the least upper (greatest lower) bound of a set $P' \subseteq P$ of the basic set of a partial order $(P, \sqsubseteq)$ is denoted by $\bigsqcup P'$ ($\bigsqcap P'$). These elements are also called supremum and infimum of P'.
- Analogously this holds for least and greatest elements. Given existence, these elements are usually denoted by $\bot$ and $\top$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

A.3
Lattices

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Lattices and Complete Lattices

Definition (A.3.1, (Complete) Lattice)

Let $(P, \sqsubseteq)$ be a partial order.

Then $(P, \sqsubseteq)$ is called a

- lattice, if each finite subset P' of P contains a least upper and a greatest lower bound in P.
- complete lattice, if each subset P' of P contains a least upper and a greatest lower bound in P.

Hence:
...(complete) lattices are special partial orders.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Properties of Complete Lattices

Lemma (A.3.2)

Let $(P, \sqsubseteq)$ *be a complete lattice. Then we have:*

1. $\bot = \bigsqcup \emptyset = \bigsqcap P$ *is the least element of* P.
2. $\top = \bigsqcap \emptyset = \bigsqcup P$ *is the greatest element of* P.

Lemma (A.3.3)

Let $(P, \sqsubseteq)$ *be a partial order. Then the following claims 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 upper bound.*

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

A.4
Complete Partially Ordered Sets

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Complete Partial Orders

...a slightly weaker notion than a lattice that, however, is often sufficient in computer science and thus often a more adequate notion:

Definition (A.4.1, Complete Partial Order)

Let $(P, \sqsubseteq)$ be a partial order.

Then $(P, \sqsubseteq)$ is called

- complete, or shorter a CPO (complete partial order), if each ascending chain $C \subseteq P$ has a least upper bound in P.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Remark

We have:

- A CPO $(C, \sqsubseteq)$ (more accurate would be: "chain-complete partially ordered set (CCPO)") has always a least element. This element is uniquely determined as the supremum of the empty chain and usually denoted by $\perp$: $\perp =_{df} \bigsqcup \emptyset$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chains

Definition (A.4.2, Chain)

Let $(P, \sqsubseteq)$ be a partial order.

A subset $C \subseteq P$ is called

- chain of P, if the elements of C are totally ordered. For $C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \ldots\}$ ($\{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \ldots\}$) we also speak more precisely of an ascending (descending) chain of P.

A chain C is called

- finite, if C is finite; infinite otherwise.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Finite Chains, finite Elements

Definition (A.4.3, Chain-finite)

A partial order $(P, \sqsubseteq)$ is called

- chain-finite (German: kettenendlich) iff P does not contain infinite chains

Definition (A.4.4, Finite Elements)

An element $p \in P$ is called

- finite iff the set $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ is free of infinite chains
- finite relative to $r \in P$ iff the set $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ does not contain infinite chains

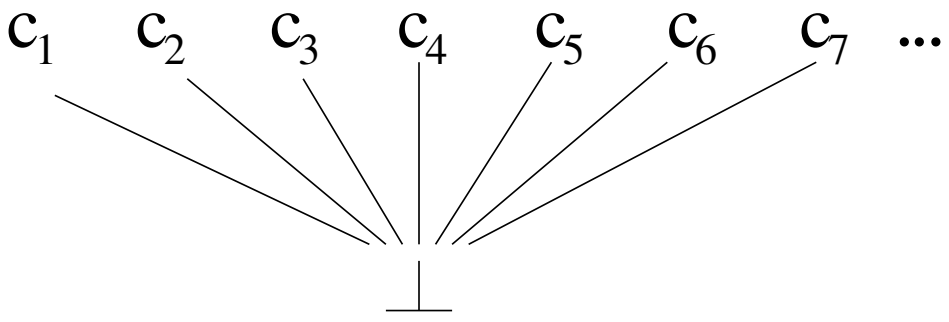
Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

(Standard) CPO Constructions (1)

Flat CPOs.

Let $(C, \sqsubseteq)$ be a CPO. Then $(C, \sqsubseteq)$ is called

- flat, if for all $c, d \in C$ holds: $c \sqsubseteq d \Leftrightarrow c = \bot \ \vee \ c = d$



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

(Standard) CPO Constructions (2)

Product construction.

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs. Then

- the non-strict (direct) product $(\bigtimes P_i, \sqsubseteq)$ with
  - $(\bigtimes P_i, \sqsubseteq) = (P_1 \times P_2 \times \ldots \times P_n, \sqsubseteq)$ with $\forall\, (p_1, p_2, \ldots, p_n),$ $(q_1, q_2, \ldots, q_n) \in \bigtimes P_i.\ (p_1, p_2, \ldots, p_n) \sqsubseteq (q_1, q_2, \ldots, q_n) \Leftrightarrow \forall\, i \in \{1, \ldots, n\}.\ p_i \sqsubseteq_i q_i$
- and the strict (direct) product (smash product) with
  - $(\bigotimes P_i, \sqsubseteq) = (P_1 \otimes P_2 \otimes \ldots \otimes P_n, \sqsubseteq)$, where $\sqsubseteq$ is defined as above under the additional constraint:

$$(p_1, p_2, \ldots, p_n) = \bot \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\ p_i = \bot_i$$

are CPOs, too.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

(Standard) CPO Constructions (3)

Sum construction.

