Optimierende Compiler

LVA 185.A04, VU 2.0, ECTS 3.0
WS 2016/2017

(Stand: 25.01.2017)

Jens Knoop

Technische Universität Wien
Institut für Computersprachen

Table of Contents

Table of Contents (1)

Table of Contents (2)

Table of Contents (3)

Part II: Intraprocedural Data Flow Analysis

Table of Contents (4)

Table of Contents (5)

Table of Contents (6)

Table of Contents (7)

Table of Contents (8)

Table of Contents (9)

Table of Contents (10)

Table of Contents (11)

Table of Contents (12)

Part III: Interprocedural Data Flow Analysis

Table of Contents (13)

Table of Contents (14)

Table of Contents (15)

Table of Contents (16)

Part V: Conclusions and Perspectives

Table of Contents (17)

Appendices

Table of Contents (18)

Table of Contents (19)

Table of Contents (20)

Part I

Introduction

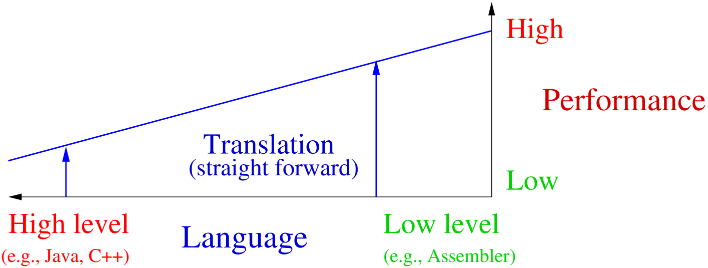
Chapter 1

Motivation

Chapter 1.1

Setting the Scene

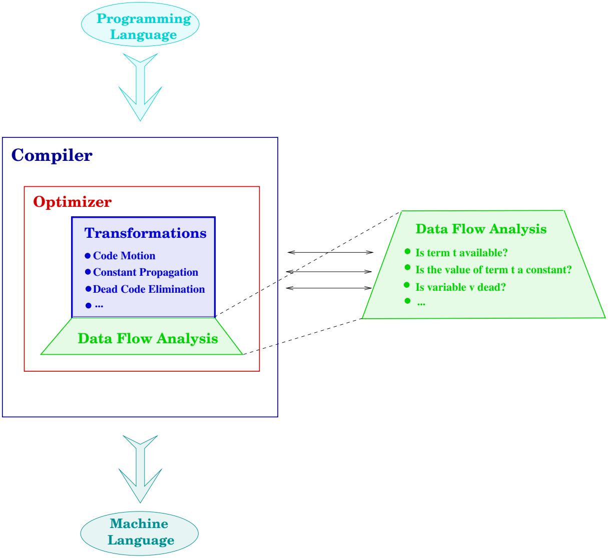
Languages and Their Perceived Performance



Common perception

- High level languages/abstraction give low level of performance.

The Optimizing Compiler to the Rescue



Translation vs. Optimizing Compilation

Translation (straight forward)

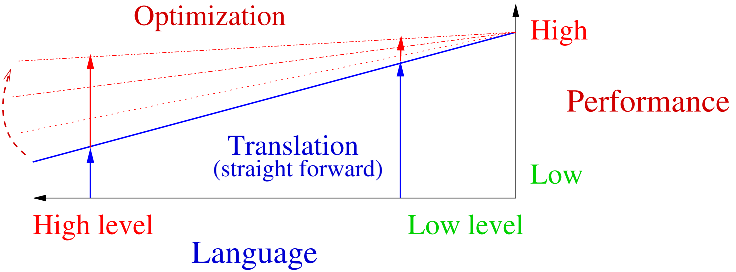
- preserves semantics but does not exploit specific opportunities of lower level languages with respect to performance.

Optimizing compilation/optimization

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

Optimizing Compilation/Optimization

...aims at closing the performance gap.

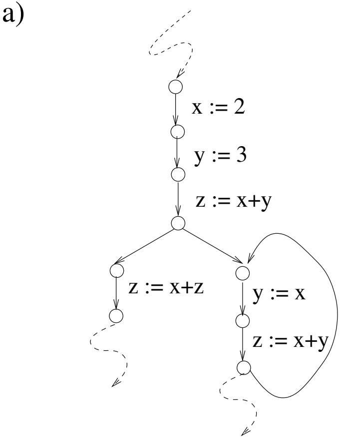


Selected Common Optimizations

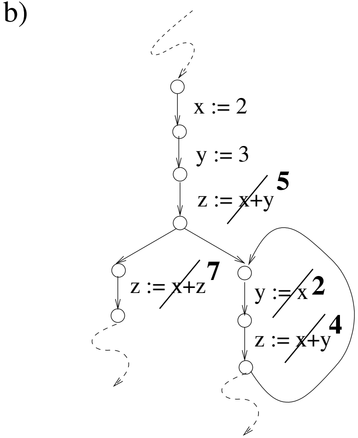
...and optimization sequences for illustration and motivation.

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

Constant Propagation: Orig. & Opt. Program



Original program

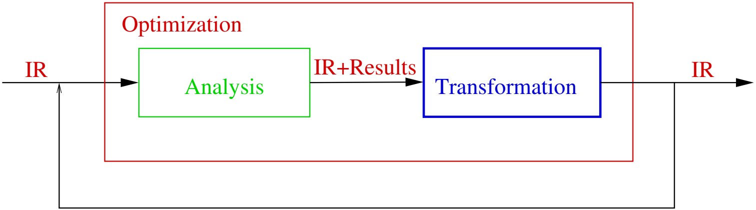


After simple constant propagation

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
  - ...
- Avoid costly computations
  - results can be computed less costly
  - ...
- ...

Optimization Considered Schematically

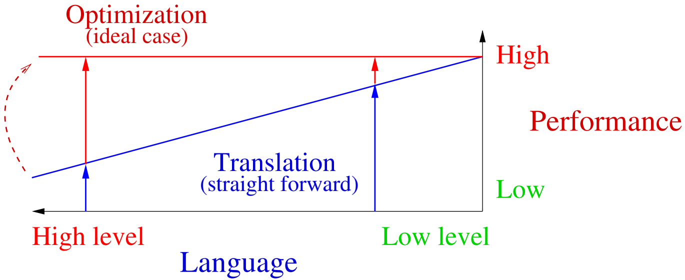


Optimization is a (repeatedly applied) two-stage process consisting of

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

Optimization: The Ideal Case

...the performance gap is closed.



Note

- The term optimization is a misnomer: usually we do not achieve an “optimal” solution – but it is the ideal case)

Chapter 1.2

An Extensive Illustrating Example

The Runnning Example

...adding two 2-dimensional matrices:

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

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

Step 1: Lowering the High-level Code to IR

...revealing the address computation:

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

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

Step 2: Optimizations on IR

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

1st Analysis: Available Expressions Analysis

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

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

1st Opt.: Common Subexpression Elimination

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

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

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

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

2nd Analysis: Loop Invariants Detection

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

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

2nd Opt.: 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;
}
```

3rd Analysis: Induction Variables Detection

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

Basic Induction Variables

- Variables 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.

3rd Optimization: Strength Reduction (1)

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

Classical example:

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

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

3rd Optimization: Strength Reduction (2)

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

```
i=0;

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

}
```

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

4th Analysis: Copy Analysis

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

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

4th Optimization: Copy Propagation

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

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

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

5th Analysis: Dead Variables Analysis (1)

A variable is

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

A live (dead) variables analysis

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

5th Analysis: Dead Variables Analysis (2)

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

- Only dead variables are marked.

5th Optimization: Dead Code Elimination

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

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

```
i=0;
t3=0;
while(i<m) {
  j=0;

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

6th Analysis: LFTR Candidates Detection

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

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

6th Opt.: Linear Function Test Replacement

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

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

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

   t3=t3+n;
}
```

Summary of Optimizations

... applied in this example.

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

The Application of Many Optimizations

... as in the preceeding example is quite typical in practice:

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

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

Challenging Problem

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

Some optimizations

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

Are their further Optimizations?

In fact, there is a plethora of further optimizations, i.e., pairs of analyses and transformations.

For example

- Optimizations for
  - object-oriented languages
  - logical and functional languages
  - parallel and distributed languages
  - ...
- Array analysis and optimization
- Pointer/alias/shape analysis and optimization
- Heap analysis, garbage collection
- ...

Note

In the preceding example

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

This, however,

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

Requiring and insisting on it

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

For Illustration

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

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

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

Key Questions

Would you like to program matrix addition

- as shown on the right-hand side

...or prefer programming it

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

Most likely

...you would even prefer programming matrix addition

- using an even higher language offering an abstraction allowing us to write

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

As a matter of fact

- Optimizing compilation is the key to render this possible!

Chapter 1.3

The Impact of Optimization: A Case Study

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 thru EC6

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

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;
}
```