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ CPOs. Then

- the direct sum $(\bigoplus P_i, \sqsubseteq)$ with
  - $(\bigoplus P_i, \sqsubseteq) = (P_1 \dot\cup P_2 \dot\cup \ldots \dot\cup P_n, \sqsubseteq)$ disjoint union of P_i, $i \in \{1, \ldots, n\}$ and $\forall\, p, q \in \bigoplus P_i.\ p \sqsubseteq q \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\ p, q \in P_i \ \wedge\ p \sqsubseteq_i q$

  is a CPO.

  Note: The least elements of $(P_i, \sqsubseteq_i)$, $i \in \{1, \ldots, n\}$, are usually identified, i.e., $\bot =_{df} \bot_i$, $i \in \{1, \ldots, n\}$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

(Standard) CPO Constructions (4)

Function-space construction.

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be two CPOs and $[C \to D] =_{df} \{f : C \to D \mid f \text{ continuous}\}$ the set of continuous functions from C to D.

Then

- the continuous function space $([C \to D], \sqsubseteq)$ is a CPO where
  - $\forall f, g \in [C \to D].\ f \sqsubseteq g \iff \forall c \in C.\ f(c) \sqsubseteq_D g(c)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Monotonic, Continuous Functions on CPOs

Definition (A.4.5, Monotonic, Continuous Function)

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be two CPOs and let $f : C \to D$ be a function from C to D.

Then f is called

- monotonic iff $\forall\, c, c' \in C.\ c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$
  (Preservation of the ordering of elements)
- continuous iff $\forall\, C' \subseteq C.\ f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$
  (Preservation of least upper bounds)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Properties

Using the notations introduced before, we have:

Lemma (A.4.6)

f is monotonic iff $\forall\, C' \subseteq C.\ f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$

Corollary (A.4.7)

A continuous function is always monotonic, i.e. f continuous implies f monotonic.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Inflationary Functions on CPOs

Definition (A.4.8, Inflationary Function)

Let $(C, \sqsubseteq)$ be a CPO and let $f : C \to C$ be a function on C. Then f is called

- inflationary (increasing) iff $\forall\, c \in C.\ c \sqsubseteq f(c)$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

A.5
Fixed Point Theorems

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Least and Greatest Fixed Points

Definition (A.5.1, (Least/Greatest) Fixed Point)

Let $(C, \sqsubseteq)$ be a CPO, $f : C \to C$ be a function on C and let c be an element of C, i.e., $c \in C$.

Then c is called

- fixed point of f iff $f(c) = c$

A fixed point c of f is called

- least fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow c \sqsubseteq d$
- greatest fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow d \sqsubseteq c$

Notation:

- The least resp. greatest fixed point of a function f is usually denoted by μf resp. νf.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Conditional Fixed Points (2)

Definition (A.5.2, Conditional Fixed Point)

Let $(C, \sqsubseteq)$ be a CPO, $f : C \to C$ be a function on C and let $d, c_d \in C$.

Then c_d is called

- conditional (German: bedingter) least fixed point of f wrt d iff c_d is the least fixed point of C with $d \sqsubseteq c_d$, i.e. for all other fixed points x of f with $d \sqsubseteq x$ holds: $c_d \sqsubseteq x$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Fixed Point Theorem

Theorem (A.5.3, Knaster/Tarski, Kleene)

Let $(C, \sqsubseteq)$ *be a CPO and let* $f : C \to C$ *be a continuous function on* C*.*

Then f *has a least fixed point* μf*, which equals the least upper bound of the chain (so-called Kleene-Chain)* $\{\bot, f(\bot), f^2(\bot), \ldots\}$*, i.e.*

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) = \bigsqcup \{\bot, f(\bot), f^2(\bot), \ldots\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Proof of Fixed Point Theorem A.5.3 (1)

We have to prove:

μf

1. exists
2. is a fixed point
3. is the least fixed point

of f.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Proof of Fixed Point Theorem A.5.3 (2)