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;
};
```

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

Programming Styles - 5&6

EC5: Functional Programming with Boost::lambda

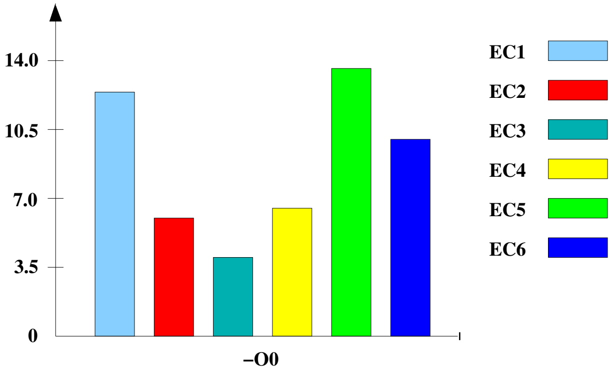
```
std::for_each(mycontainer.begin(),
      mycontainer.end(),
      boost::lambda::_1 +=1.0 );
```

EC6: Functional Programming with Boost::phoenix

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

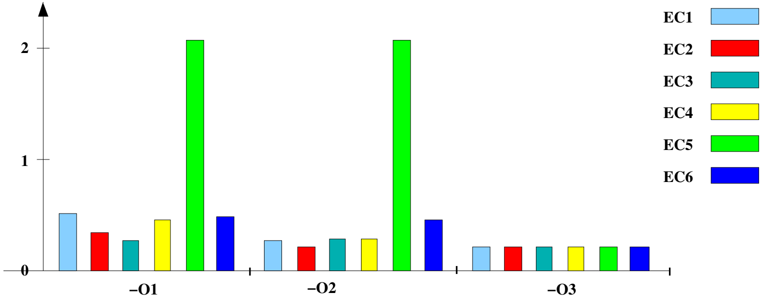
- Use of unnamed function object.

Evaluation: EC1-6 w/out optimization



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

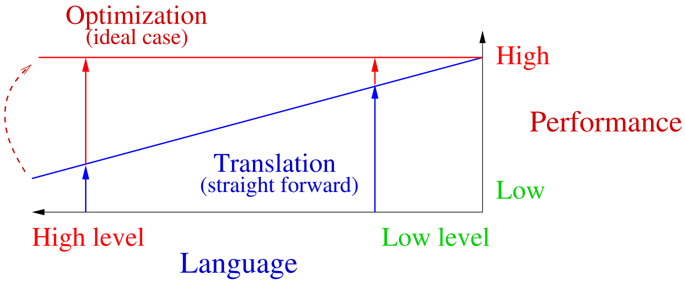
Evaluation: EC1-6 w/ optimization levels O1-3



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

In this Case Study

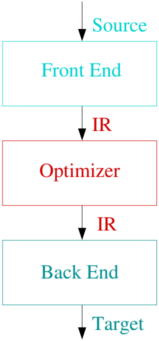
...the performance gap is closed!



Chapter 1.4

Compilers, Optimizing Compilers, and their Structure

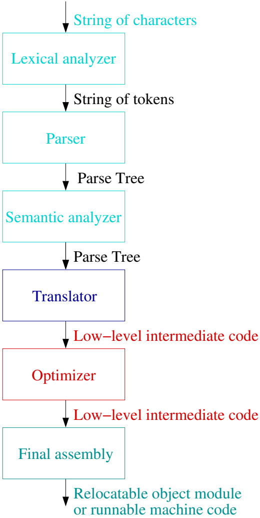
Generic Structure of an Optimizing Compiler



Goal of code optimization

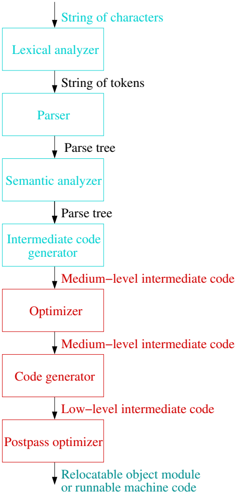
- Discover, at compile-time, information about the run-time behavior of the program and use that information to improve the code generated by the compiler.

Model of a Low Level Optimizer



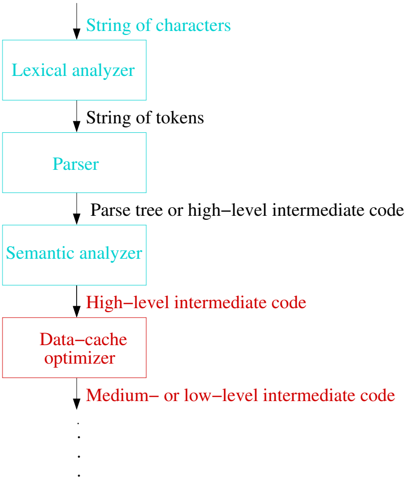
- All optimization is done on a low level intermediate code.

Model of a Mixed Level Optimizer



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

Model of a High Level Cache Optimizer



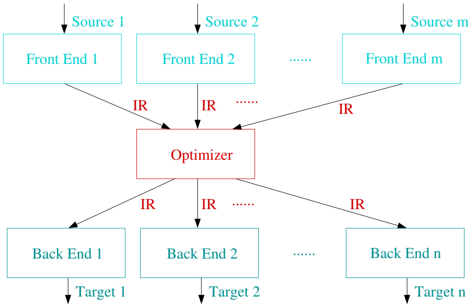
Adding data-cache optimization to an optimizing compiler

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

Examples

- High-Level optimizations
  - IBM's PowerPC compiler: first translates to LL code (XIL) and then generates a HL representation (YIL) from it to do data-cache optimization.
  - Source-To-Source Optimizer Tools: Sage++, LLNL-ROSE, JTransformer.
- Mixed model
  - Sun Microsystem's compilers for SPARC.
  - Intel's compilers for the 386 architecture family.
  - Silicon Graphic's compilers for MIPS.
- Low level model
  - IBM's compilers for PowerPC.
  - Hewlett-Packard's compilers for PA-RISC.

Practice: m-2-n Compilers and Optimizers



Idea: Decoupling of Compiler Front Ends from Back Ends

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

IR-Decoupling of Compiler Front/Back Ends

...is an application of Conway's famous UNCOL concept:

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

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

Intermediate Representation (IR)

- High level
  - quite close to source language, e.g., abstract syntax tree
  - code generation issues are quite clumsy at high-level
  - adequate for high-level optimizations (cache, loops)
- Medium level
  - represent source variables, temporaries, (and registers)
  - reduce control flow to conditional and unconditional branches
  - adequate to perform machine independent optimizations
- Low level
  - correspond to target-machine instructions
  - adequate to perform machine dependent optimizations

Chapter 1.5

Optimizations: Objectives and Categorization

Optimizations: Kinds and Objectives

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

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

Considerations for Optimization

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

Scope of Optimization (1)

Local

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

Scope of Optimization (2)

Global

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

Optimization Taxonomy

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

- Machine independent
  - largely ignore the details of the target machine
  - in many cases profitability of a transformation depends on detailed machine-dependent issues, but those are ignored
- Machine dependent
  - explicitly consider details of the target machine
  - many of these transformations fall into the realm of code generation
  - some are within the scope of the optimizer (some cache optimizations, some expose instruction level parallelism)

Machine Independent Optimizations (1)

- Dead code elimination
  - eliminate useless or unreachable code
  - algebraic identities
- 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

Machine Independent Optimizations (2)

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

Machine Dependent Optimizations

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

Chapter 1.6

Tools for Compiler Construction and Optimization

On-line Resources

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

- Compilers.net Team. *Search Machine on Compilers and Programming Languages, Directory of Compiler and Language Resources*, 1997-2007. `http://www.compilers.net`

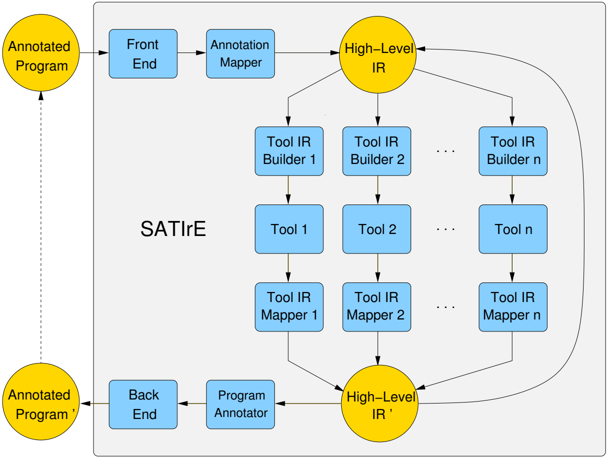
In this course

...we will focus on

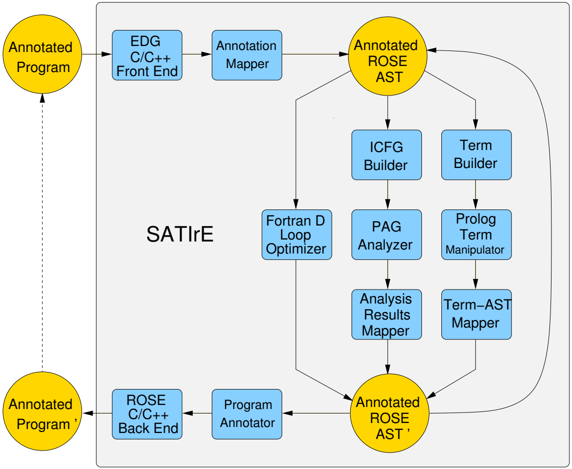
- LLNL-ROSE: Source-to-Source C/C++ Optimization Framework, Lawrence Livermore National Laboratory (LLNL), CA, USA, http://rosecompiler.org/
- SATIrE: Static Analysis and Tool Integration Engine, TU Vienna, Austria, http://www.complang.tuwien.ac.at/satire/
- PAG: Program Analysis Generator, AbsInt Angewandte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

SATIrE: Abstract Architecture

Static Analysis and Tool Integration Engine (SATIrE)



SATIrE: Concrete Architecture



(Oct'07)

SATIrE Components (1)

Basic components

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

SATIrE Components (2)

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

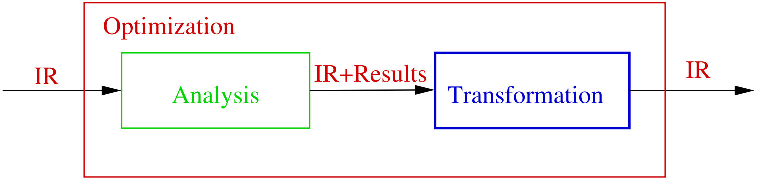
SATIrE Components (3)

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

Chapter 1.7

Summary, Looking Ahead

Optimization: The General Schema



Optimization, a combination of

- Analysis
  - determines properties of program.
  - relies on safe, pessimistic assumptions.
- Transformation
  - based on analysis results.
  - must preserve the program semantics, i.e., the observable program behaviour.

Program Analysis: The Essence (1)

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

- Safe: faithful to the semantics
- Efficient: implementation with
  - good time performance
  - low space consumption

Program Analysis:The Essence (2)

Important Approaches for Program Analysis

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

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

Assessing the Power/Success of Optimization

...via validation and/or verification.

Validation

- Experimentally: Benchmark Suite(s)
  - General purpose suites (ACET-focused): SPEC (Standard Performance Evaluation Corporation), Dhrystone, Whetstone,...
  - Special purpose suites (WCET-focused): TACLe (EU FP7 COST Action "Timing Analysis on Code Level"), Märlardalen,...

Verification

- Analytically: Formal Program and Cost Models
  - Rigorous mathematical proving

Key Issues in Optimization

...and in this lecture:

Optimal

- Program Analysis
- Program Transformation

...and based thereon:

Optimal Optimization

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

Chapter 1.8

References, Further Reading

Further Reading for Chapter 1 (1)

Compilers

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

Further Reading for Chapter 1 (2)

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

Further Reading for Chapter 1 (3)

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

Further Reading for Chapter 1 (4)

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

Optimizing Compilers

- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 1, Compiler Challenges for High-Performance Architectures)
- Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 1, Overview; Chapter 2, Compiler Structure)

Further Reading for Chapter 1 (5)

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

Compiler Implementation

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

Further Reading for Chapter 1 (6)

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

Further Reading for Chapter 1 (7)

Program Analysis

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

Further Reading for Chapter 1 (8)

Optimization

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

Further Reading for Chapter 1 (9)

Miscellaneous

- Clement A. Baker-Finch, Kevin Glynn, Simon L. Peyton Jones. *Constructed Product Result Analysis for Haskell*. Journal of Functional Programing 14(2):211-245, 2004.
- Melvin E. Conway. *Proposal for an UNCOL*. Communications of the ACM 1(3):5, 1958.

On-line Textbooks

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

Further Reading for Chapter 1 (10)

On-line Resources of Compilers and Compiler Writing Tools

- German National Research Center for Information Technology, Fraunhofer Institute for Computer Architecture and Software Technology. *The Catalog of Compiler Construction Tools*, 1996-2006. `http://catalog.compilertools.net/`
- Compilers.net Team. *Search Machine on Compilers and Programming Languages, Directory of Compiler and Language Resources*, 1997-2007. `http://www.compilers.net`

Further Reading for Chapter 1 (11)

- Nullstone Corporation. *The Compiler Connection: A Resource for Compiler Developers and Those who use Their Products and Services (Books, Tools, Techniques, Conferences, Jobs and more)*, 2011-2012.
  http://www.compilerconnection.com
- Olaf Langmack. *Catalog of Compiler Construction Products 01-98*. 13th Issue, 1998.
  http://compilers.iecc.com/tools.html
- William M. Waite, Uwe Kastens et al. *Eli: Translator Construction made Easy*, 1989-today.
  http://eli-project.sourceforge.net/

Further Reading for Chapter 1 (12)

- Free Software Foundation (FSF). *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/
- LLVM Foundation. *The LLVM Compiler Infrastructure*. http://llvm.org/
- The SUIF Group (Monica S. Lam et al.), Stanford University. *The SUIF (Stanford University Intermediate Format) Compiler System*. http://suif.stanford.edu/
- Sable Research Group (Laurie Hendren et al.), McGill University, Secure Software Engineering Group (Eric Bodden et al.), TU Darmstadt/U. Paderborn. *Soot: A Framework for Analyzing and Transforming Java and Android Applications*. https://sable.github.io/soot/

Chapter 2

Classical Gen/Kill Data Flow Analyses

Classical Gen/Kill Data Flow Analyses (1)

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

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

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

Classical Gen/Kill Data Flow Analyses (2)

...are classifyable according to the direction of the information flow:

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

Classical Gen/Kill Data Flow Analyses (3)

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

Forward Problems

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

Classical Gen/Kill Data Flow Analyses (4)

Backward Problems

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

In the following sections

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

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

Chapter 2.1

Programs, Flow Graphs

Formalising the Development

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

WHILE Language

Syntactic categories

a	$\in$ 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	$\in \mathsf{Op}_a$	arithmetic operators
op_b	$\in \mathsf{Op}_b$	boolean operators
op_r	$\in \mathsf{Op}_r$	relational operators

Abstract Syntax

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

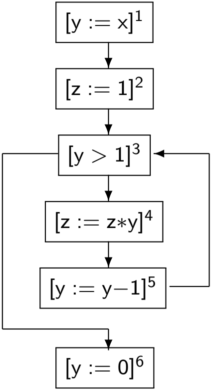
Assignments and tests are (uniquely) labelled to allow analyses to refer to these program fragments – the labels correspond to pointers into the syntax tree. We use abstract syntax and insert parentheses to disambiguate syntax.

We will often refer to labelled fragments as elementary blocks.

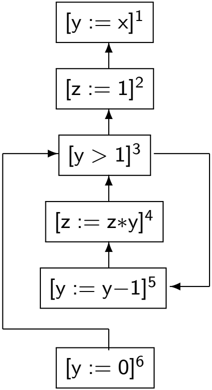
A Program and its Flow Graph

Example:

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



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



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

Auxiliary Functions for Flow Graphs

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

Computing the Auxiliary Information (1)

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

Computing the Auxiliary 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	$\{(\ell, \mathsf{init}(S))\} \cup \mathsf{flow}(S) \cup \{(\ell', \ell) \mid \ell' \in \mathsf{final}(S)\}$	$\{[b]^\ell\} \cup \mathsf{blocks}(S)$

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

Further Notations (1)

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

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

Further Notations (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:

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

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

Illustration

Example:

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

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

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.

Chapter 2.2

Forward Analyses

Chapter 2.2.1

Reaching Definitions

Reaching Definitions Analysis

Definition 2.2.1.1 (Reaching Definitions)

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

Reaching Definitions Analysis

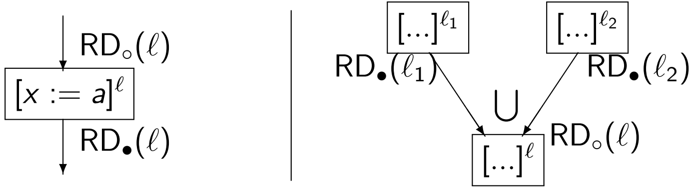
...determines for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.

Example:

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

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

RD Analysis Information and Characteristics



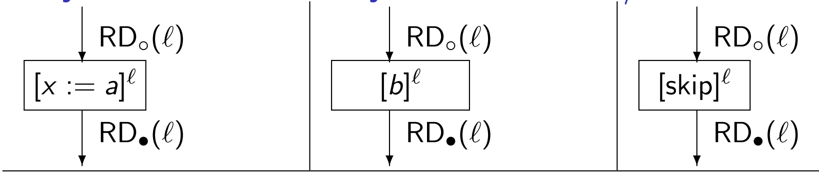
Analysis information: $RD_\circ(\ell), RD_\bullet(\ell) : \mathbf{Lab}_\star \rightarrow \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}^?_\star)$

- $RD_\circ(\ell)$: the definitions that reach entry of block ℓ.
- $RD_\bullet(\ell)$: the definitions that reach exit of block ℓ.

Analysis characteristics:

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

Analysis of Elementary Blocks: Gen/Kill-Defs.



$$
\begin{aligned}
\text{gen}_{RD}([x := a]^\ell) &= \{(x,\ell)\}\\
\text{gen}_{RD}([b]^\ell) &= \emptyset\\
\text{gen}_{RD}([\text{skip}]^\ell) &= \emptyset
\end{aligned}
$$

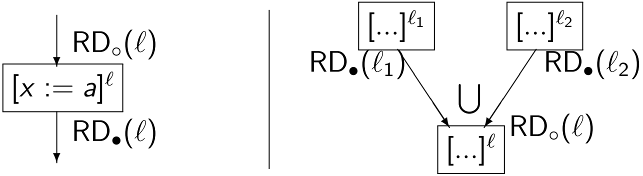
$$
\begin{aligned}
\text{kill}_{RD}([x := a]^\ell) &= \{(x,?)\} \cup \{(x,\ell') \mid B^{\ell'} \text{is assignment to } x\}\\
\text{kill}_{RD}([b]^\ell) &= \emptyset\\
\text{kill}_{RD}([\text{skip}]^\ell) &= \emptyset
\end{aligned}
$$

Example:

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

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

Analysis of the Program: The RD Equations



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

Illustration

Example:

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

Equations: Let

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

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

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

Solving the RD Equations: The Algorithm (1)

Input

- A set of reaching definitions equations

Output

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

Data structures

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

Solving the RD Equations: The Algorithm (2)

```
W:=nil;
foreach (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
```

Application/Usage of RD Information

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

- Use-Definition chains or *ud* chains

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

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

- Definition-Use chains or *du* chains

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

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

UD/DU Chains: Defined via RDs

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

are defined by

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

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

and

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

Chapter 2.2.2

Available Expressions

Available Expressions Analysis

Definition 2.2.2.1 (Available Expressions)

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

Available Expression Analysis

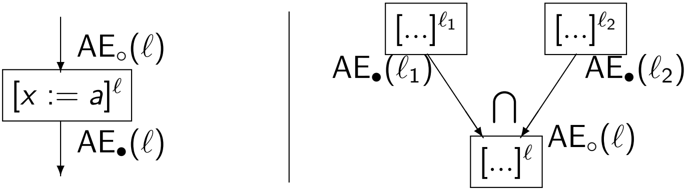
...determines for each program point, which expressions must have already been computed, and not later modified, on all paths to the program point.

Example:

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

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

AE Analysis Information and Characteristics



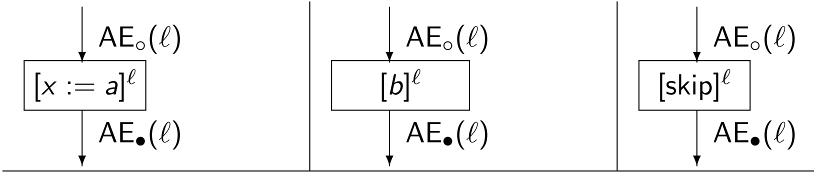
Analysis information: $\mathsf{AE}_\circ(\ell), \mathsf{AE}_\bullet(\ell) : \mathsf{Lab}_\star \to \mathcal{P}(\mathsf{AExp}_\star)$

- $\mathsf{AE}_\circ(\ell)$: the expressions that have been comp. at entry of block ℓ.
- $\mathsf{AE}_\bullet(\ell)$: the expressions that have been comp. at exit of block ℓ.

Analysis characteristics:

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

Analysis of Elementary Blocks: Gen/Kill-Defs.



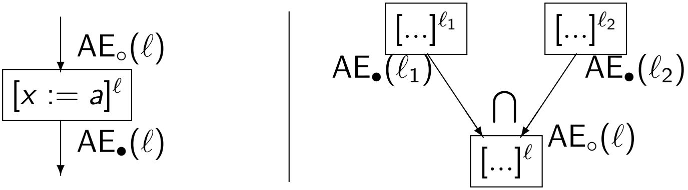
$$
\begin{aligned}
\text{gen}_{AE}([x := a]^\ell) &= \{a' \in \text{AExp}(a) \mid x \notin FV(a')\}\\
\text{gen}_{AE}([b]^\ell) &= \text{AExp}(b)\\
\text{gen}_{AE}([\text{skip}]^\ell) &= \emptyset
\end{aligned}
$$

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

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

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

Analysis of the Program: The AE Equations



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

Illustration

Example:

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

Equations:

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

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

Solving AE Equations: The Algorithm (1)

Input

- A set of available expressions equations

Output

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

Data structures

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

Solving AE Equations: The Algorithm (2)