1. Existence
   - It holds $f^0 \perp = \perp$ and $\perp \sqsubseteq c$ for all $c \in C$.
   - By means of (natural) induction we can show: $f^n \perp \sqsubseteq f^n c$ for all $c \in C$.
   - Thus we have $f^n \perp \sqsubseteq f^m \perp$ for all n, m with $n \leq m$. Hence, $\{f^n \perp \mid n \geq 0\}$ is a (non-finite) chain of C.
   - The existence of $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is thus an immediate consequence of the CPO properties of $(C, \sqsubseteq)$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Proof of Fixed Point Theorem A.5.3 (3)

2. Fixed point property

$$
\begin{array}{rcl}
 & & f(\bigsqcup_{i\in\mathbb{N}_0} f^i(\bot)) \\
(f \text{ continuous}) & = & \bigsqcup_{i\in\mathbb{N}_0} f(f^i\bot) \\
 & = & \bigsqcup_{i\in\mathbb{N}_1} f^i\bot \\
(K \text{ chain} \Rightarrow \bigsqcup K = \bot \sqcup \bigsqcup K) & = & (\bigsqcup_{i\in\mathbb{N}_1} f^i\bot) \sqcup \bot \\
(f^0\bot = \bot) & = & \bigsqcup_{i\in\mathbb{N}_0} f^i\bot
\end{array}
$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Proof of Fixed Point Theorem A.5.3 (4)

3. Least fixed point

- Let c be an arbitrarily chosen fixed point of f. Then we have $\bot \sqsubseteq c$, and hence also $f^n \bot \sqsubseteq f^n c$ for all $n \geq 0$.
- Thus, we have $f^n \bot \sqsubseteq c$ because of our choice of c as fixed point of f.
- Thus, we also have that c is an upper bound of $\{f^i(\bot) \mid i \in \mathbb{N}_0\}$.
- Since $\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)$ is the least upper bound of this chain by definition, we obtain as desired $\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) \sqsubseteq c$.

$\square$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Conditional Fixed Points

Theorem (A.5.4, Conditional Fixed Points)

Let $(C, \sqsubseteq)$ be a CPO, let $f : C \to C$ be a continuous, inflationary function on C, and let $d \in C$.

Then f has a unique conditional fixed point μf_d. This fixed point equals the least upper bound of the chain $\{d, f(d), f^2(d), \ldots\}$, i.e.

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \ldots\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Finite Fixed Points

Theorem (A.5.5, Finite Fixed Points)

Let $(C, \sqsubseteq)$ *be a CPO and let* $f : C \to C$ *be a continuous function on* C.

Then we have: If two elements in a row occurring in the Kleene-chain of f *are equal, e.g.* $f^i(\bot) = f^{i+1}(\bot)$, *then we have:* $\mu f = f^i(\bot)$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Existence of Finite Fixed Points

Sufficient conditions for the existence of finite fixed points e.g. are

- Finiteness of domain and range of f
- f is of the form $f(c) = c \sqcup g(c)$ for monotone g on some chain-complete domain

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Appendix A: Further Reading (1)

- A. Arnold, I. Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
- B. A. Davey, H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 1990.
- Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings LFP'92, ACM Press, 96-108, 1992.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 4, Denotational Semantics)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Appendix A: Further Reading (2)

- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 5, Denotational Semantics)
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Appendix A, Partially Ordered Sets)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Chapter 10, Beispiel: Berechnung von Fixpunkten)

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

B

Intricacies of Basis Block Graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter B.1

Motivation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Basic Block vs. Single Instruction Graphs

In this chapter we investigate the adequacy of different program representations.

To this end we will consider and compare programs in form of node and edge-labelled flow graphs with basic blocks and single instructions and investigate their

- advantages and disadvantages for program analysis

...thereby addressing the question:

- Basic Block vs. Single Instruction Graphs: Just a Matter of Taste?

On the fly we will learn:

- Some further examples of real world data flow analysis problems and data flow analyses.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Basic Blocks: Supposed Advantages

Advantages of basic blocks in applications that are commonly attributed to them ("folk knowledge"):

Better scalability because

- less nodes are involved in the (potentially) compu- tationally expensive fixed point iteration and thus
- larger programs fit into the main memory.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Basic Blocks: Definite Disadvantages