```
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
```

1st Application/Usage of AE-Information

Common Subexpression Elimination (CSE)

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

Example:

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

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

The Optimization: CSE

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

For each $[...a...]^\ell$ in $S_\star^N$ with $a \in \mathrm{AE}_\circ(\ell)$ 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 $[...a...]^\ell$
- create a fresh variable u and
  - replace each occurrence of $[y_i := a]^{\ell_i}$ with $[u := a]^{\ell_i}; [y_i := u]^{\ell_i'}$ for $1 \leq i \leq k$
  - replace $[...a...]^\ell$ with $[...u...]^\ell$

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

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

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

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

Illustration: CSE

Example:

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

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

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

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

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

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

Copy Analysis

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

Example:

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

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

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

Copy Propagation

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

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

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

The Optimization: Copy Propagation

For each copy statement $[x := y]^{\ell_j}$ in $S_\star$ 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 $\{(x', y') \in \mathsf{C}_\circ(\ell_i) \mid x' = x\} = \{(x, y)\}$; 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 $1 \leq i \leq k$.

Illustration: Copy Propagation (1)

Example 1

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

becomes after Copy Propagation

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

Illustration: Copy Propagation (2)

Example 2

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

becomes after Copy Propagation

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

Example 3

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

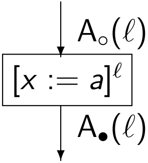
becomes after Copy Propagation

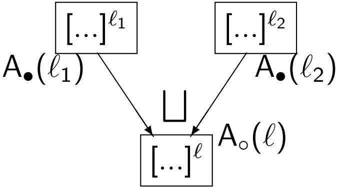
$[a := 10]^1;$; while $[a > 1]^3$ do $[a := a - 1]^4;$; od $[y := y*a]^6;$

Chapter 2.2.3

Summary: Forward Analyses

Summary: Forward Analyses, RD and AE (1)





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

where

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

Summary: Forward Analyses, RD and AE (2)

This means effect functions of blocks are of the form

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

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

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

Chapter 2.2.4

References, Further Reading

Further Reading for Chapter 2.2

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

Chapter 2.3

Backward Analyses

Chapter 2.3.1

Live Variables

Live Variable Analysis

Definition 2.3.1.1 (Live Variables)

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

Live Variables Analysis

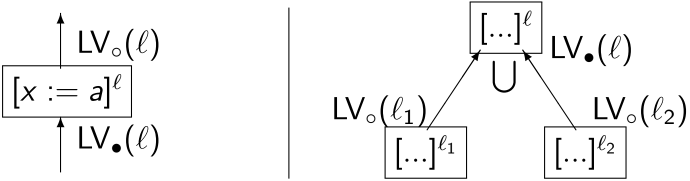
...determines for each program point, which variables may be live at the exit from the point.

Example

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

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

LV Analysis Information and Characteristics



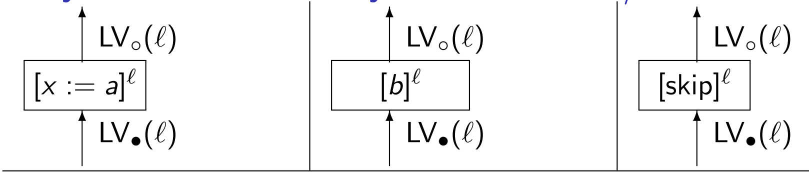
Analysis information: $\mathsf{LV}_\circ(\ell), \mathsf{LV}_\bullet(\ell) : \mathsf{Lab}_\star \to \mathcal{P}(\mathsf{Var}_\star)$

- $\mathsf{LV}_\circ(\ell)$: 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 characteristics:

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

Analysis of Elementary Blocks: Gen/Kill-Defs.



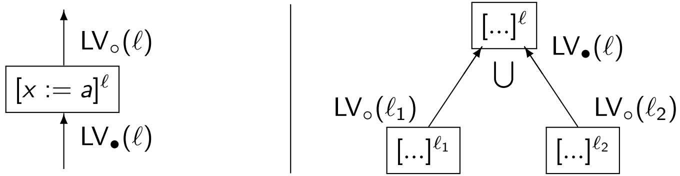
$$
\begin{aligned}
gen_{LV}([x := a]^\ell) &= FV(a)\\
gen_{LV}([b]^\ell) &= FV(b)\\
gen_{LV}([\text{skip}]^\ell) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
kill_{LV}([x := a]^\ell) &= \{x\}\\
kill_{LV}([b]^\ell) &= \emptyset\\
kill_{LV}([\text{skip}]^\ell) &= \emptyset
\end{aligned}
$$

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

- $gen_{LV}([u := a+b]^1)=\{a,b\}$
- $kill_{LV}([u := a+b]^1)=\{u\}$

Analysis of the Program: The LV Equations



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

Illustration

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}
while$[y > u]^3$do	{a, b, y}	{u, a, b, y}
$[a := a + 1]^4$;	{a, b, y}	{a, b, y}
$[u := a + b]^5$;	{u, a, b, y}	{a, b, y}
$[x := u]^6$ od	{u, a, b, y}	{u, a, b, y}
$[skip]^7$	$\emptyset$	$\emptyset$

ℓ	$kill_{LV}(\ell)$	$gen_{LV}(\ell)$
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	$\emptyset$	$\emptyset$

Application/Usage of LV Information

Dead Code Elimination (DCE):

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

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

Example:

Before DCE:

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

After DCE:

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

Combining Optimizations

...usually strengthens the overall impact.

Example:

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

1. Common Subexpression Elimination gives

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

2. Copy Propagation gives

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

3. Dead Code Elimination gives

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

What are the results for other optimization sequences?

Faint Variables

...generalize the notion of dead variables.

Consider the following program consisting of three statements:
$[x := 1]^1; [x := 2]^2; [y := x]^3;$

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

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

Chapter 2.3.2

Very Busy Expressions

Very Busy Expressions Analysis

Definition 2.3.2.1 (Very Busy Expressions)

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

Very Busy Expression Analysis

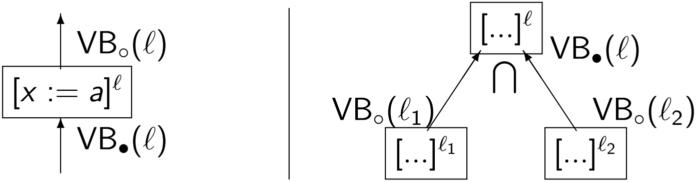
...determines for each program point, which expressions must be very busy at the exit from the point.

Example

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

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

VB Analysis Information and Characteristics



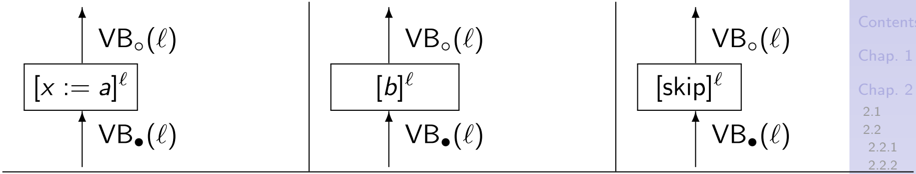
Analysis information: $\mathsf{VB}_\circ(\ell), \mathsf{VB}_\bullet(\ell) : \mathsf{Lab}_\star \to \mathcal{P}(\mathsf{AExp}_\star)$

- $\mathsf{VB}_\circ(\ell)$: 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 characteristics:

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

Analysis of Elementary Blocks: Gen/Kill-Defs.



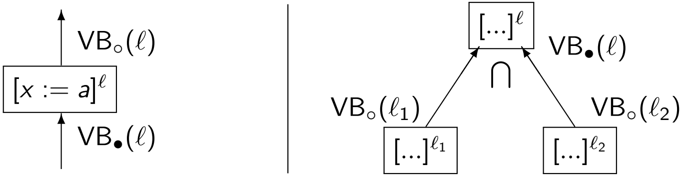
$$
\begin{aligned}
\mathrm{gen}_{VB}([x := a]^\ell) &= \mathrm{AExp}(a)\\
\mathrm{gen}_{VB}([b]^\ell) &= \mathrm{AExp}(b)\\
\mathrm{gen}_{VB}([\mathsf{skip}]^\ell) &= \emptyset
\end{aligned}
$$

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

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

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

Analysis of the Program: The VB Equations



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

Illustration

Example

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

ℓ	$VB_\bullet(\ell)$	$VB_\circ(\ell)$
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	$\emptyset$	$\{a{-}b\}$

Application/Usage of VB Information

Code Hoisting

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

Example:

Before Code Hoisting:

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

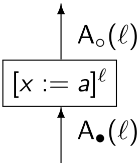
After Code Hoisting:

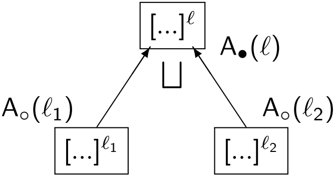
$[t1 := a - b]^0; [t2 := b - a]^{0'}$;
if $[a > b]^1$ then $([x := t2]^2; [y := t1]^3)$ else $([y := t2]^4; [x := t1]^5)$

Chapter 2.3.3

Summary: Backward Analyses

Summary: Backward Analyses, LV and VB (1)





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

where

Analysis	LV	VB
ι_A	$\emptyset$	$\emptyset$
$\bigsqcup_A$	$\cup$	$\cap$

Summary: Backward Analyses, LV and VB (2)

This means effect functions of blocks are of the form

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

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

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

Chapter 2.3.4

References, Further Reading

Further Reading for Chapter 2.3

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

Chapter 2.4

Taxonomy of Gen/Kill Analyses

Taxonomy of Gen/Kill Analyses

Analysis	may (existential)	must (universal)
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Analysis	may (existential)	must (universal)
Combination Op.	$\cup$	$\cap$
Solution of equ.	smallest	largest

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

Chapter 2.5

Summary, Looking Ahead

Gen/Kill Data Flow Analyses

...are also known as

- Bitvector Data Flow Analyses.

This notion refers to a common implementation strategy for

- Gen/Kill Data Flow Analyses.

Bit Vectors and Bit Vector Analyses

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

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

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

Then we have efficient implementations of

- set union as logical 'or'
- set intersection as logical 'and'

More Bit Vector Framework Examples

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

Some Non-Bit Vector Framework Examples (1)

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

Some Non-Bit Vector Framework Examples (2)

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

Flow-Sensitive/Flow-Insensitive DFA Problems

...another categorization of DFA problems and analyses:

- Flow-sensitive problems and analyses
  - The validity of a property at some program point depends on the control flow path(s) involving it.

    E.g., Gen/Kill Problems (RD, AE, LV, VB, etc.), constant propagation and folding, partial redundancy elimination, etc.
- Flow-insensitive problems and analyses
  - The validity of a property at some program point is independent of the control flow path(s) involving it.

    E.g., type analysis (for many programming languages, e.g., C but not Ruby), Procedure_X_Can_Modify_Variable_V, Procedure_X_Can_Have_Side_Effects, etc.

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

Outlook

Gen/Kill Data Flow Analyses

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

Chapter 2.6

References, Further Reading

Further Reading for Chapter 2 (1)

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

Further Reading for Chapter 2 (2)

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

Part II

Intraprocedural Data Flow Analysis

Chapter 3

The Intraprocedural DFA Framework

Chapter 3.1

Preliminaries

Outlook

Next, we (re-) consider:

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

Flow Graphs

Definition 3.1.1 (Flow Graph)

A (non-deterministic) flow graph is a quadruple tuple $G = (N, E, \mathbf{s}, \mathbf{e})$ with

- node set N
- edge set $E \subseteq N \times N$
- distinguished start node $\mathbf{s}$ w/out any predecessors
- distinguished end node $\mathbf{e}$ w/out any successors

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

Node-labelled vs. Edge-labelled Flow Graphs

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

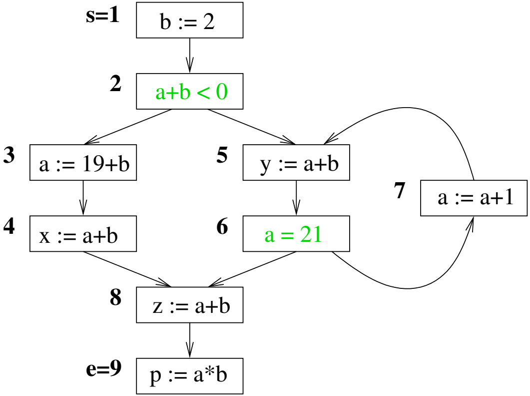
- nodes
- edges

Depending on the choice this leads to

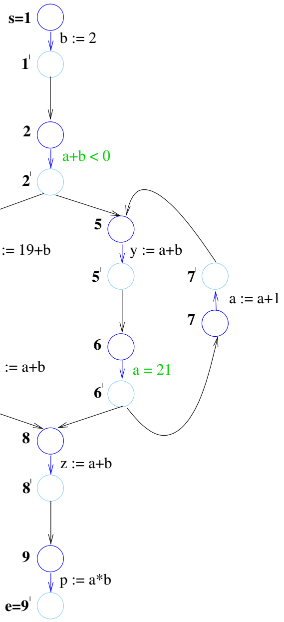
- node-labelled flow graphs
- edge-labelled flow graphs

respectively.

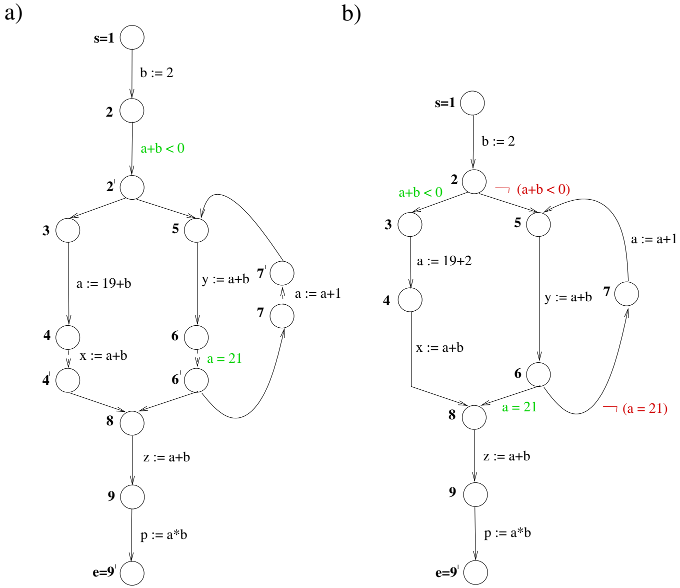
A Node-Labelled Flow Graph



An Edge-Labelled Flow Graph



Edge-Labelled Flow Graph after Cleaning Up



Reverse Flow Graph

Definition 3.1.2 (Reverse Flow Graph)

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

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

Note

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

In the following

...we consider

- edge-labelled flow graphs

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

- Appendix B: Pragmatics of Flow Graph Representations

Notations for Flow Graphs (1)

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

Predecessor and Successor Nodes

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

Notations for Flow Graphs (2)

Paths

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

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

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

Partially Ordered Sets, Complete Lattices

Definition 3.1.3 (Partially Ordered Set)

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

Definition 3.1.4 (Lattice, Complete Lattice)

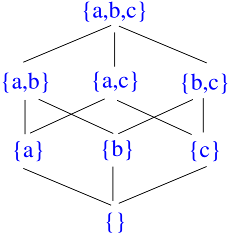
Let $(P, \sqsubseteq)$ be a partially ordered set.

Then $(P, \sqsubseteq)$ is a

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

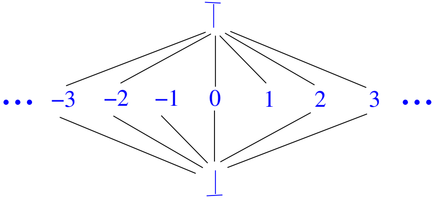
Examples: Complete Lattices

a)

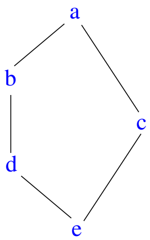


b)

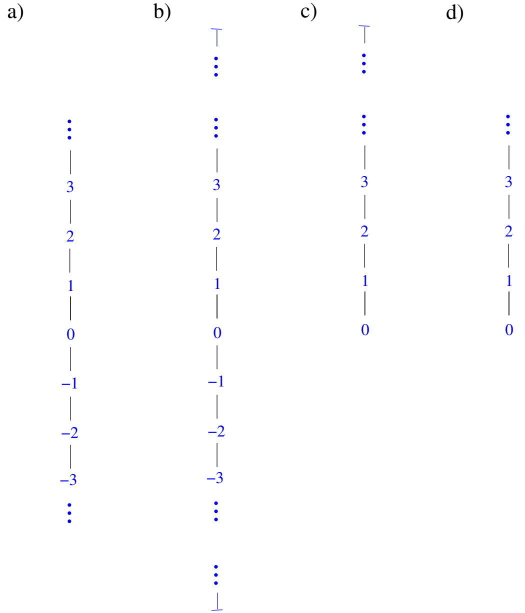
c)



d)



Examples: Partially Ordered Sets and Lattices



Notations for Lattices

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

- $\bigsqcap C'$ denotes the greatest lower bound of C'.
- $\bigsqcup C'$ denotes the least upper bound of C'.
- $\bot =_{df} \bigsqcap C = \bigsqcup \emptyset$ denotes the least element of C.
- $\top =_{df} \bigsqcup C = \bigsqcap \emptyset$ denotes the greatest element of C.

This gives rise to write a complete lattice as a quintuple

- $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$

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

Descending, Ascending Chain Condition

Definition 3.1.5 (Chain Condition)

Let $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$ be a lattice.

$\widehat{C}$ satisfies the

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

Monotonicity, Distributivity, and Additivity

...are important properties of functions on lattices:

Definition 3.1.6 (Monotonicity)

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

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

Definition 3.1.7 (Distributivity, Additivity)

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

- distributive iff $\forall\, C' \subseteq \mathcal{C}.\ \ f(\bigsqcap C') = \bigsqcap \{ f(c) \mid c \in C' \}$
  (Preservation of greatest lower bounds)
- additive iff $\forall\, C' \subseteq \mathcal{C}.\ \ f(\bigsqcup C') = \bigsqcup \{ f(c) \mid c \in C' \}$
  (Preservation of least upper bounds)

Characterizing Monotonicity

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

Lemma 3.1.8

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

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

Useful Results

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

Lemma 3.1.9

f is distributive iff f is additive.

Lemma 3.1.10

f is monotonic if f is distributive (additive).

Chapter 3.2

DFA Specification, DFA Problem

DFA Specification

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

Definition 3.2.1 (DFA Specification)

A DFA specification for G is a quadruple $\mathcal{S}_G = (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, d)$ with

- $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$ a complete lattice
- $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$ a local abstract semantics
- $c_{\mathbf{s}} \in \mathcal{C}$ an initial information/assertion
- $d \in \{fw, bw\}$ a direction of information flow

Note:

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

Notations for DFA Specifications

Let $\mathcal{S}_G = (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, d)$ be a DFA specification for G.

Then

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

Thus

- $\widehat{C}$ is called a DFA lattice.
- $[\![\ ]\!]$ is called a DFA functional.
- $[\![\, e \,]\!]$, $e \in E$, is called a (local) DFA function.

DFA Problem

Definition 3.2.2 (DFA Problem)

A DFA specification $\mathcal{S}_G = (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, d)$ defines a DFA problem for G.

Practically Relevant DFA Problems

...DFA problems are practically relevant, if they are

- monotonic
- distributive (additive)

and satisfy the

- descending (ascending) chain condition.

Properties of DFA Functionals

Definition 3.2.3 (Properties of DFA Functionals)

Let $\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, d)$ be a DFA specification for G.

The DFA functional $[\![\ ]\!] : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ of $\mathcal{S}_G$ is

- monotonic/distributive/additive

iff for every $e \in E$ the local DFA function $[\![\, e \,]\!]$ is

- monotonic/distributive/additive, respectively.

Properties of DFA Problems

Definition 3.2.4 (Properties of DFA Problems)

Let $\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!], c_s, d)$ be a DFA specification for G.

The DFA problem induced by $\mathcal{S}_G$

- is monotonic/distributive/additive iff the DFA functional $[\![\ ]\!]$ of $\mathcal{S}_G$ is monotonic/distributive/additive.
- satisfies the descending (ascending) chain condition iff the DFA lattice $\widehat{C}$ of $\mathcal{S}_G$ satisfies the descending (ascending) chain condition.

Towards a Global Abstract Semantics

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

Actually, we introduce two globalization approaches:

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

Chapter 3.3

The Meet Over All Paths Approach

The Meet Over All Paths (*MOP*) Approach

Let $\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, fw)$ be a DFA specification.

Definition 3.3.1 (Extending $[\![\ ]\!]$ to Paths)

The DFA functions $[\![\, e \,]\!]$, $e \in E$, are extended onto paths $p = \langle e_1, e_2 \ldots, e_q \rangle$ in G by defining:

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

where $Id_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ denotes the identical mapping on $\mathcal{C}$, i.e., $Id_{\mathcal{C}}(c) = c$, $c \in \mathcal{C}$.

The Meet Over All Paths (*MOP*) Solution

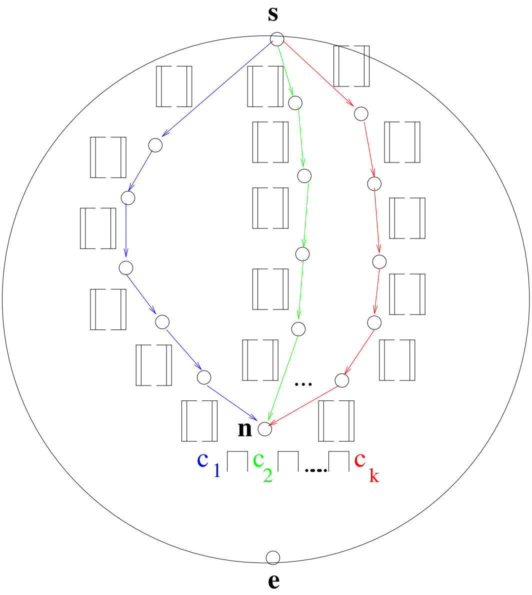
Definition 3.3.2 (The *MOP* Solution)

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

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

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

Illustrating *MOP* Approach and *MOP* Solution



The Specifying Solution of a DFA Problem

...as illustrated by the previous figure:

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

This gives rise to consider the *MOP* solution the

- specifying solution of a DFA problem.

Conservative and Optimal DFA Algorithms

Definition 3.3.4 (Conservative DFA Algorithm)

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

Definition 3.3.5 (Optimal DFA Algorithm)

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

Unfortunately

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

- effective computation procedure

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

Even worse, the *MOP* solution is

- not even decidable!

Undecidability of the *MOP* Solution

Theorem 3.3.3 (Undecidability, Kam&Ullman 1977)

There is no algorithm A satisfying:

- The input of A are
  - a DFA specification $\mathcal{S}_G = (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, fw)$
  - algorithms for the computation of the meet, the equality test, and the application of monotonic functions on the elements of a complete lattice
- The output of A is the *MOP* solution of $\mathcal{S}_G$.

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

Towards a Conservative and Optimal DFA Alg.

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

- Maximum Fixed Point (*MaxFP*) Approach.

The *MaxFP* approach leads to the

- Maximum Fixed Point (*MaxFP*) Solution

of a DFA problem and an

- effective computation procedure

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

Chapter 3.4

The Maximum Fixed Point Approach

The Maximum Fixed Point (*MaxFP*) Approach

Let $\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, fw)$ be a DFA specification.

Equation System 3.4.1 (*MaxFP* Equation System)

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

Let

- $inf^{\star}_{c_{\mathbf{s}}}(n),\; n \in N$

denote the greatest solution of Equation System 3.4.1.

The Maximum Fixed Point (*MaxFP*) Solution

Definition 3.4.2 (The *MaxFP* Solution)

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

$$MaxFP_{\mathcal{S}_G} : N \to \mathcal{C}$$

$$\forall\, n \in N.\ MaxFP_{\mathcal{S}_G}(n) =_{df} inf^{\star}_{c_s}(n)$$

The *MaxFP* Approach

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

- iterative computation procedure (Algorithm 3.4.3)

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

The Generic Fixed Point Algorithm 3.4.3 (1)

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

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

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

- $[\![\ ]\!]$ distributive: $inf[n]$ stores
- $[\![\ ]\!]$ monotonic: $inf[n]$ stores a lower approximation of

the *MOP* solution of $\mathcal{S}_G$ at node n.

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

The Generic Fixed Point Algorithm 3.4.3 (2)

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

FORALL $n \in N \backslash \{\mathbf{s}\}$ DO $inf[n] := \top$ OD;
$inf[\mathbf{s}] := c_{\mathbf{s}}$;
$workset := N$;

(Main loop: The iterative fixed point computation)