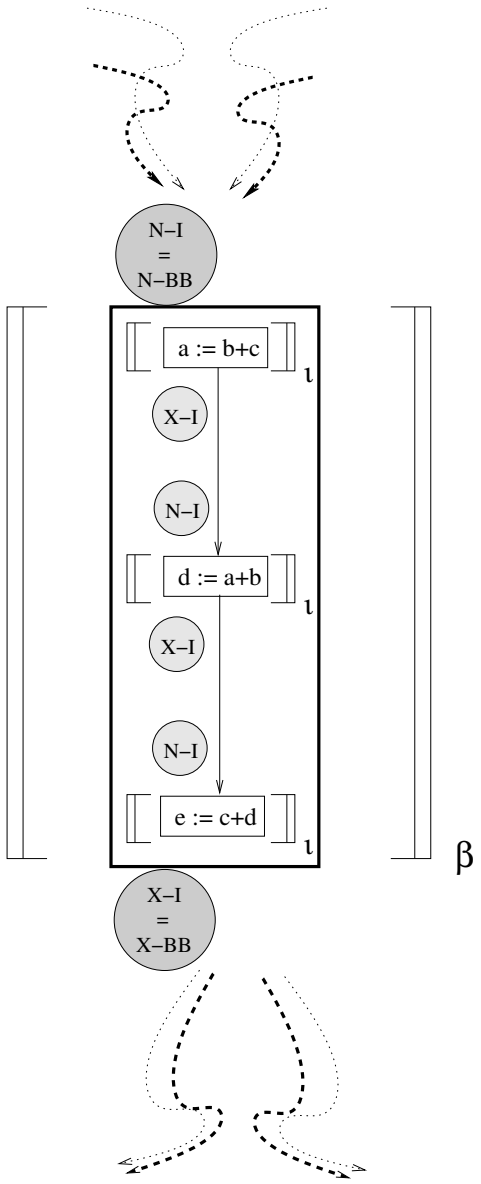
Definite disadvantages of basic blocks in applications:

- Higher conceptual complexity: Basic blocks introduce an undesired hierarchy into flow graghs that makes both theoretical reasoning and practical implemenentations more difficult.
- Necessity of 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 graph (e.g. in dead code elimination, constant propagation,...); or that require "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).

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Hierarchy by Basic Blocks

Illustration:



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

In the following

Investigating the

- advantages and disadvantages of basic block (BB) flow graphs compared to single instructions (SI) flow graphs.

by means of examples

- of some data flow analysis problems we already considered
  - Availability of expressions
  - Simple constants

  and some new ones:
  - Faint variables

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter B.1.1

Edge-labelled Single Instruction Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The MOP_{SI} Approach

...for edge-labelled single instruction flow graphs.

The *MOP* Solution:

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

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The $MaxFP_{SI}$ Approach

...for edge-labelled single instruction flow graphs.

The *MaxFP* Solution:

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ MaxFP_{(\llbracket\ \rrbracket_\iota, c_s)}(n) =_{df} \mathtt{inf}^*_{c_s}(n)$$

where $\mathtt{inf}^*_{c_s}$ denotes the greatest solution of the *MaxFP* Equation System:

$$inf(n) = \begin{cases} c_s & \text{if } n = s \\ \bigsqcap \{\, \llbracket\, (m,n)\, \rrbracket_\iota(inf(m)) \mid m \in pred_G(n) \,\} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter B.1.2
Node-labelled Basic Block Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Notations

In the following we denote

- basic block nodes by boldface letters (**m**, **n**,...)
- single instruction nodes by normalface letters (m, n,...)

Furthermore we denote by

- $[\![\ ]\!]_\beta$ and
- $[\![\ ]\!]_\iota$

(local) abstract data flow analysis functionals on the level of basic blocks and single instructions, respectively.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The MOP_{BB} Approach (1)

...for node-labelled basic block flow graphs.

The *MOP* Solution on the BB-Level:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, \mathbf{n} \in \mathbf{N}.\ MOP_{(\llbracket\ \rrbracket_\beta, c_{\mathbf{s}})}(\mathbf{n}) =_{df} \left(N\text{-}MOP_{(\llbracket\ \rrbracket_\beta, c_{\mathbf{s}})}(\mathbf{n}),\ X\text{-}MOP_{(\llbracket\ \rrbracket_\beta, c_{\mathbf{s}})}(\mathbf{n}) \right)$$

with

$$N\text{-}MOP_{(\llbracket\ \rrbracket_\beta, c_{\mathbf{s}})}(\mathbf{n}) =_{df} \bigsqcap \{\, \llbracket\, p\, \rrbracket_\beta(c_{\mathbf{s}}) \mid p \in \mathbf{P_G}[\mathbf{s}, \mathbf{n}[\,\}$$

$$X\text{-}MOP_{(\llbracket\ \rrbracket_\beta, c_{\mathbf{s}})}(\mathbf{n}) =_{df} \bigsqcap \{\, \llbracket\, p\, \rrbracket_\beta(c_{\mathbf{s}}) \mid p \in \mathbf{P_G}[\mathbf{s}, \mathbf{n}]\,\}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The MOP_{BB} Approach (2)

...and its continuation on the SI-Level:

$$\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.\ MOP_{(\llbracket\ \rrbracket_\iota, c_s)}(n) =_{df} \\ (\ N\text{-}MOP_{(\llbracket\ \rrbracket_\iota, c_s)}(n),\ X\text{-}MOP_{(\llbracket\ \rrbracket_\iota, c_s)}(n)\ )$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The MOP_{BB} Approach (3)

...with

$$N\text{-}MOP_{(\llbracket\ \rrbracket_\iota, c_s)}(n) \quad =_{df} \quad \begin{cases} N\text{-}MOP_{(\llbracket\ \rrbracket_\beta, c_s)}(\texttt{block}(n)) \\ \quad \text{if } n = start(\texttt{block}(n)) \\ \\ \llbracket\, p\, \rrbracket_\iota(N\text{-}MOP_{(\llbracket\ \rrbracket_\beta, c_s)}(\texttt{block}(n))) \\ \quad \text{otherwise } (p \text{ prefix path from} \\ \quad start(\texttt{block}(n)) \text{ to (exclusively) } n) \end{cases}$$

$$X\text{-}MOP_{(\llbracket\ \rrbracket_\iota, c_s)}(n) \quad =_{df} \quad \llbracket\, p\, \rrbracket_\iota(N\text{-}MOP_{(\llbracket\ \rrbracket_\beta, c_s)}(\texttt{block}(n))) \\ (p \text{ prefix path from } start(\texttt{block}(n)) \\ \text{up to (inclusively) } n)$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The $MaxFP_{BB}$ Approach (1)

...for node-labelled basic block flow graphs:

The *MaxFP* Solution on the BB-Level:

$$\forall\, c_s \in \mathcal{C}\, \forall\, \mathbf{n} \in \mathbf{N}.\ MaxFP_{(\llbracket\ \rrbracket_\beta, c_s)}(\mathbf{n}) =_{df} \\ (\ N\text{-}MFP_{(\llbracket\ \rrbracket_\beta, c_s)}(\mathbf{n}),\ X\text{-}MFP_{(\llbracket\ \rrbracket_\beta, c_s)}(\mathbf{n})\ )$$

with

$$N\text{-}MFP_{(\llbracket\ \rrbracket_\beta, c_s)}(\mathbf{n}) =_{df} \mathrm{pre}^{\beta}_{c_s}(\mathbf{n}) \qquad \text{and}$$

$$X\text{-}MFP_{(\llbracket\ \rrbracket_\beta, c_s)}(\mathbf{n}) =_{df} \mathrm{post}^{\beta}_{c_s}(\mathbf{n})$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The $MaxFP_{BB}$ Approach (2)

...where $pre^{\beta}_{c_s}$ and $post^{\beta}_{c_s}$ denote the greatest solution of the equation system

$$pre(\mathbf{n}) = \begin{cases} c_{\mathbf{s}} & \text{if } \mathbf{n} = \mathbf{s} \\ \bigsqcap \{ post(\mathbf{m}) \mid \mathbf{m} \in pred_{\mathbf{G}}(\mathbf{n}) \} & \text{otherwise} \end{cases}$$

$$post(\mathbf{n}) = [\![\mathbf{n}]\!]_{\beta}(pre(\mathbf{n}))$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The $MaxFP_{BB}$ Approach (3)

...and its continuation on the SI-Level:

$$\forall\, c_s \in \mathcal{C}\, \forall\, n \in N.\ MaxFP_{(\llbracket\ \rrbracket_\iota, c_s)}(n) =_{df} \\ (\ N\text{-}MFP_{(\llbracket\ \rrbracket_\iota, c_s)}(n),\ X\text{-}MFP_{(\llbracket\ \rrbracket_\iota, c_s)}(n)\ )$$

with

$$N\text{-}MFP_{(\llbracket\ \rrbracket_\iota, c_s)}(n) =_{df} \mathrm{pre}^{\iota}_{c_s}(n) \qquad \text{and}$$

$$X\text{-}MFP_{(\llbracket\ \rrbracket_\iota, c_s)}(n) =_{df} \mathrm{post}^{\iota}_{c_s}(n)$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The $MaxFP_{BB}$ Approach (4)

...where $pre^{\iota}_{c_s}$ and $post^{\iota}_{c_s}$ denote the greatest solution of the equation system

$$pre(n) = \begin{cases} pre^{\beta}_{c_s}(\texttt{block}(n)) \\ \quad \text{if } n = start(\texttt{block}(n)) \\ \\ post(m) \\ \quad \text{otherwise } (m \text{ is here the uniquely} \\ \quad \text{determined predecessor of } n \\ \quad \text{in } \texttt{block}(n)) \end{cases}$$

$$post(n) = [\![n]\!]_{\iota}(pre(n))$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter B.2

Availability of Expressions

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter B.2.1
Node-labelled Basic Block Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Availability of Expressions (1)

...for node-labelled basic-block flow graphs.