```
WHILE workset ≠ ∅ DO
    CHOOSE m ∈ workset;
      workset := workset \ { m };
      ( Updating 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.
```

Termination

Theorem 3.4.4 (Termination)

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

1. $[\![\ ]\!]$ is monotonic
2. $\widehat{\mathcal{C}}$ satisfies the descending chain condition.

The Computable Solution of a DFA Problem

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

- computable solution of a DFA problem.

Flow Sensitivity and May/Must Problems

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

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

Obviously, the Generic Fixed Point Algorithm 3.4.3 is formulated for

- must forward problems.

This raises the question

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

Uniform Handling of All Four Problem Kinds

The Generic Fixed Point Algorithm 3.4.3 allows us to handle instances of all four problem kinds uniformly:

- must/forward: directly
- may/forward: defining $\sqsubseteq$ in terms of $\sqsupseteq$
- must/backward: using G_{rev} instead of G
- may/backward: using G_{rev} instead of G and definining $\sqsubseteq$ in terms of $\sqsupseteq$

...this will be illustrated in detail in Chapter 4.

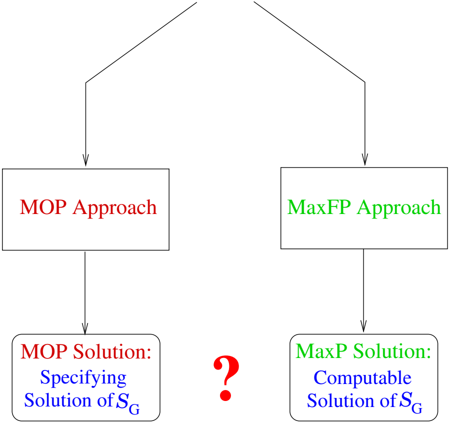
Chapter 3.5

Safety and Coincidence

MOP / *MaxFP* Solution of a DFA Specification

...how are they related?

$$\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, fw)$$



Safety

Theorem 3.5.1 (Safety)

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

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

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

Coincidence

Theorem 3.5.2 (Coincidence)

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

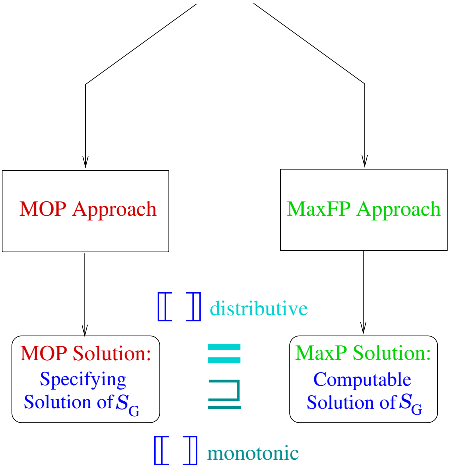
$$\forall\, n \in N.\ \mathit{MaxFP}_{\mathcal{S}_G}(n) = \mathit{MOP}_{\mathcal{S}_G}(n)$$

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

MOP / MaxFP Solution of a DFA Specification

...and their relationship:

$$\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, fw)$$



Conservativity, Optimality of Algorithm 3.4.3

Corollary 3.5.3 (*MOP* Conservativity)

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

Corollary 3.5.4 (*MOP* Optimality)

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

Chapter 3.6

Soundness and Completeness

Soundness and Completeness (1)

Analysis Scenario:

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

Definition 3.6.1 (Soundness)

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

Definition 3.6.2 (Completeness)

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

Soundness and Completeness (2)

Intuitively

- Soundness means: $MOP_{\mathcal{S}_G^\phi}$ implies ϕ.
- Completeness means: ϕ implies $MOP_{\mathcal{S}_G^\phi}$.

Soundness and Completeness (3)

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

We compute

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

In other words

- We compute the program property of interest accurately!

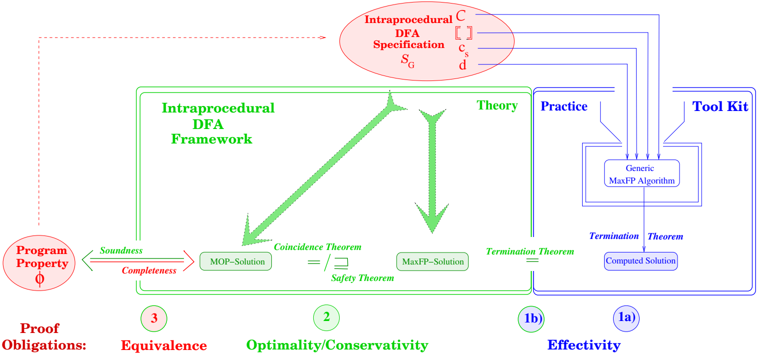
Chapter 3.7

A Uniform Framework and Toolkit View to DFA

Intraprocedural DFA: A Holistic Uniform View

...considering intraprocedural DFA from a holistic angle:

- A Uniform Framework and Toolkit View



Intraprocedural DFA in Practice

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

The Three-Stage Process

1. Identifying a Program Property of Interest

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

2. Designing a DFA Specification

   Design a DFA specification $\mathcal{S}_G^\phi = (\widehat{\mathcal{C}}, [\![\ ]\!], c_s, d)$ for ϕ.

3. Accomplishing Proof Obligations, Obtaining Guarantees

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

Proof Obligations and Guarantees (1)

Proof obligations and guarantees in detail:

- Proof Obligations 1a), 1b): Descending Chain Condition for $\widehat{\mathcal{C}}$, Monotonicity for $[\![\ ]\!]$

  Guarantees:

  - Effectivity: Termination of Algorithm 3.4.3 with the *MaxFP* solution of $\mathcal{S}_G^{\phi}$.
  - Conservativity: The *MaxFP* solution of $\mathcal{S}_G^{\phi}$ is *MOP* conservative.

- Proof Obligation 2): Distributivity for $[\![\ ]\!]$

  Guarantee:

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

Proof Obligations and Guarantees (2)

- Proof Obligation 3): Equivalence of $MOP_{\mathcal{S}_G^\phi}$ and ϕ

  Guarantees:

  - Whenever the *MOP* solution of $\mathcal{S}_G^\phi$ indicates the validity of ϕ, then it is valid: Soundness.

    $\leadsto$ We compute the property of interest, and only the property of interest.

  - Whenever ϕ is valid, this is indicated by the *MOP* solution of $\mathcal{S}_G^\phi$: Completeness.

    $\leadsto$ We compute the whole property of interest.

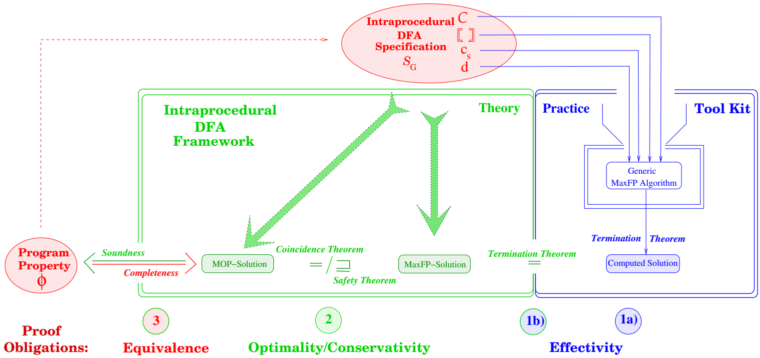
  Guarantee of combined Soundness and Completeness:

  - We compute program property ϕ accurately!

Chapter 3.8

Summary, Looking Ahead

The Holistic View to Intraprocedural DFA



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

Reconsidering Correctness and Precision

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

Framework/Toolkit **internally**: captured by

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

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

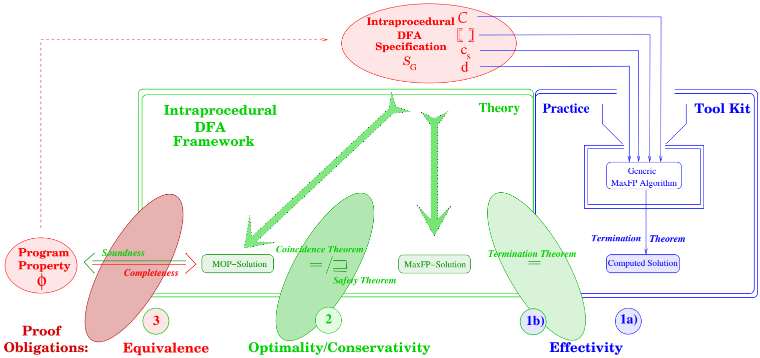
Framework/Toolkit **externally**: captured by

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

...relating *MOP* solution and ϕ.

Illustrating

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

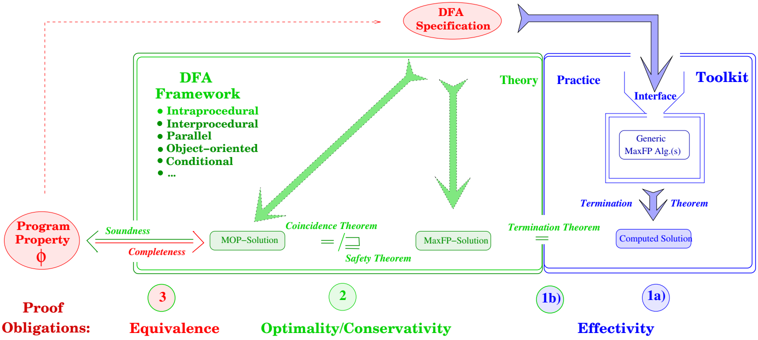


Outlook: The Holistic Uniform View to DFA

In the course of this lecture, we will see

- The Uniform Framework and Toolkit View of DFA

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



Next

...we will consider applications of the intraprodural DFA framework for

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

Chapter 3.9

References, Further Reading

Further Reading for Chapter 3 (1)

Textbook Representations

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

Further Reading for Chapter 3 (2)

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

Further Reading for Chapter 3 (3)

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

Further Reading for Chapter 3 (4)

Pioneering, Groundbreaking Articles

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

Further Reading for Chapter 3 (5)

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

Frameworks and Toolkits

- Marion Klein, Jens Knoop, Dirk Koschützki, Bernhard Steffen. *DFA&OPT-METAFrame: A Toolkit for Program Analysis and Optimization*. In Proceedings of the 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), Springer-V., LNCS 1055, 422-426, 1996.
- Jens Knoop. *From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Springer-V., LNCS 1579, 360-374, 1999.

Further Reading for Chapter 3 (6)

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

Further Reading for Chapter 3 (7)

Solving Equation Systems, Computing Fixed Points

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

Further Reading for Chapter 3 (8)

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

Flow Graph Pragmatics

Jens Knoop, Dirk Koschützki, Bernhard Steffen. *Basic-block Graphs: Living Dinosaurs?* In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 65-79, 1998.

Further Reading for Chapter 3 (9)

Miscellaneous

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

- Janusz Laski, William Stanley. *Software Verification and Analysis*. Springer-V., 2009. (Chapter 7, What can one tell about a Program without its Execution: Static Analysis)

Chapter 4

Gen/Kill Analyses Reconsidered

Gen/Kill Analyses

...are common examples of

- distributive DFA problems.

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

- Intraprocedural DFA Framework of Chapter 3

using

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

for illustration.

Remarks

Note that

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

can be dealt with analogously.

Throughout Chapter 4, let

- $G = (N, E, \mathbf{s}, \mathbf{e})$

be an edge-labelled flow graph.

Chapter 4.1

Reaching Definitions

Chapter 4.1.1

Reaching Definitions for a Single Definition

Scenario 1: The Setting (1)

...reaching definitions for a single definition $Def(v@\hat{e}) \equiv d^v_{\hat{e}}$.

Lattice

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

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

Special Functions

- Constant Functions $Cst_{true}, Cst_{false} : \mathbb{B} \to \mathbb{B}$

  $\forall\, b \in \mathbb{B}.\; Cst_{true}(b) =_{df} true$

  $\forall\, b \in \mathbb{B}.\; Cst_{false}(b) =_{df} false$

- Identity $Id_{\mathbb{B}} : \mathbb{B} \to \mathbb{B}$

  $\forall\, b \in \mathbb{B}.\; Id_{\mathbb{B}}(b) =_{df} b$

Scenario 1: The Setting (2)

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

Local Predicates

- $At_e^{\hat{e}}$

  ...*true*, if $e = \hat{e}$, otherwise *false*.

- Mod_e^v

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

Scenario 1: DFA Specification

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df}$
  $$(\mathrm{IB}, \vee, \wedge, \geq, \mathit{true}, \mathit{false}) = \widehat{\mathrm{IB}_\vee}$$
- DFA functional
  $[\![\ ]\!]_{rd}^{d_{\hat{e}}^{v}} : E \to (\mathrm{IB} \to \mathrm{IB})$ where
  $\forall\, e \in E\ \forall\, b \in \mathrm{IB}.\ [\![\, e \,]\!]_{rd}^{d_{\hat{e}}^{v}}(b) =_{df} (b \wedge \neg Mod_{e}^{v}) \vee At_{e}^{\hat{e}}$
- Initial information: $b_{\mathbf{s}} \in \mathrm{IB}$
- Direction of information flow: forward

Reaching Definitions Specification for $d_{\hat{e}}^{v}$

- Specification: $\mathcal{S}_{G}^{rd, d_{\hat{e}}^{v}} = (\widehat{\mathrm{IB}_\vee}, [\![\ ]\!]_{rd}^{d_{\hat{e}}^{v}}, b_{\mathbf{s}}, fw)$

Lemma 4.1.1.1 (Data Flow Functions)

$$\forall\, e \in E.\ [\![\ e\ ]\!]^{d^v_{\hat{e}}}_{rd} = \begin{cases} Cst_{true} & \text{if } At^{\hat{e}}_e \\ Id_{\mathbb{B}} & \text{if } \neg At^{\hat{e}}_e \wedge \neg Mod^{\,v}_e \\ Cst_{false} & \text{otherwise} \end{cases}$$

Lemma 4.1.1.2 (Descending Chain Condition)

$\widehat{\mathbb{B}_{\vee}}$ satisfies the descending chain condition (wrt $\sqsupseteq =_{df} \leq$).

Lemma 4.1.1.3 (Distributivity)

$[\![\ ]\!]^{d^v_{\hat{e}}}_{rd}$ is distributive (wrt $\sqcap =_{df} \vee$).

Proof. Immediately with Lemma 4.1.1.1.

Corollary 4.1.1.4 (Monotonicity)

$[\![\ ]\!]^{d^v_{\hat{e}}}_{rd}$ is monotonic.

Termination and Optimality

Theorem 4.1.1.5 (Termination)

Applied to $\mathcal{S}_G^{rd,d_{\hat{e}}^v} = (\widehat{\mathrm{IB}_\vee}, [\![\ ]\!]_{rd}^{d_{\hat{e}}^v}, b_{\mathbf{s}}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{rd,d_{\hat{e}}^v}$.

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

Theorem 4.1.1.6 (Optimality)

Applied to $\mathcal{S}_G^{rd,d_{\hat{e}}^v} = (\widehat{\mathrm{IB}_\vee}, [\![\ ]\!]_{rd}^{d_{\hat{e}}^v}, b_{\mathbf{s}}, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{rd,d_{\hat{e}}^v}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{rd,d_{\hat{e}}^v}$).

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

Chapter 4.1.2

Reaching Definitions for a Set of Definitions

Scenario 2: The Setting

...reaching definitions for a set of definitions

$$\{d^{v_1}_{\hat{e}_1}, \ldots, d^{v_k}_{\hat{e}_k}\} \equiv \mathcal{D}^V_{\hat{E}},\ k \in \mathbb{N}.$$

Lattice

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

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

Scenario 2: DFA Specification

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df}$
  $$(\mathcal{P}(\mathcal{D}^V_{\hat{E}}), \cup, \cap, \supseteq, \mathcal{D}^V_{\hat{E}}, \emptyset) = \widehat{\mathcal{P}(\mathcal{D}^V_{\hat{E}})_\cup}$$
- DFA functional
  $[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd} : E \rightarrow (\ \mathcal{P}(\mathcal{D}^V_{\hat{E}}) \rightarrow \mathcal{P}(\mathcal{D}^V_{\hat{E}})\ )$ where
  $\forall\, e \in E\ \forall\, \mathcal{D} \in \mathcal{P}(\mathcal{D}^V_{\hat{E}}).\ [\![\, e\,]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd}(\Delta) =_{df}$
  $$\{d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}} \mid (d^v_{\hat{e}} \in \mathcal{D} \wedge \neg Mod^{\,v}_{\,e}) \vee At^{\hat{e}}_e\}$$
- Initial information: $\mathcal{D}_\mathbf{s} \in \mathcal{P}(\mathcal{D}^V_{\hat{E}})$
- Direction of information flow: forward

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

- Specification: $\mathcal{S}^{rd,\mathcal{D}^V_{\hat{E}}}_G = (\widehat{\mathcal{P}(\mathcal{D}^V_{\hat{E}})_\cup}, [\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd}, \mathcal{D}_\mathbf{s}, fw)$

Towards Termination and Optimality

Lemma 4.1.2.1 (Descending Chain Condition)

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

Lemma 4.1.2.2 (Distributivity)

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

Corollary 4.1.2.3 (Monotonicity)

$[\![\ ]\!]_{rd}^{\mathcal{D}_{\hat{E}}^{V}}$ is monotonic.

Termination and Optimality

Theorem 4.1.2.4 (Termination)

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

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

Theorem 4.1.2.5 (Optimality)

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

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

Chapter 4.1.3

Reaching Definitions for a Set of Definitions: Bitvector Implementation

Scenario 3: The Setting (1)

...reaching definitions for a set of definitions

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

Lattice

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

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

Scenario 3: The Setting (2)

Auxiliary Functions

- $var : \mathcal{D}^V_{\hat{E}} \rightarrow V$, $edge : \mathcal{D}^V_{\hat{E}} \rightarrow E$ defined by $\forall\, d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}}.\ var(d^v_{\hat{e}}) =_{df} v,\ edge(d^v_{\hat{e}}) =_{df} \hat{e}$
- $ix : \mathcal{D}^V_{\hat{E}} \rightarrow \{1, \ldots, n\},\ \ ix^{-1} : \{1, \ldots, n\} \rightarrow \mathcal{D}^V_{\hat{E}}$

  ...bijective mappings which map every definition $d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}}$ to a number in $\{1, \ldots, n\}$ and vice versa.

  The $ix(d^v_{\hat{e}})^{th}$ element of an element $\bar{b} = (b_1, \ldots, b_{ix(d^v_{\hat{e}})}, \ldots, b_n) \in \mathbb{B}^n$ is the reaching definitions information for $d^v_{\hat{e}}$ stored in $\bar{b}$.
- $\cdot \downarrow_\cdot : \mathbb{B}^n \rightarrow \{1, \ldots, n\} \rightarrow \mathbb{B}$

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

Scenario 3: DFA Specification

DFA Specification

- DFA lattice
  $$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df}$$
  $$(\mathrm{IB}^n, \vee_{pw}, \wedge_{pw}, \geq_{pw}, \overline{true}, \overline{false}) = \widehat{\mathrm{IB}^n_{\vee}}$$
- DFA functional
  $$[\![\ ]\!]_{rd,cps}^{\mathcal{D}^V_{\hat{E}}} : E \to (\,\mathrm{IB}^n \to \mathrm{IB}^n\,) \text{ where}$$
  $$\forall\, e \in E \ \forall\, d^v_{\hat{e}} \in \mathrm{IB}^n.\ [\![\, e\,]\!]_{rd,cps}^{\mathcal{D}^V_{\hat{E}}}(\bar{b}) =_{df} \bar{b}'$$
  where $\forall\, i \in \{1, \ldots, n\}.\ \bar{b}'\!\downarrow_i \ =_{df}$
  $$(\bar{b}\!\downarrow_i \ \wedge \neg Mod_e^{var(ix^{-1}(i))}) \vee At_e^{edge(ix^{-1}(i))}$$
- Initial information: $\bar{b}_{\mathbf{s}} \in \mathrm{IB}^n$
- Direction of information flow: forward

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

- Specification: $\mathcal{S}_G^{rd,\mathcal{D}^V_{\hat{E}},cps} = (\widehat{\mathrm{IB}^n_{\vee}}, [\![\ ]\!]_{rd,cps}^{\mathcal{D}^V_{\hat{E}}}, \bar{b}_{\mathbf{s}}, fw)$

Scenario 3: Bitvector Implementation (1)

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

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

Scenario 3: Bitvector Implementation (2)

Auxiliary Functions

- $V_{\mathcal{D}} =_{df} \{ var(d^v_{\hat{e}}) \mid d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}} \}$,
  $\hat{E}_{\mathcal{D}} =_{df} \{ edges(d^v_{\hat{e}}) \mid d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}} \}$
- $ix : \mathcal{D}^V_{\hat{E}} \to \{1, \ldots, n\}, \;\; ix^{-1} : \{1, \ldots, n\} \to \mathcal{D}^V_{\hat{E}}$

  ...bijective mappings which map every definition $d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}}$ to a number in $\{1, \ldots, n\}$ and vice versa.

  The $ix(d^v_{\hat{e}})^{th}$ element of a bitvector $\vec{bv} = [d_1, \ldots, d_{ix(d^v_{\hat{e}})}, \ldots, d_n)] \in \mathcal{BV}^n$ is the reaching definitions information for $d^v_{\hat{e}}$ stored in $\vec{bv}$.

Scenario 3: Bitvector Implementation (3)

Extending and Transforming Local Predicates to Bitvectors

- $\overrightarrow{Mod}_e^{V_{\mathcal{D}}} \in \mathcal{BV}^n$

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

- $\overrightarrow{At}_e^{\hat{E}_{\mathcal{D}}} \in \mathcal{BV}^n$

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

Scenario 3: Bitvector Implementation (4)

Bitvector Implementation

- DFA lattice
$$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df}$$
$$(\mathcal{BV}^n, max_{\mathcal{BV}}, min_{\mathcal{BV}}, \geq_{\mathcal{BV}}, \vec{1}, \vec{0}) = \widehat{\mathcal{BV}^n_{max}}$$
- DFA functional
$[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,bvi} : E \to (\mathcal{BV}^n \to \mathcal{BV}^n)$ where
$$\forall\, e \in E\ \forall\, \vec{bv} \in \mathcal{BV}^n.\ [\![e]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,bvi}(\vec{bv}) =_{df}$$
$$(\vec{bv}\ min_{\mathcal{BV}}\ \neg \overrightarrow{Mod}^{V_{\mathcal{D}}}_e)\ max_{\mathcal{BV}}\ \overrightarrow{At}^{\hat{E}_{\mathcal{D}}}_e$$
- Initial information: $\vec{bv}_{\mathbf{s}} \in \mathcal{BV}^n$
- Direction of information flow: forward

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

- Specification: $\mathcal{S}^{rd,\mathcal{D}^V_{\hat{E}},bvi}_G = (\widehat{\mathcal{BV}^n_{max}}, [\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,bvi}, \vec{bv}_{\mathbf{s}}, fw)$

Towards Termination and Optimality

Lemma 4.1.3.1 (Descending Chain Condition)

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

Lemma 4.1.3.2 (Distributivity)

$[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,bvi}$ is distributive (wrt $\sqcap =_{df} max_{\mathcal{BV}}$).

Corollary 4.1.3.3 (Monotonicity)

$[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,bvi}$ is monotonic.

Termination

Theorem 4.1.3.4 (Termination)

Applied to $\mathcal{S}_G^{rd,\mathcal{D}_{\hat{E}}^V,bvi} = (\widehat{\mathcal{BV}^n_{max}}, [\![\ ]\!]_{rd,bvi}^{\mathcal{D}_{\hat{E}}^V}, \vec{bv}_{\mathsf{s}}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{rd,\mathcal{D}_{\hat{E}}^V,bvi}$.

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

Optimality

Theorem 4.1.3.5 (Optimality)

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

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

Note

- All results of Chapter 4.1.3 hold for $\mathcal{S}_G^{rd,\mathcal{D}_{\hat{E}}^V,cps} = (\widehat{\mathbb{B}_\vee^n}, [\![\ ]\!]_{rd,cps}^{\mathcal{D}_{\hat{E}}^V}, \bar{b}_{\mathbf{s}}, fw)$, too.
- Applied to $\mathcal{S}_G^{rd,\mathcal{D}_{\hat{E}}^V,bvi} = (\widehat{\mathcal{BV}_{max}^n}, [\![\ ]\!]_{rd,bvi}^{\mathcal{D}_{\hat{E}}^V}, \vec{bv}_{\mathbf{s}}, fw)$, Algorithm 3.4.3 takes advantage of the efficient bitvector operations of actual processors.

Chapter 4.1.4

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

Scenario 4: The Setting

...reaching definitions for a set of definitions

$$\{d^{v_1}_{\hat{e}_1}, \ldots, d^{v_k}_{\hat{e}_k}\} \equiv \mathcal{D}^V_{\hat{E}},\ k \in \mathbb{N}.$$

Defining Gen/Kill Predicates

- $Gen_e^{\mathcal{D}^V_{\hat{E}}} =_{df} \{d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}} \mid At^{\hat{e}}_e\}$
- $Kill_e^{\mathcal{D}^V_{\hat{E}}} =_{df} \{d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}} \mid Mod^{\,v}_e\}$

Scenario 4: DFA Specification

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df}$
  $$(\mathcal{P}(\mathcal{D}_{\hat{E}}^V), \cup, \cap, \supseteq, \mathcal{D}_{\hat{E}}^V, \emptyset) = \widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_\cup$$
- DFA functional
  $[\![\ ]\!]_{rd,gk}^{\mathcal{D}_{\hat{E}}^V} : E \to (\, \mathcal{P}(\mathcal{D}_{\hat{E}}^V) \to \mathcal{P}(\mathcal{D}_{\hat{E}}^V)\,)$ where
  $\forall\, e \in E\ \forall\, \mathcal{D} \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V).\ [\![\, e\,]\!]_{rd,gk}^{\mathcal{D}_{\hat{E}}^V}(\mathcal{D}) =_{df}$
  $$(\mathcal{D} \backslash Kill_e^{\mathcal{D}_{\hat{E}}^V}) \cup Gen_e^{\mathcal{D}_{\hat{E}}^V}$$
- Initial information: $\mathcal{D}_\mathbf{s} \in \mathcal{P}(\mathcal{D}_{\hat{E}}^V)$
- Direction of information flow: forward

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

- Specification: $\mathcal{S}_G^{rd,\mathcal{D}_{\hat{E}}^V,gk} = (\widehat{\mathcal{P}(\mathcal{D}_{\hat{E}}^V)}_\cup, [\![\ ]\!]_{rd,gk}^{\mathcal{D}_{\hat{E}}^V}, \mathcal{D}_\mathbf{s}, fw)$

Towards Termination and Optimality

Compare

- $[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,gk} : E \to (\,\mathcal{P}(\mathcal{D}^V_{\hat{E}}) \to \mathcal{P}(\mathcal{D}^V_{\hat{E}})\,)$ where

  $\forall\, e \in E\ \forall\, \mathcal{D} \in \mathcal{P}(\mathcal{D}^V_{\hat{E}}).\ [\![\, e\,]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,gk}(\mathcal{D}) =_{df}$

  $$(\mathcal{D} \backslash Kill_e^{\mathcal{D}^V_{\hat{E}}}) \cup Gen_e^{\mathcal{D}^V_{\hat{E}}}$$

- $[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd} : E \to (\,\mathcal{P}(\mathcal{D}^V_{\hat{E}}) \to \mathcal{P}(\mathcal{D}^V_{\hat{E}})\,)$ where

  $\forall\, e \in E\ \forall\, \mathcal{D} \in \mathcal{P}(\mathcal{D}^V_{\hat{E}}).\ [\![\, e\,]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd}(\mathcal{D}) =_{df}$

  $$\{ d^v_{\hat{e}} \in \mathcal{D}^V_{\hat{E}} \mid (d^v_{\hat{e}} \in \mathcal{D} \wedge \neg Mod^{\,v}_{e}) \vee At^{\hat{e}}_{e} \}$$

Obviously

Lemma 4.1.4.1 (Equality)

$$[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd} = [\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,gk}$$

Termination and Optimality

Theorem 4.1.4.2 (Termination)

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

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

Theorem 4.1.4.3 (Optimality)

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

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

Recalling the *MaxFP* Equation System

Equation System 3.4.1 (*MaxFP* Equation System)

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

The *MinFP* Equation System

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

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

Specializing Equation System 3.4.1$_{min}$ for Reaching Definitions yields:

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

$Reaches(n) =$

$$\begin{cases} \mathcal{D}_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigcup \{ [\![(m,n)]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,gk}(Reaches(m)) \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

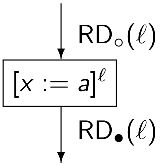
Specializing EQS 3.4.1$_{min}$ for Reaching Def's

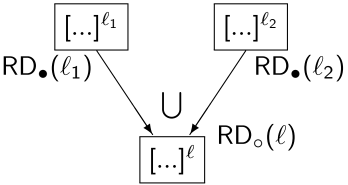
Expanding $[\![\ ]\!]^{\mathcal{D}^V_{\hat{E}}}_{rd,gk}$ in EQS 4.1.4.4 yields:

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

$$Reaches(n) = \begin{cases} \mathcal{D}_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigcup \{ \, (Reaches(m) \backslash Kill^{\mathcal{D}^V_{\hat{E}}}_{(m,n)}) \cup Gen^{\mathcal{D}^V_{\hat{E}}}_{(m,n)} \\ \qquad\qquad\qquad \mid m \in pred(n) \, \} & \text{otherwise} \end{cases}$$

Recalling the RD Analysis of Chapter 2.1.1 (1)





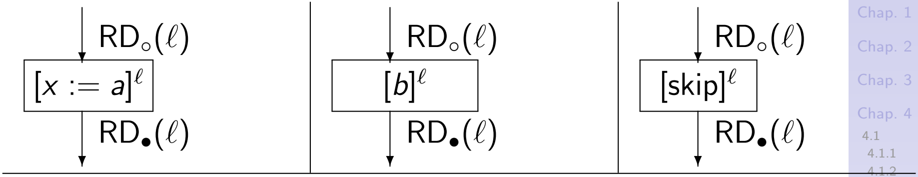
$$
\begin{array}{lcll}
\mathrm{RD}_\circ(\ell) & = & \begin{cases} \emptyset & : \quad \text{if } \ell = \mathrm{init}(S_\star) \\ \bigcup\{\mathrm{RD}_\bullet(\ell') \mid (\ell',\ell) \in \mathrm{flow}(S_\star)\} & : \quad \text{otherwise} \end{cases} & \\
\mathrm{RD}_\bullet(\ell) & = & (\mathrm{RD}_\circ(\ell)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

EQS 4.1.4.5 (EQS 3.4.1$_{min}$ for R'g Def's) – recalled

$Reaches(n) =$

$$
\begin{cases}
\mathcal{D}_{\mathbf{s}} & \text{if } n = \mathbf{s} \\
\bigcup\{\, (Reaches(m) \backslash Kill^{\mathcal{D}^V_{\hat{E}}}_{(m,n)}) \cup Gen^{\mathcal{D}^V_{\hat{E}}}_{(m,n)} \;\mid\; m \in pred(n) \,\} & \text{otherwise}
\end{cases}
$$

Recalling the RD Analysis of Chapter 2.1.1 (2)



$$
\begin{aligned}
\mathrm{gen}_{\mathrm{RD}}([x := a]^{\ell}) &= \{(x, \ell)\}\\
\mathrm{gen}_{\mathrm{RD}}([b]^{\ell}) &= \emptyset\\
\mathrm{gen}_{\mathrm{RD}}([\mathrm{skip}]^{\ell}) &= \emptyset
\end{aligned}
$$

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

Summing up

The Reaching Definitions Equations of

- Equation System 4.1.4.5
- Chapter 2.2.1

are equivalent up to the insignificant formal difference of considering

- edge-labelled
- node-labelled

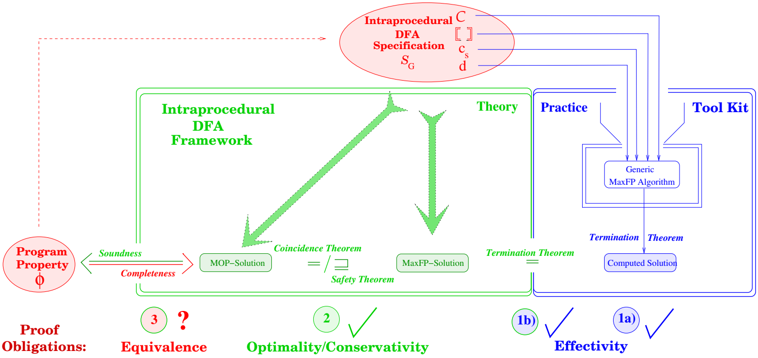
flow graphs, respectively.

Chapter 4.1.5

Reaching Definitions Analysis: Soundness and Completeness

Closing the Proof Final Gap

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



Defining Reaching Definitions Informally

...intuitively:

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

Note

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

Useful Notation

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

We define for paths:

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

We define for paths and predicates:

- $Predicate_p^{\forall} \iff \forall\, 1 \leq i \leq \lambda_p.\ Predicate_{p_i}$
- $Predicate_p^{\exists} \iff \exists\, 1 \leq i \leq \lambda_p.\ Predicate_{p_i}$

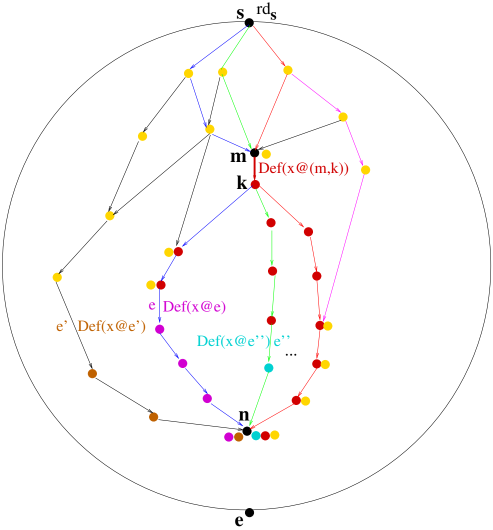
Defining Reaching Definitions Formally

Definition 4.1.5.1 (Reaching Definitions)

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

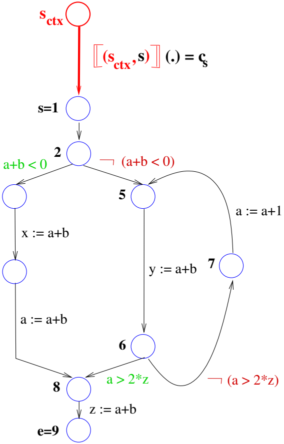
$$Reaches^{d^v_{\hat{e}}}(n) \iff_{df} \begin{cases} rd_\mathbf{s} & \text{if } n = \mathbf{s} \\ \exists\, p \in \mathbf{P}[\mathbf{s}, n]. & \\ \quad (rd_\mathbf{s} \wedge \neg Mod^{v\,\forall}_{\;p}) \;\vee & \\ \quad \exists\, i \leq \lambda_p.\; At^{\hat{e}}_{p_i} \wedge (i = \lambda_p \vee \neg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

Illustrating the Essence of Definition 4.1.5.1

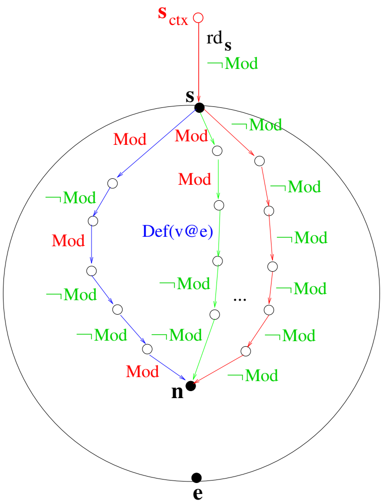


Context Edges

...introducing a context edge would allow a simpler uniform definition of reaching definitions:



Context Edges and Reaching Definitions



...leading to:

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

Closing the Final Proof Gap

Theorem 4.1.5.2 (Soundness and Completeness)

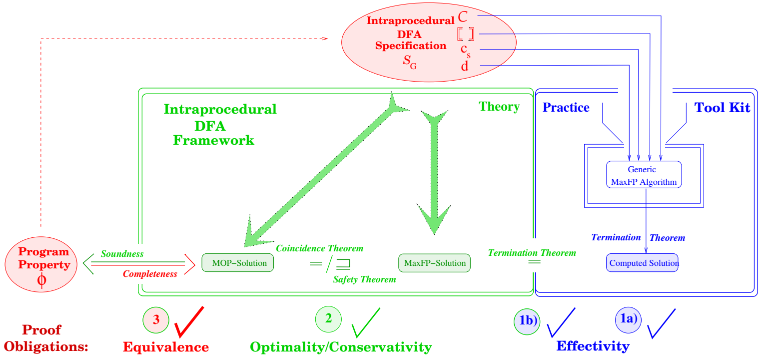
Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let $d^v_{\hat{e}}$ be a definition, let $rd_{\mathbf{s}} \in \mathbb{B}$ be the reaching definitions information at $\mathbf{s}$ ensured by the calling context of G, and let $MOP_{\mathcal{S}_G^{rd,d^v_{\hat{e}}}}$ be the *MOP* solution of the DFA specification $\mathcal{S}_G^{rd,d^v_{\hat{e}}} = (\widehat{\mathbb{B}_\vee}, [\![\ ]\!]^{d^v_{\hat{e}}}_{rd}, rd_{\mathbf{s}}, fw)$.

Then:

$$\forall\, n \in N.\ Reaches^{d^v_{\hat{e}}}(n) \iff MOP_{\mathcal{S}_G^{rd,d^v_{\hat{e}}}}(n)$$

Gap Closed: Soundness&Completeness Proven

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



Chapter 4.2

Very Busy Expressions

Chapter 4.2.1

Very Busyness for a Single Term

Scenario 1: The Setting (1)

...very busyness for a single term t.

Lattice

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

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

Special Functions

- Constant Functions $Cst_{true}, Cst_{false} : \mathrm{IB} \to \mathrm{IB}$

  $\forall\, b \in \mathrm{IB}.\ Cst_{true}(b) =_{df} \mathit{true}$
  $\forall\, b \in \mathrm{IB}.\ Cst_{false}(b) =_{df} \mathit{false}$

- Identity $Id_{\mathrm{IB}} : \mathrm{IB} \to \mathrm{IB}$

  $\forall\, b \in \mathrm{IB}.\ Id_{\mathrm{IB}}(b) =_{df} b$

Scenario 1: The Setting (2)

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

Local Predicates

- $Comp_e^t$

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

- Mod_e^t

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

- $Transp_e^t =_{df} \neg Mod_e^t$

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

Scenario 1: DFA Specification

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathbb{B}, \wedge, \vee, \leq, \mathit{false}, \mathit{true}) = \widehat{\mathbb{B}}$
- DFA functional
  $[\![\ ]\!]^t_{vb} : E \to (\mathbb{B} \to \mathbb{B})$ where
  $\forall\, e \in E\ \forall\, b \in \mathbb{B}.\ [\![e]\!]^t_{vb}(b) =_{df} (b \wedge \mathit{Transp}^t_e) \vee \mathit{Comp}^t_e$
- Initial information: $b_{\mathbf{e}} \in \mathbb{B}$
- Direction of information flow: backward

Very Busyness Specification for t

- Specification: $\mathcal{S}^{vb,t}_G = (\widehat{\mathbb{B}}, [\![\ ]\!]^t_{vb}, b_{\mathbf{e}}, bw)$

Towards Termination and Optimality

Lemma 4.2.1.1 (Data Flow Functions)

$$\forall\, e \in E.\ [\![\, e \,]\!]^t_{vb} = \begin{cases} Cst_{true} & \text{if } Comp^t_e \\ Id_{\mathbb{B}} & \text{if } \neg Comp^t_e \wedge Transp^t_e \\ Cst_{false} & \text{otherwise} \end{cases}$$

Lemma 4.2.1.2 (Descending Chain Condition)

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

Lemma 4.2.1.3 (Distributivity)

$[\![\ ]\!]^t_{vb}$ is distributive.

Proof. Immediately with Lemma 4.2.1.1.

Corollary 4.2.1.4 (Monotonicity)

$[\![\ ]\!]^t_{vb}$ is monotonic.

Termination and Optimality

Theorem 4.2.1.5 (Termination)

Applied to $\mathcal{S}_G^{vb,t} = (\widehat{\mathsf{IB}}, [\![\ ]\!]_{vb}^t, b_{\mathbf{e}}, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb,t}$.

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

Theorem 4.2.1.6 (Optimality)

Applied to $\mathcal{S}_G^{vb,t} = (\widehat{\mathsf{IB}}, [\![\ ]\!]_{vb}^t, b_{\mathbf{e}}, bw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{vb,t}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{vb,t}$).

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

Chapter 4.2.2

Very Busyness for a Set of Terms

Scenario 2: The Setting

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

Lattice

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

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

Scenario 2: DFA Specification

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{P}(T), \cap, \cup, \subseteq, \emptyset, T) = \widehat{\mathcal{P}(T)}$
- DFA functional
  $[\![\ ]\!]_{vb}^{T} : E \to (\mathcal{P}(T) \to \mathcal{P}(T))$ where
  $\forall\, e \in E\ \forall\, T' \in \mathcal{P}(T).\ [\![\, e\,]\!]_{vb}^{T}(T') =_{df}$
  $\{t \in T \mid (t \in T' \wedge Transp_e^t) \vee Comp_e^t\}$
- Initial information: $T_{\mathbf{e}} \in \mathcal{P}(T)$
- Direction of information flow: backward

Very Busyness Specification for T

- Specification: $\mathcal{S}_G^{vb,T} = (\widehat{\mathcal{P}(T)}, [\![\ ]\!]_{vb}^{T}, T_{\mathbf{e}}, bw)$

Towards Termination and Optimality

Lemma 4.2.2.1 (Descending Chain Condition)

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

Lemma 4.2.2.2 (Distributivity)

$[\![\ ]\!]_{vb}^{T}$ is distributive.

Corollary 4.2.2.3 (Monotonicity)

$[\![\ ]\!]_{vb}^{T}$ is monotonic.

Termination and Optimality

Theorem 4.2.2.4 (Termination)

Applied to $\mathcal{S}_G^{vb,T} = (\widehat{\mathcal{P}(T)}, [\![\ ]\!]_{vb}^T, T_{\mathbf{e}}, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb,T}$.

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

Theorem 4.2.2.5 (Optimality)

Applied to $\mathcal{S}_G^{vb,T} = (\widehat{\mathcal{P}(T)}, [\![\ ]\!]_{vb}^T, T_{\mathbf{e}}, bw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{vb,T}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{vb,T}$).

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

Chapter 4.2.3

Very Busyness for a Set of Terms: Bitvector Implementation

Scenario 3: The Setting (1)

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

Lattice

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

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

Scenario 3: The Setting (2)

Auxiliary Functions

- $ix : T \to \{1, \ldots, n\}, \; ix^{-1} : \{1, \ldots, n\} \to T$

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

  The $ix(t)^{th}$ element of an element
  $\bar{b} = (b_1, \ldots, b_{ix(t)}, \ldots, b_n) \in \mathbb{B}^n$
  is the very busyness information for t stored in $\bar{b}$.

- $\cdot \downarrow_{\cdot} : \mathbb{B}^n \to \{1, \ldots, n\} \to \mathbb{B}$

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

Scenario 3: DFA Specification

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df}$
  $$(\mathbb{B}^n, \wedge_{pw}, \vee_{pw}, <_{pw}, \overline{false}, \overline{true}) = \widehat{\mathbb{B}^n}$$
- DFA functional
  $[\![\ ]\!]^T_{vb,cps} : E \to (\mathbb{B}^n \to \mathbb{B}^n)$ where
  $\forall\, e \in E\ \forall\, v \in \mathbb{B}^n.\ [\![\, e\,]\!]^T_{vb,cps}(\bar{b}) =_{df} \bar{b}'$
  where $\forall\, i \in \{1, \ldots, n\}.\ \bar{b}'\!\downarrow_i =_{df}$
  $$(\bar{b}\!\downarrow_i\ \wedge Transp_e^{ix^{-1}(i)}) \vee Comp_e^{ix^{-1}(i)}$$
- Initial information: $\bar{b}_{\mathbf{e}} \in \mathbb{B}^n$
- Direction of information flow: backward

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

- Specification: $\mathcal{S}_G^{vb,T,cps} = (\widehat{\mathbb{B}^n}, [\![\ ]\!]^T_{vb,cps}, \bar{b}_{\mathbf{e}}, bw)$

Scenario 3: Bitvector Implementation (1)

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

- $\widehat{\mathbb{B}^n}$ can efficiently be implemented in terms of bitvectors $\vec{bv} = [d_1, \ldots, d_n]$, $d_i \in \{0, 1\}$, $1 \leq i \leq n$, of length n.
- Let $\mathcal{BV}^n$ denote the set of all bitvectors of length n.
- Let $\vec{bv}[i] = d_i$ for all $\vec{bv} = [d_1, \ldots, d_n] \in \mathcal{BV}^n$, $1 \leq i \leq n$.
- Let $\vec{0} =_{df} [0, \ldots, 0] \in \mathcal{BV}^n$ and $\vec{1} =_{df} [1, \ldots, 1] \in \mathcal{BV}^n$.
- Let $min_{\mathcal{BV}}$ and $max_{\mathcal{BV}}$ be the bitwise minimum ("logical $\wedge$") and the bitwise maximum function ("logical $\vee$") over bitvectors, i.e., $\forall\, \vec{bv_1}, \vec{bv_2} \in \mathcal{BV}^n\ \forall\, i \in \{1, \ldots, n\}$.
  - $(\vec{bv_1}\ min_{\mathcal{BV}}\ \vec{bv_2})[i] =_{df} min(\vec{bv_1}[i], \vec{bv_2}[i])$
  - $(\vec{bv_1}\ max_{\mathcal{BV}}\ \vec{bv_2})[i] =_{df} max(\vec{bv_1}[i], \vec{bv_2}[i])$

Scenario 3: Bitvector Implementation (2)

Auxiliary Functions

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

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

  The $ix(t)^{th}$ element of a bitvector
  $\vec{bv} = [d_1, \ldots, d_{ix(t)}, \ldots, d_n)] \in \mathcal{BV}^n$
  is the very busyness information for t stored in $\vec{bv}$.

Scenario 3: Bitvector Implementation (3)

Extending and Transforming Local Predicates to Bitvectors

- $\overrightarrow{Comp_e^T} \in \mathcal{BV}^n$

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

- $\overrightarrow{Transp_e^T} \in \mathcal{BV}^n$

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

Scenario 3: Bitvector Implementation (4)

Bitvector Implementation

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df}$
  $$(\mathcal{BV}^n, min_{\mathcal{BV}}, max_{\mathcal{BV}}, <_{\mathcal{BV}}, \vec{0}, \vec{1}) = \widehat{\mathcal{BV}^n}$$
- DFA functional
  $[\![\ ]\!]^T_{vb,bvi} : E \to (\mathcal{BV}^n \to \mathcal{BV}^n)$ where
  $\forall\, e \in E\ \forall\, \vec{bv} \in \mathcal{BV}^n.\ [\![\, e\,]\!]^T_{vb,bvi}(\vec{bv}) =_{df}$
  $$(\vec{bv}\ min_{\mathcal{BV}}\ \overrightarrow{Transp^T_e})\ max_{\mathcal{BV}}\ \overrightarrow{Comp^T_e}$$
- Initial information: $\vec{bv}_{\mathbf{e}} \in \mathcal{BV}^n$
- Direction of information flow: backward

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

- Specification: $\mathcal{S}^{vb,T,bvi}_G = (\widehat{\mathcal{BV}^n}, [\![\ ]\!]^T_{vb,bvi}, \vec{bv}_{\mathbf{e}}, bw)$

Towards Termination and Optimality

Lemma 4.2.3.1 (Descending Chain Condition)

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

Lemma 4.2.3.2 (Distributivity)

$[\![\ ]\!]^{T}_{vb,bvi}$ is distributive.

Corollary 4.2.3.3 (Monotonicity)

$[\![\ ]\!]^{T}_{vb,bvi}$ is monotonic.

Termination

Theorem 4.2.3.4 (Termination)

Applied to $\mathcal{S}_G^{vb,T,bvi} = (\widehat{\mathcal{BV}^n}, [\![\ ]\!]_{vb,bvi}^T, \vec{bv}_{\mathbf{e}}, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb,T,bvi}$.

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

Optimality

Theorem 4.2.3.5 (Optimality)

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

Proof. Immediately with Lemma 4.2.3.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.2.3.4.

Note

- All results of Chapter 4.2.3 hold for $\mathcal{S}_G^{vb,T,cps} = (\widehat{\mathbb{B}^n}, [\![\ ]\!]_{vb,cps}^T, \bar{b}_{\mathbf{e}}, bw)$, too.
- Applied to $\mathcal{S}_G^{vb,T,bvi} = (\widehat{\mathcal{BV}^n}, [\![\ ]\!]_{vb,bvi}^T, \vec{bv}_{\mathbf{e}}, bw)$, Algorithm 3.4.3 takes advantage of the efficient bitvector operations of actual processors.

Chapter 4.2.4

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

Scenario 4: The Setting

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

Defining Gen/Kill Predicates

- $Gen_e^T =_{df} \{t \in T \mid Comp_e^t\}$
- $Kill_e^T =_{df} \{t \in T \mid Mod_e^t\}$

Note

- $Kill_e^T = \{t \in T \mid \neg Transp_e^t\}$

Scenario 4: DFA Specification

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{P}(T), \cap, \cup, \subseteq, \emptyset, T) = \widehat{\mathcal{P}(T)}$
- DFA functional
  $[\![\ ]\!]^{T}_{vb,gk} : E \rightarrow (\mathcal{P}(T) \rightarrow \mathcal{P}(T))$ where
  $\forall\, e \in E\ \forall\, T' \in \mathcal{P}(T).\ [\![\, e\,]\!]^{T}_{vb,gk}(T') =_{df}$
  $(T' \backslash Kill^{T}_{e}) \cup Gen^{T}_{e}$
- Initial information: $T_{\mathbf{e}} \in \mathcal{P}(T)$
- Direction of information flow: backward

Very Busyness Specification for T

- Specification: $\mathcal{S}^{vb,T,gk}_{G} = (\widehat{\mathcal{P}(T)}, [\![\ ]\!]^{T}_{vb,gk}, T_{\mathbf{e}}, bw)$

Towards Termination and Optimality

Compare

- $[\![\ ]\!]^T_{vb,gk} : E \to (\mathcal{P}(T) \to \mathcal{P}(T))$ where
  $\forall\, e \in E\ \forall\, T' \in \mathcal{P}(T).\ [\![\, e\,]\!]^T_{vb,gk}(T') =_{df}$
  $$(T' \backslash Kill^T_e) \cup Gen^T_e$$
- $[\![\ ]\!]^T_{vb} : E \to (\mathcal{P}(T) \to \mathcal{P}(T))$ where
  $\forall\, e \in E\ \forall\, T' \in \mathcal{P}(T).\ [\![\, e\,]\!]^T_{vb}(T') =_{df}$
  $$\{t \in T \mid (t \in T' \wedge Transp^t_e) \vee Comp^t_e\}$$

Obviously

Lemma 4.2.4.1 (Equality)

$$[\![\ ]\!]^T_{vb} = [\![\ ]\!]^T_{vb,gk}$$

Termination and Optimality

Theorem 4.2.4.2 (Termination)

Applied to $\mathcal{S}_G^{vb,T,gk} = (\widehat{\mathcal{P}(T)}, [\![\ ]\!]_{vb,gk}^T, T_\mathbf{e}, bw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{vb,T,gk}$.

Proof. Immediately with Lemma 4.2.4.1, Corollary 4.2.2.3, and Termination Theorem 3.4.4.

Theorem 4.2.4.3 (Optimality)

Applied to $\mathcal{S}_G^{vb,T,gk} = (\widehat{\mathcal{P}(T)}, [\![\ ]\!]_{vb,gk}^T, T_\mathbf{e}, bw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{vb,T,gk}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{vb,T,gk}$).

Proof. Immediately with Lemma 4.2.4.1, Lemma 4.2.2.2, Coincidence Theorem 3.5.2, and Termination Theorem 4.2.4.2.

Recalling the *MaxFP* Equation System

Equation System 3.4.1 (*MaxFP* Equation System)

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

The Backward *MaxFP* Equation System

Equation System 3.4.1^{bw} (Backward *MaxFP* EQS)

$$inf(n) \;=\; \begin{cases} c_{\mathbf{e}} & \text{if } n = \mathbf{e} \\ \sqcap \{ [\![(n,m)]\!](inf(m)) \mid m \in succ(n) \} & \text{otherwise} \end{cases}$$