Stage I: The Basic-block Level

Local Predicates (associated with BB-nodes):

- BB-XCOMP$_\beta(t)$: β contains a statement ι that com- putes t, and neither ι nor a statement following ι in β modifies an operand of t.
- BB-TRANSP$_\beta(t)$: β does not contain a statement that modifies an operand of t.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Availability of Expressions (2)

The BB-Equation System of Stage I:

$$\text{BB-N-AVAIL}_\beta = \begin{cases} \textbf{false} & \text{if } \beta = \textbf{s} \\ \prod_{\hat{\beta} \in \mathit{pred}(\beta)} \text{BB-X-AVAIL}_{\hat{\beta}} & \text{otherwise} \end{cases}$$

$$\text{BB-X-AVAIL}_\beta = \text{BB-N-AVAIL}_\beta \cdot \text{BB-TRANSP}_\beta + \text{BB-XCOMP}_\beta$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Availability of Expressions (3)

Stage II: The Instruction Level

Lokale Prädikate (associated with SI-nodes):

- $\text{COMP}_\iota(t)$: ι computes t.
- $\text{TRANSP}_\iota(t)$: ι does not modify an operand of t.
- BB-N-AVAIL*, BB-X-AVAIL*: the greatest solution of the equation system of Stage I.

The SI-Equation System of Stage II:

$$\text{N-AVAIL}_\iota = \begin{cases} \text{BB-N-AVAIL}^\star_{\texttt{block}(\iota)} & \text{if } \iota = \mathit{start}(\texttt{block}(\iota)) \\ \text{X-AVAIL}_{\mathit{pred}(\iota)} & \text{otherwise} \\ & (\text{note: } |\mathit{pred}(\iota)| = 1) \end{cases}$$

$$\text{X-AVAIL}_\iota = \begin{cases} \text{BB-X-AVAIL}^\star_{\texttt{block}(\iota)} & \text{if } \iota = \mathit{end}(\texttt{block}(\iota)) \\ (\text{N-AVAIL}_\iota + \text{COMP}_\iota) \cdot \text{TRANSP}_\iota & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter B.2.2

Node-labelled Single Instruction Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Availability of Expressions

...for node-labelled single instruction flow graphs.

Local Predicates (associated with SI-nodes):

- $\mathrm{COMP}_\iota(t)$: ι computes t.
- $\mathrm{TRANSP}_\iota(t)$: ι does not modify an operand of t.

The EA-Equation System:

$$\begin{aligned}
\text{N-AVAIL}_\iota &= \begin{cases} \textbf{false} & \text{if } \iota = s \\ \prod\limits_{\hat{\iota} \in pred(\iota)} \text{X-AVAIL}_{\hat{\iota}} & \text{otherwise} \end{cases} \\
\text{X-AVAIL}_\iota &= (\text{N-AVAIL}_\iota + \text{COMP}_\iota) \cdot \text{TRANSP}_\iota
\end{aligned}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter B.2.3

Edge-labelled Single Instruction Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Availability of Expressions

...for edge-labelled single-instruction flow graphs.

Locale Predicates (associated with SI-edges):

- $\mathrm{COMP}_{\varepsilon}(t)$: Staement ι of edge ε computes t.
- $\mathrm{TRANSP}_{\varepsilon}(t)$: Statement ι of edge ε does not modify an operand of t.

The SI-Equation System:

$$\mathtt{Avail}_n = \begin{cases} \textbf{false} & \text{if } n = s \\ \prod\limits_{m \in pred(n)} (\mathtt{Avail}_m + \mathrm{COMP}_{(m,n)}) \cdot \mathrm{TRANSP}_{(m,n)} & \\ & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Outlook

Next we consider two further examples in order to illustrate the impact of the chosen flow graph representation variant on the conceptual and practical complexity of data flow analysis:

- Constant propagation and folding
- Faint variable elimination

To this end we consider formulations of these problems for:

- node-labelled basic-block flow grapns
- edge-labelled single instruction flow graphs

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter B.3
Constant Propagation and Folding

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Constant Propagation and Folding

...considering the example of so-called simple constants.

To this end we need two auxiliary functions:

- Backward substitution
- State transformation

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Backward Substitution and State Transformation for Assignments

Let $\iota \equiv (x := t)$ be a statement. Then we define:

- Backward substitution

  $\delta_\iota : \mathbf{T} \to \mathbf{T}$ by $\delta_\iota(s) =_{df} s[t/x]$ for all $s \in \mathbf{T}$, where $s[t/x]$ denotes the simultaneous replacement of all occurrences of x by t in s.

- State transformation

$$\theta_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

The Relationship of δ and θ

Let $\mathcal{I}$ denote the set of all statements.

Lemma (B.3.1, Substitution Lemma)

$$\forall\, t \in \mathbf{T}\ \forall\, \sigma \in \Sigma\ \forall\, \iota \in \mathcal{I}.\ \mathcal{E}(\delta_\iota(t))(\sigma) = \mathcal{E}(t)(\theta_\iota(\sigma))$$

Proof by induction on the structure of t.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Chapter B.3.1

Edge-labelled Single Instruction Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Simple Constants

...for edge-labelled single instruction flow graphs.