Specializing Equation System 3.4.1^{bw} for Very Busyness yields:

EQS 4.2.4.4 (EQS 3.4.1$_{bw}$ for Very Busyness)

$$VeryBusy(n) =$$

$$\begin{cases} T_{\mathbf{e}} & \text{if } n = \mathbf{e} \\ \bigcap \{ [\![(n,m)]\!]^{T}_{vb,gk}(VeryBusy(m)) \mid m \in succ(n) \} & \text{otherwise} \end{cases}$$

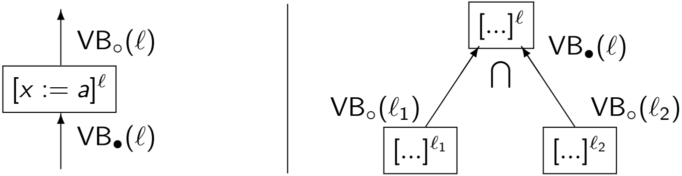
Specializing EQS 3.4.1^{bw} for Very Busyness

Expanding $[\![\ ]\!]^T_{vb,gk}$ in EQS 4.2.4.4 yields:

EQS 4.2.4.5 (EQS 3.4.1^{bw} for Very Busyness)

$$VeryBusy(n) = \begin{cases} T_{\mathbf{e}} & \text{if } n = \mathbf{e} \\ \bigcap \{ \, (VeryBusy(m) \backslash Kill^T_{(n,m)}) \cup Gen^T_{(n,m)} \mid m \in succ(n) \, \} & \text{otherwise} \end{cases}$$

Recalling the VB Analysis of Chapter 2.3.2 (1)



$$\mathsf{VB}_\circ(\ell) \;=\; (\mathsf{VB}_\bullet(\ell)\backslash \mathsf{kill}_{\mathsf{VB}}(B^\ell)) \;\cup\; \mathsf{gen}_{\mathsf{VB}}(B^\ell) \qquad \text{where } B^\ell \in \mathsf{blocks}(S_\star)$$

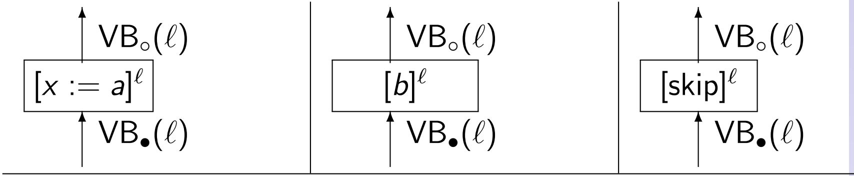
$$\mathsf{VB}_\bullet(\ell) \;=\; \begin{cases} \emptyset & : \;\; \text{if } \ell = \mathsf{final}(S_\star) \\ \bigcap\{\mathsf{VB}_\circ(\ell') \mid (\ell',\ell) \in \mathsf{flow}^R(S_\star)\} & : \;\; \text{otherwise} \end{cases}$$

EQS 4.2.4.5 (EQS 3.4.1^{bw} for V. B'ness) – recalled

$VeryBusy(n) =$

$$\begin{cases} T_{\mathbf{s}} & \text{if } n = \mathbf{e} \\ \bigcap\{\, (VeryBusy(m)\backslash Kill^T_{(n,m)}) \cup Gen^T_{(n,m)} \mid m \in succ(n)\,\} & \text{otherwise} \end{cases}$$

Recalling the VB Analysis of Chapter 2.3.2 (2)



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

Summing up

The Very Busyness Equations of

- Equation System 4.2.4.5
- Chapter 2.3.2

are equivalent up to the insignificant formal difference of considering

- edge-labelled
- node-labelled

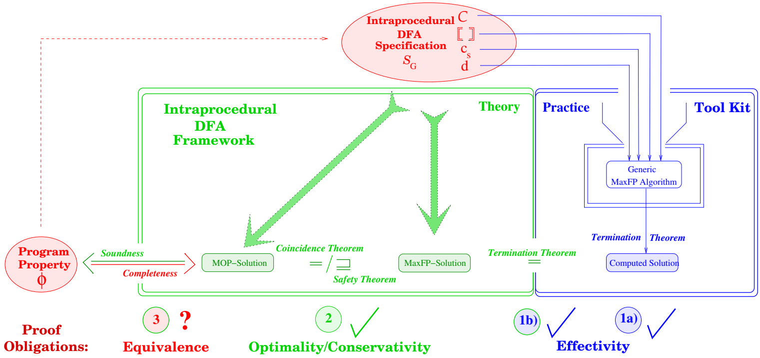
flow graphs, respectively.

Chapter 4.2.5

Very Busyness Analysis: Soundness and Completeness

Closing the Final Proof Gap

...proving soundness and completeness of $\mathcal{S}_G^{vb,t}$ for the very busyness property:



Defining Very Busyness Informally

...intuitively:

- An expression is very busy at a node if, no matter what path is taken from that node to the exit of the program, the expression is computed before any of the variables occurring in it is redefined (cf. Definition 2.3.2.1).

Note

- If exit of the program is replaced by exit of the procedure, the informal “definition” of very busyness does not foresee the possibility of the very busyness of an expression at the procedure exit itself.
- Situations where this very busyness is ensured by the calling context of the procedure, are thus not captured and can not be dealt with.

Useful Notation

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

We define for paths:

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

We define for paths and predicates:

- $Predicate_p^{\forall} \iff \forall\, 1 \leq i \leq \lambda_p.\ Predicate_{p_i}$
- $Predicate_p^{\exists} \iff \exists\, 1 \leq i \leq \lambda_p.\ Predicate_{p_i}$

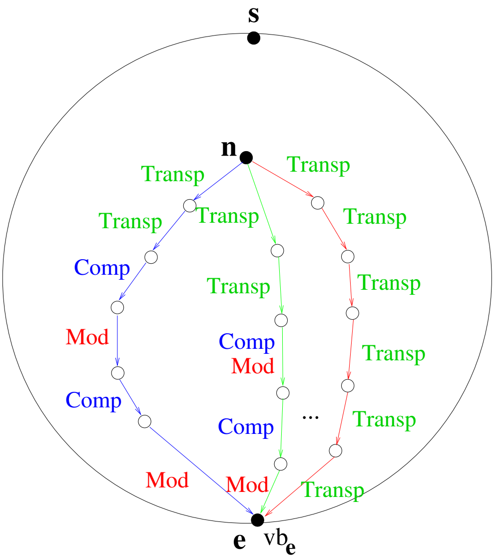
Defining Very Busyness Formally

Definition 4.2.5.1 (Very Busyness)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let t be an expression, let $vb_{\mathbf{e}} \in \mathbb{B}$ be the very busyness information for t at $\mathbf{e}$ ensured by the calling context of G.

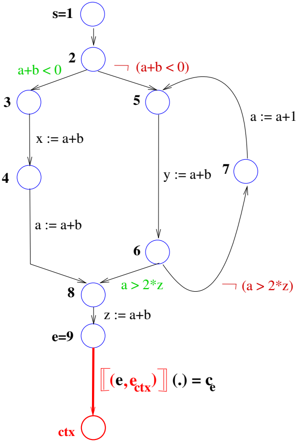
$$VeryBusy^t(n) \iff_{df}$$
$$\begin{cases} vb_{\mathbf{e}} & \text{if } n = \mathbf{e} \\ \forall\, p \in \mathbf{P}[n, \mathbf{e}].\ (vb^t_{\mathbf{e}} \wedge Transp^{t\,\forall}_{\ p}) \ \vee & \\ \qquad\qquad\qquad \exists\, i \leq \lambda_p.\ Comp^t_{p_i} \wedge Transp^{t\,\forall}_{\ p[1,i[} & \text{otherwise} \end{cases}$$

Illustrating the Essence of Definition 4.2.5.1

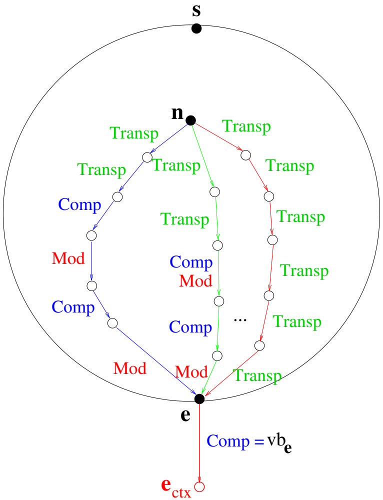


Context Edges

...introducing a context edge would allow a simpler uniform definition of very busyness:



Context Edges and Very Busyness



...leading to:

$$\forall\, n \in N \backslash \{\mathbf{e}_{ctx}\}.\ VeryBusy^t(n) \iff_{df}$$
$$\forall\, p \in \mathbf{P}[n, \mathbf{e}_{ctx}].\ \exists\, i \leq \lambda_p.\ Comp^t_{p_i} \wedge Transp^{t\,\forall}_{\ p[1,i[}$$

Closing the Final Proof Gap

Theorem 4.2.5.2 (Soundness and Completeness)

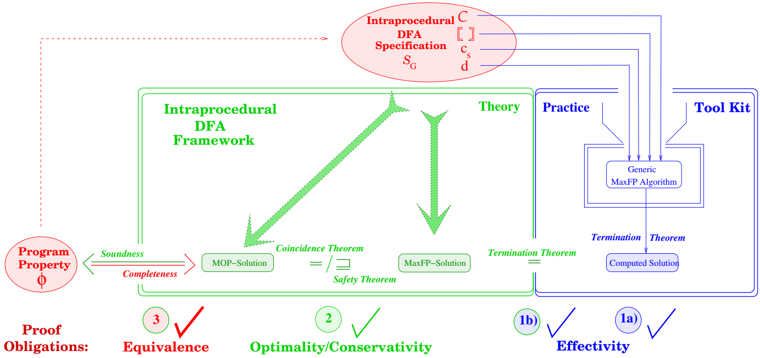
Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, t an expression, $vb_{\mathbf{e}} \in \mathbb{B}$ the very busyness information for t at $\mathbf{e}$ ensured by the calling context of G, and let $MOP_{\mathcal{S}_G^{vb,t}}$ be the *MOP* solution of the DFA specification $\mathcal{S}_G^{vb,t} = (\widehat{\mathbb{B}}, [\![\ ]\!]_{vb}^{t}, vb_{\mathbf{e}}, bw)$.

Then:

$$\forall\, n \in N.\ VeryBusy^{t}(n) \iff MOP_{\mathcal{S}_G^{vb,t}}(n)$$

Gap Closed: Soundness&Completeness Proven

...soundness and completeness of $\mathcal{S}_G^{vb,t}$ for very busyness proven:



Chapter 4.3

Summary, Looking Ahead

Summary, Looking Ahead (1)

The terms

- Gen/Kill Analyses
- Bitvector Analyses

are used synonymously.

Gen/Kill Analyses are

- efficient
- scale well to more complex DFA scenarios (interprocedural, parallel, etc.)
- are most important in practice.

Summary, Looking Ahead (2)

In fact, despite their conceptual simplicity, information obtained by Gen/Kill Analyses is fundamental for

- numerous powerful and widely used optimizations, e.g.,
  - Partial Redundancy Elimination (Busy Code Motion, Lazy Code Motion, cf. Chapter 7 and Chapter 8)
  - Strength Reduction (Lazy Strength Reduction, cf. Chapter 11)
  - Partial Dead-Code Elimination (cf. LVA 185.276 Analyse und Verifikation)
  - Assignment Motion (cf. LVA 185.276 Analyse und Verifikation)
  - ...

Chapter 4.4

References, Further Reading

Further Reading for Chapter 4 (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 1, Introduction; Chapter 9.2, Introduction to Data-Flow Analysis; Chapter 9.3, Foundations of Data-Flow Analysis)
- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 4.4, Data Flow Analysis)
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 10.2, A Taxonomy for Transformations — Machine-Independent Transformations, Machine-Dependent Transformations)

Further Reading for Chapter 4 (2)

- Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 2, Classical Bit Vector Data Flow Analysis)
- Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 4, Flow Graph)
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 8.3, Taxonomy of Data-Flow Problems and Solution Methods)

Further Reading for Chapter 4 (3)

- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)
- Barry K. Rosen. *High-level Data Flow Analysis*. Communications of the ACM 20(10):141-156, 1977.

Chapter 5

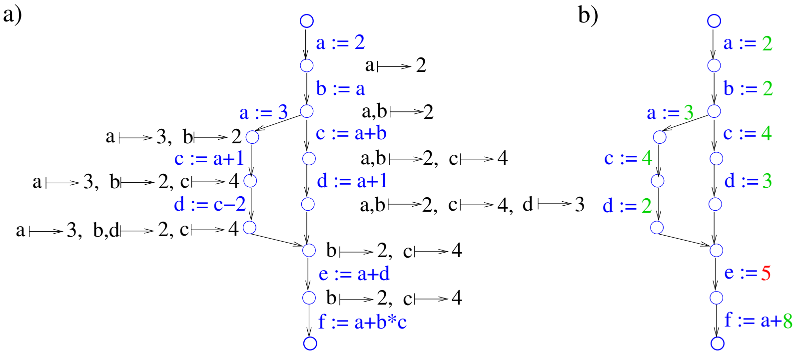
Constant Propagation

Chapter 5.1

Motivation

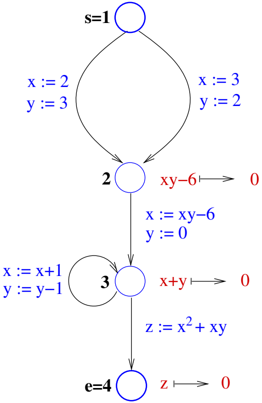
Motivating Example

...constant propation (CP) aims at discovering and replacing occurrences of terms whose computation will always yield the same value at runtime by this value.



Illustrating the Challenges of CP

...show that the terms xy-6, x+y, and z are constants of value 0 at the nodes 2, 3, and 4, respectively, .



Markus Müller-Olm, Helmut Seidl (SAS 2002)

Undecidability of Constant Propagation

Theorem 5.1.1 (Undecidability, Reif&Lewis 1977)

In the arithmetic domain, the problem of discovering all text expressions covered by constant signs is undecidable.

(John H. Reif, Harry R. Lewis. Symbolic Evaluation and the Global Value Graph. In Conference Record of the 4th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), 104-118, 1977)

Proof Sketch of Theorem 5.1.1 (1)

The proof of Theorem 5.1.1 proceeds by reducing Hilbert's 10th problem to the problem of discovering all text expressions covered by constant signs:

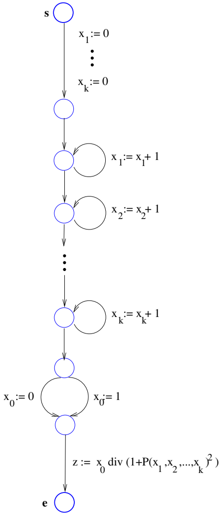
- Hilbert's 10th Problem

  Let $\{x_1, \ldots, x_k\}$ be a set of variables, $k > 5$, and let $P(x_1, \ldots, x_k)$ be a (multivariate) polynomial.

  It is not decidable, if determining if $P(x_1, \ldots, x_k)$ has a root in the natural numbers (Matijasevic 1970).

Proof Sketch of Theorem 5.1.1 (2)

Consider the program G below:



Proof Sketch of Theorem 5.1.1 (3)

Proving the equivalence

P has no root in the natural numbers iff
z is constant (at node $\mathbf{e}$ of G)

completes the proof. $\square$

The Bad News of Theorem 5.1.1

There is no hope of developing an algorithm that, when applied to an arbitrary program G,

- determines for every term occurrence in G in a finite number of steps, if the evaluation of this occurrence will always yield the same value when its site is reached and its value computed at the runtime of G.

The Good News of Theorem 5.1.1 (1)

The impossibility of solving the constant propagation (CP) problem once and for all

- gives room for both theoreticians and practitioners to strive for constant propagation algorithms tailored and optimized for different purposes and goals.

The Good News of Theorem 5.1.1 (2)

The undecidability of the general CP problem inspires...

For the theoretician

- ...a quest for discovering settings with a decidable CP problem
  - Restricting the class of admissible programs
    - Finite constants: arbitrary term operators, decidable for arbitrary control flow, complete for acyclic control flow, EXPTIME algorithm (Steffen&Knoop 1989)
  - Restricting the set of admissible expression operators
    - Presburger constants: $+$, $-$ as term operators, decidable and complete for programs with arbitrary control flow, polynomial time algorithm (Müller-Olm&Rüthing 2001)
    - Polynomial constants: $+$, $-$, $*$ as term operators, decidable and complete for programs with arbitrary control flow, time complexity of the proposed decision algorithm not yet known, PSPACE-hardness as a lower complexity bound (Müller-Olm&Seidl 2002)

The Good News of Theorem 5.1.1 (3)

For the practitioner

- ...a quest for discovering sweet spot settings with a useful and efficiently, scalable decidable CP problem
  - Simple constants, intraprocedural
    - based on expression pools (Kildall 1973)
    - based on definition-use chains
    - based on abstract state transformers
    - based on the global value graph (Reif&Lewis 1977)
    - based on the SSA value graph (Knoop&Rüthing 2000)
  - Q constants, intraprocedural (Kam&Ullman 1977)
  - Conditional constants, intraprocedural (Wegman&Zadeck 1985)
  - Linear constants, interprocedural (Sagiv, Reps, Horwitz 1996)
  - Copy constants, interprocedural
  - Strong constants, parallel (Knoop 1998)
  - ...

The Good News of Theorem 5.1.1 (4)

In essence

Theoreticians cope with the undecidability of CP by

- trading generality as little as possible for decidability (neglecting efficiency and scalability).

Practitioners cope with the undecidability of CP by

- trading generality as much as necessary for efficient, scalable and useful decidability.

Note

- This is quite a typical situation in program analysis and optimization, and a virtually unexhaustable source of challenging and important research questions.

In the following

...we will focus on some of the approaches for constant propagation inspired by practical demands and needs, i.e., approaches offering a good cost/benefit ratio:

- Simple constants
- Linear constants
- Copy constants
- Q constants
- Conditional constants

Additionally, we consider

- Finite constants

as an example of an optimal class of constants.

Chapter 5.2

Preliminaries, Problem Definition

Workplan

Introducing and defining

- Syntax of terms (variables, operator and constant symbols,...)
- Semantics of terms (data domain, interpretation of operator and constant symbols, states,...)
- Semantics of instructions (state transformers,...)
- Semantics of programs (collecting semantics)
- Constant propagation problem

formally.

Towards the Syntax of Terms

...variables, constants, operators.

Let

- **V** be a set of variables,
- **C** be a set of constant symbols (constants),
- **O** be a set of k-ary operator symbols (or operators), $k \geq 1$.

Let

- **V**, **C**, and **O** be disjoint.

Syntax of Terms

Definition 5.2.1 (Terms)

1. Every variable $v \in \mathbf{V}$, every constant $c \in \mathbf{C}$ is a term.
2. If $op \in \mathbf{O}$ is a k-ary operator and $t_1, \ldots, t_k$ are terms, then $(op, t_1, \ldots, t_k)$ is a term.
3. There are no terms other than those which can be constructed by means of the above two rules.

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

Towards the Semantics of Terms

We require

- a data domain ID,
- an interpretation of constant and operator symbols over ID,
- a set of states over ID.

Data Domain

Let

- $\mathbb{D}$ be a data domain of interest.

  (E.g., the set of natural numbers $\mathbb{N}$, the set of integers $\mathbb{Z}$, the set of Boolean truth values $\mathbb{B}$, etc.).

The

- elements of $\mathbb{D}$ are called (data) values.

We assume

- $\mathbb{D}$ includes a distinguished element $\perp$ representing the value *undefined*.

Interpreting Constant and Operator Symbols

Definition 5.2.2 (Interpretation)