- $\mathrm{CP}_n \in \Sigma$
- $\sigma_0 \in \Sigma$ start information

The SI-Equation System:

$$\forall\, v \in \mathbf{V}.\ \mathrm{CP}_n = \begin{cases} \sigma_0(v) & \text{if } n = s \\ \bigsqcap \{\, \mathcal{E}(\delta_{(m,n)}(v))(\mathrm{CP}_m) \mid m \in \mathit{pred}(n) \,\} & \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Chapter B.3.2
Node-labelled Basic Block Approach

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Backward Substitution and State Transformation on Paths

Extending δ and θ to paths (and hence to basic blocks, too):

- $\Delta_p : \mathbf{T} \to \mathbf{T}$ defined by $\Delta_p =_{df} \delta_{n_q}$ for $q = 1$ and by $\Delta_{(n_1,\ldots,n_{q-1})} \circ \delta_{n_q}$ for $q > 1$
- $\Theta_p : \Sigma \to \Sigma$ defined by $\Theta_p =_{df} \theta_{n_1}$ for $q = 1$ and by $\Theta_{(n_2,\ldots,n_q)} \circ \theta_{n_1}$ for $q > 1$.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

The Relationship of Δ and Θ

Let $\mathcal{B}$ denote the set of all basic blocks.

Lemma (B.3.1.1, Generalized Substitution Lemma)

$$\forall\, t \in \mathbf{T}\ \forall\, \sigma \in \Sigma\ \forall\, \beta \in \mathcal{B}.\ \mathcal{E}(\Delta_\beta(t))(\sigma) = \mathcal{E}(t)(\Theta_\beta(\sigma))$$

Proof by induction on the length of p.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Simple Constants (1)

...for node-labelled basic-block flow graphs.

Stage I: Basic-block Level

Remark:

- $\Delta_\beta(v) =_{df} \delta_{\iota_1} \circ \ldots \circ \delta_{\iota_q}(v)$, where $\beta \equiv \iota_1; \ldots; \iota_q$.
- $\text{BB-N-CP}_\beta, \text{BB-X-CP}_\beta, \text{N-CP}_\iota, \text{X-CP}_\iota \in \Sigma$
- $\sigma_0 \in \Sigma$ start information

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Simple Constants (2)

The BB-Equation System of Stage I:

$$\text{BB-N-CP}_\beta = \begin{cases} \sigma_0 & \text{if } \beta = \mathbf{s} \\ \bigsqcap\{\text{BB-X-CP}_{\hat{\beta}} \mid \hat{\beta} \in \mathit{pred}(\beta)\} & \\ \text{otherwise} & \end{cases}$$

$$\forall\, v \in \mathbf{V}.\ \text{BB-X-CP}_\beta(v) = \mathcal{E}(\Delta_\beta(v))(\text{BB-N-CP}_\beta)$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Simple Constants (3)

Stage II: Instruction Level

Pre-computed Results (of Stage I):

- BB-N-CP*, BB-X-CP*: the greatest solution of the equation system of Stage I.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Simple Constants (4)

The SI-Equation System of Stage II:

$$\text{N-CP}_\iota = \begin{cases} \text{BB-N-CP}^\star_{\texttt{block}(\iota)} \\ \quad \text{if } \iota = \mathit{start}(\texttt{block}(\iota)) \\ \text{X-CP}_{\mathit{pred}(\iota)} \\ \quad \text{otherwise (note: } |\mathit{pred}(\iota)| = 1) \end{cases}$$

$$\forall\, v \in \mathbf{V}.\ \text{X-CP}_\iota(v) = \begin{cases} \text{BB-X-CP}^\star_{\texttt{block}(\iota)}(v) \\ \quad \text{if } \iota = \mathit{end}(\texttt{block}(\iota)) \\ \mathcal{E}(\delta_\iota(v))(\text{N-CP}_\iota) \quad \text{otherwise} \end{cases}$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

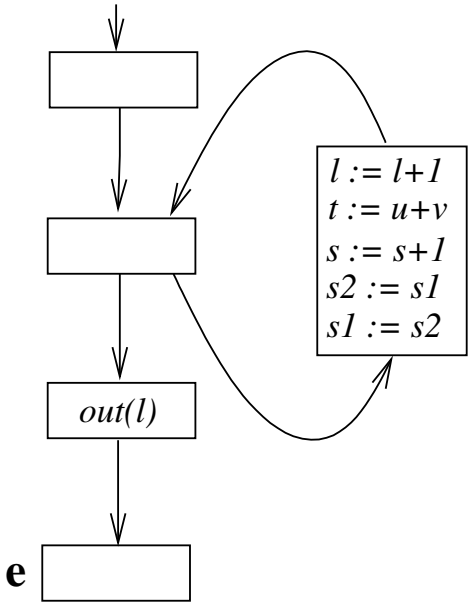
Chapter B.4
Faint Variables

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Motivation

Statement

- $l := l + 1$ is live.
- $t := u + v$ is dead.
- $s := s + 1$ as well as $s1 := s2;\ s2 := s1$ are live but faint (schwach, kraftlos, ohnmächtig, schattenhaft).



Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Preliminaries

...for edge-labelled single instruction flow graphs.

Local Predicates (associated with single instruction edges):

- $\text{USED}_\varepsilon(v)$: Statement ι of edge ε uses v.
- $\text{MOD}_\varepsilon(v)$: Statement ι of edge ε modifies v.
- $\text{REL-USED}_\varepsilon(v)$: v is a variable that occurs in the statement ι of the edge ε and "is forced to live" by it (e.g. for ι being an output operation).
- $\text{ASS-USED}_\varepsilon(v)$: v is a variable that occurs in the right-hand side expression of the assignment ι of the edge ε.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Faint Variable Analysis

The SI-Equation System:

$$\mathrm{FAINT}_n(v) =$$

$$\prod_{m \in succ(n)} \overline{\mathrm{REL\text{-}USED}_{(n,m)}(v)} *$$

$$(\mathrm{FAINT}_m(v) + \mathrm{MOD}_{(n,m)}(v)) *$$

$$(\mathrm{FAINT}_m(LhsVar_{(n,m)}) + \overline{\mathrm{ASS\text{-}USED}_{(n,m)}(v)})$$

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Summary

The faint variables problem is an example of a DFA problem, for which a formulation is

- obvious on (node- and edge-labelled) single instruction flow graphs,
- not at all obvious, if not impossible at all on (node- and edge-labelled) basic-block flow graphs.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

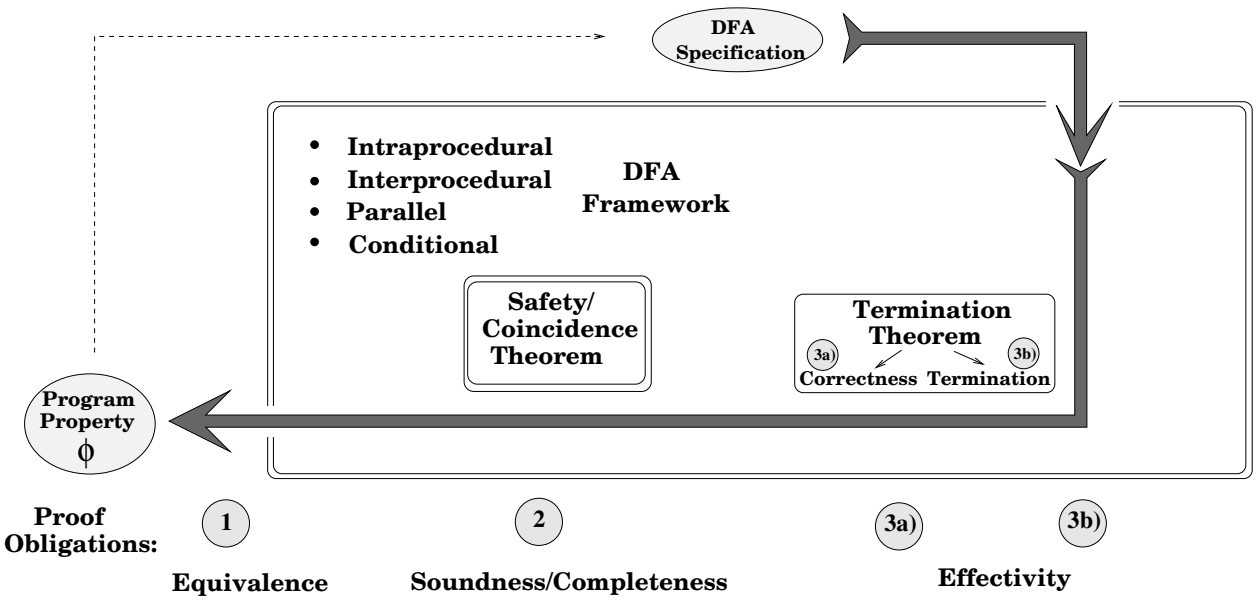
Chapter B.5
Conclusion

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18

Conclusion

In principle, all 4 representation variants of flow graphs are equally powerful.

Hence, conceptually the general framework resp. tool kit view



and knowing that the variants differ in their adequacy and in the specification, implementation and proof obligations they require depending on the task at hand suffices.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17

Appendix B: Further Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 9.4, Constant Propagation)
- 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.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Chap. 14
Chap. 15
Chap. 16
Chap. 17
Chap. 18