An interpretation $I =_{df} (\mathbb{D}, I_0)$ of $\mathbf{C}$ and $\mathbf{O}$ is a pair, where

- $\mathbb{D}$ is a data domain,
- I_0 is a function, which maps every
  - constant symbol $c \in \mathbf{C}$ to a datum $I_0(c) \in \mathbb{D}$,
  - k-ary operator symbol $op \in \mathbf{O}$ to a total strict function $I_0(op) : \mathbb{D}^k \to \mathbb{D}$, i.e., $I_0(op)(d_1, \ldots, d_k) = \perp$, if there is a $j \in \{1, \ldots, k\}$ with $d_j = \perp$.

States

Definition 5.2.3 (States over ID)

A state $\sigma : \mathbf{V} \to \mathrm{ID}$ is a total mapping, which maps every variable to a data value $d \in \mathrm{ID}$.

We denote the set of all states by

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

Semantics of Terms

Definition 5.2.4 (Semantics of Terms)

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

$$\mathcal{E} : \mathbf{T} \rightarrow (\Sigma \rightarrow \mathrm{I\!D})$$

defined by

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

Semantics of Instructions

Definition 5.2.5 (Semantics of Instructions)

- Let $\iota \equiv x := t$ be an assignment instruction. The semantics of ι is defined by the state transformation function (or state transformer) $\theta_\iota : \Sigma \to \Sigma$ defined by

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

- Let $\iota \equiv \mathit{skip}$ be the empty instruction. The semantics of ι is defined by the identical state transformation function (or state transformer) Id_Σ , i.e., $\theta_\iota =_{df} Id_\Sigma$, where $Id_\Sigma : \Sigma \to \Sigma$ is defined by $\forall\, \sigma \in \Sigma.\ Id_\Sigma(\sigma) =_{df} \sigma$.

Extending State Transformers to Paths

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let ι_e denote the instruction at edge e, $e \in E$.

Definition 5.2.6 (Extending θ from Edges to Paths)

The state transformers θ_{ι_e}, $e \in E$, are extended onto paths $p = \langle e_1, e_2, \ldots, e_q \rangle$ in G by defining:

$$\theta_p =_{df} \begin{cases} Id_\Sigma & \text{if } q < 1 \\ \theta_{\langle e_2, \ldots, e_q \rangle} \circ \theta_{\iota_{e_1}} & \text{otherwise} \end{cases}$$

Semantics of Programs: Collecting Semantics

Definition 5.2.7 (Collecting Semantics)

- The collecting semantics of G is defined by:

$$\mathcal{CS}_G : \Sigma \to N \to \mathcal{P}(\Sigma)$$

$$\forall\, n \in N.\ \forall\, \sigma \in \Sigma.\ \mathcal{CS}_G(n) =_{df} \{\, \theta_p(\sigma) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\}$$

- The collecting semantics of G with respect to a fixed initial state $\sigma_{\mathbf{s}} \in \Sigma$ is defined by

$$\mathcal{CS}_G^{\sigma_{\mathbf{s}}} : N \to \mathcal{P}(\Sigma)$$

$$\forall\, n \in N.\ \mathcal{CS}_G^{\sigma_{\mathbf{s}}}(n) =_{df} \{\, \theta_p(\sigma_{\mathbf{s}}) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\}$$

Non-deterministic Constants

Let $\sigma_{\mathbf{s}} \in \Sigma$ be an (initial) state, let $t \in \mathbf{T}$ be a term, and let $d \in \mathrm{ID} \backslash \{\perp\}$ be a data value.

Definition 5.2.8 ((Non-deterministic) Constant)

t is a constant at node n for $\sigma_{\mathbf{s}}$, i.e., the value of t at node n, $n \in N$, is a constant, if

$$\forall \sigma, \sigma' \in \mathcal{CS}_G^{\sigma_{\mathbf{s}}}(n).\ \mathcal{E}(t)(\sigma) = \mathcal{E}(t)(\sigma') \neq \perp$$

Definition 5.2.9 ((Non-det.) Constant of Value d)

t is a constant of value d for $\sigma_{\mathbf{s}}$ at node n, $n \in N$, if

$$\{\mathcal{E}(t)(\sigma) \mid \sigma \in \mathcal{CS}_G^{\sigma_{\mathbf{s}}}(n)\} = \{d\}$$

Constant Terms and Variables

Let $\sigma_{\mathbf{s}} \in \Sigma$ be a state, and let n be a node of G.

Definition 5.2.10 (Constant Terms and Variables)

The set of terms and variables being constants of some value at n are given by the sets:

- $CT_G^{\sigma_{\mathbf{s}}}(n) =_{df}$
  $\{(t, d) \in \mathbf{T} \times \mathrm{ID} \mid t \text{ is a constant of value } d \text{ for } \sigma_{\mathbf{s}} \text{ at } n\}$
- $CV_G^{\sigma_{\mathbf{s}}}(n) =_{df}$
  $\{(v, d) \in \mathbf{V} \times \mathrm{ID} \mid v \text{ is a constant of value } d \text{ for } \sigma_{\mathbf{s}} \text{ at } n\}$

The Non-Det. Constant Propagation Problem

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let $\sigma_{\mathbf{s}} \in \Sigma$ be a state.

The non-deterministic constant propagation problems for terms and variables are defined by:

Definition 5.2.11 (CP Problem for Terms (CT))

The (non-deterministic) term constant propagation problem, CT, is to determine for every node $n \in N$ of G the set $CT_G^{\sigma_{\mathbf{s}}}(n)$.

Definition 5.2.12 (CP Problem for Variables (CV))

The (non-deterministic) variable constant propagation problem, CV, is to determine for every node $n \in N$ of G the set $CV_G^{\sigma_{\mathbf{s}}}(n)$.

States Induced by $CT_G^{\sigma_s}$ and $CV_G^{\sigma_s}$

The sets $CT_G^{\sigma_s}$ and $CV_G^{\sigma_s}$ induce for every node n two states $\sigma^{\mathbf{T}}_{CT_G^{\sigma_s}}(n)$ and $\sigma^{\mathbf{V}}_{CV_G^{\sigma_s}}(n)$, respectively, which we use alternatively to the sets $CT_G^{\sigma_s}(n)$ and $CV_G^{\sigma_s}(n)$ (cf. Lemma 5.2.13):

- $\sigma^{\mathbf{T}}_{CT_G^{\sigma_s}} : N \to \mathbf{T} \to \mathrm{ID}$ defined by

$$\sigma^{\mathbf{T}}_{CT_G^{\sigma_s}}(n)(t) =_{df} \begin{cases} d & \text{if } (t,d) \in CT_G^{\sigma_s}(n) \\ \perp & \text{otherwise} \end{cases}$$

- $\sigma^{\mathbf{V}}_{CV_G^{\sigma_s}} : N \to \mathbf{V} \to \mathrm{ID}$ defined by

$$\sigma^{\mathbf{V}}_{CV_G^{\sigma_s}}(n)(v) =_{df} \begin{cases} d & \text{if } (v,d) \in CV_G^{\sigma_s}(n) \\ \perp & \text{otherwise} \end{cases}$$

Equivalence of Set&State-like Characterization

We have:

Lemma 5.2.13 (Equivalence)

$\forall\, n \in N\ \forall\, t \in \mathbf{T}\ \forall\, v \in \mathbf{V}\ \forall\, d \in \mathrm{ID}\backslash\{\perp\}.$

- $(t,d) \in CT_G^{\sigma_\mathbf{s}}(n)$ iff $\sigma^{\mathbf{T}}_{CT_G^{\sigma_\mathbf{s}}}(n)(t) = d$
- $(v,d) \in CV_G^{\sigma_\mathbf{s}}(n)$ iff $\sigma^{\mathbf{V}}_{CV_G^{\sigma_\mathbf{s}}}(n)(v) = d$

Equivalent Characterization of the CP Problem

The Equivalence Lemma 5.2.13 yields:

Lemma 5.2.14 (Problem Equivalence)

Solving the non-deterministic

- term constant propagation problem CT
- variable constant propagation problem CV

is equivalent to computing the functionals

- $\sigma^{\mathbf{T}}_{CT^{\sigma_s}_G} : N \to \mathbf{T} \to \mathbb{ID}$
- $\sigma^{\mathbf{V}}_{CV^{\sigma_s}_G} : N \to \mathbf{V} \to \mathbb{ID}$

respectively.

CP Algorithms: Soundness and Completeness

Let A be a constant propagation algorithm for CT (CV); let $A_{CT_G^{\sigma_s}}(n) \subseteq \mathbf{T} \times \mathrm{ID}$ ($A_{CV_G^{\sigma_s}}(n) \subseteq \mathbf{V} \times \mathrm{ID}$) denote the sets of terms (variables) discovered by A to be constant at node n.

Definition 5.2.15 (Soundness of CP Algorithms)

A is sound for CT (CV) if

$$\forall\, n \in N.\, \forall\, \sigma_s \in \Sigma.\; CT_G^{\sigma_s}(n) \supseteq A_{CT_G^{\sigma_s}}(n) \;\; (\, CV_G^{\sigma_s}(n) \supseteq A_{CV_G^{\sigma_s}}(n)\,)$$

Definition 5.2.16 (Completeness of CP Algorithms)

A is complete for CT (CV) if

$$\forall\, n \in N.\, \forall\, \sigma_s \in \Sigma.\; CT_G^{\sigma_s}(n) \subseteq A_{CT_G^{\sigma_s}}(n) \;\; (\, CV_G^{\sigma_s}(n) \subseteq A_{CV_G^{\sigma_s}}(n)\,)$$

CP Algorithms: Conservativity and Optimality

Definition 5.2.17 (Conservativity of CP Algorithms)

A CP algorithm A is conservative for CT (CV), if it is sound for CT (CV), i.e.

$$\forall\, n \in N.\, \forall\, \sigma_{\mathbf{s}} \in \Sigma.\ CT_G^{\sigma_{\mathbf{s}}}(n) \supseteq A_{CT_G^{\sigma_{\mathbf{s}}}}(n) \ \ (\, CV_G^{\sigma_{\mathbf{s}}}(n) \supseteq A_{CV_G^{\sigma_{\mathbf{s}}}}(n)\,)$$

Definition 5.2.18 (Optimality of CP Algorithms)

A CP algorithm A is optimal for CT (CV), if it is sound and complete for CT (CV), i.e.

$$\forall\, n \in N.\, \forall\, \sigma_{\mathbf{s}} \in \Sigma.\ CT_G^{\sigma_{\mathbf{s}}}(n) = A_{CT_G^{\sigma_{\mathbf{s}}}}(n) \ \ (\, CV_G^{\sigma_{\mathbf{s}}}(n) = A_{CV_G^{\sigma_{\mathbf{s}}}}(n)\,)$$

Relating $CV_G^{\sigma_{\mathbf{s}}}$ and $CT_G^{\sigma_{\mathbf{s}}}$ (1)

The functional

$$\sigma^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}} : \mathbf{V} \to \mathrm{ID}$$

induced by the variable CP problem $CV_G^{\sigma_{\mathbf{s}}}$ induces for every node a solution of the term constant propagation problem in terms of states and sets, respectively:

- State-based:

$$\sigma^{\mathbf{T}}_{CV_G^{\sigma_{\mathbf{s}}}} : N \to \mathbf{T} \to \mathrm{ID}$$

$$\sigma^{\mathbf{T}}_{CV_G^{\sigma_{\mathbf{s}}}}(n)(t) =_{df} \mathcal{E}(t)(\sigma^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n))$$

- Set-based:

$$CT_{CV_G^{\sigma_{\mathbf{s}}}}(n) =_{df} \{(t,d) \in \mathbf{T} \times \mathrm{ID} \mid \mathcal{E}(t)(\sigma^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n)) = d \neq \perp\}$$

Relating $CV_G^{\sigma_\mathbf{s}}$ and $CT_G^{\sigma_\mathbf{s}}$ (2)

We have:

Lemma 5.2.13 (Equivalence Lemma)

$$\forall\, n \in N.\ \forall\, \sigma_\mathbf{s} \in \Sigma.\ CT_{CV_G^{\sigma_\mathbf{s}}}(n) = \sigma^{\mathbf{T}}_{CV_G^{\sigma_\mathbf{s}}}(n)$$

Lemma 5.2.14 (Approximation Lemma)

$$\forall\, n \in N.\ \forall\, \sigma_\mathbf{s} \in \Sigma.\ CT_G^{\sigma_\mathbf{s}}(n) \supseteq CT_{CV_G^{\sigma_\mathbf{s}}}(n)$$

In general, this inclusion is a proper inclusion.

Interpretation, Conclusions (1)

Intuitively

- The Approximation Lemma 5.2.13 states that a term can be a constant at some node n without that all of its variables are constants at n.

  E.g., the equality of $xy - yx$ and 0 can be concluded without knowing the values of x and y; actually, they need not be constant at all. For a more complex case consider $x^2 + xy = 0$ in the example of Müller-Olm and Seidl in Chapter 5.1.

Hence

- Any sound algorithm for the CV constant propagation problem is in general conservative and suboptimal for the CT constant propagation problem.
- This holds even for a (hypothetical) optimal algorithm (cf. Th. 5.1.1) for the CV constant propagation problem.

Interpretation, Conclusions (2)

As a matter of fact

- The Undecidability Theorem 5.1.1 rules out the possibility and existence of CT and CV optimal constant propagation algorithms.

Hence

- The best we can hope for are conservative CT and CV constant propagation algorithms trading optimality for decidability (and efficiency, scalability).

Characterizing CP Algorithms (1)

In practice, CP algorithms fall into two groups, algorithms $\mathcal{A}$, which compute and store values for

- variables at a program node, hence computing a mapping

$$\mathcal{A}_{CV} : N \to \mathbf{V} \to \mathcal{P}(\mathrm{ID})$$

- terms at a program node, hence computing a mapping

$$\mathcal{A}_{CT} : N \to \mathbf{T} \to \mathcal{P}(\mathrm{ID})$$

as the result of the analysis, called variable and term valuation function, respectively.

Characterizing CP Algorithms (2)

We call algorithms falling into these two groups

- CV algorithms
- CT algorithms

for constant propagation, respectively.

Moreover, we call CV and CT algorithms

- singleton CV algorithms, if they store at most one value per variable a program node, i.e., if they compute and store a mapping $\mathcal{A}_{CV} : N \to \mathbf{V} \to \mathbb{D}$
- singleton CT algorithms, if they store at most one value per term at a program node, i.e., if they compute and store a mapping $\mathcal{A}_{CT} : N \to \mathbf{T} \to \mathbb{D}$

respectively.

Characterizing CP Algorithms (3)

The algorithms for

- Simple constants
- Linear constants
- Copy constants
- Q constants

are singleton CV algorithms.

The algorithm for

- Finite constants

is a singleton CT algorithm.

Note: The algorithm for conditional constants is a singleton algorithm, too, but addresses the deterministic CV constant propagation problem.

Induced Term Valuation Function

The variable valuation function

- $\mathcal{A}_{CV} : N \to \mathbf{V} \to \mathrm{ID}$

of a CV algorithm $\mathcal{A}$ induces a term valuation function

- $\mathcal{A}_{CV}^{\mathbf{T}} : N \to \mathbf{T} \to \mathrm{ID}$

defined by

$$\forall\, n \in N \;\forall\, t \in \mathbf{T}.\; \mathcal{A}_{CV}^{\mathbf{T}}(n)(t) =_{df}$$

$$\begin{cases} d & \text{if } \mathcal{E}(t)(\mathcal{A}_{CV}(n)) = d \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

CV/CT Solutions induced by Valuations (1)

Let $\mathcal{A}$ be a singleton CV constant propagation algorithm.

The variable and term valuation functions

- $\mathcal{A}_{CV} : N \rightarrow \mathbf{V} \rightarrow \mathrm{ID}$
- $\mathcal{A}^{\mathbf{T}}_{CV} : N \rightarrow \mathbf{T} \rightarrow \mathrm{ID}$

of $\mathcal{A}$ induce solutions for the CV and the CT constant propagation problems:

- $CV_{\mathcal{A}_{CV}}(n) =_{df} \{(v, d) \in \mathbf{V} \times \mathrm{ID} \mid \mathcal{A}_{CV}(n)(v) = d \neq \perp\}$
- $CT_{\mathcal{A}_{CV}}(n) =_{df} \{(t, d) \in \mathbf{T} \times \mathrm{ID} \mid \mathcal{A}^{\mathbf{T}}_{CV}(n)(t) = d \neq \perp\}$

CV/CT Solutions induced by Valuations (2)

Let $\mathcal{A}$ be a singleton CT constant propagation algorithm.

The term valuation function

- $\mathcal{A}_{CT} : N \to \mathbf{T} \to \mathrm{ID}$

of $\mathcal{A}$ induces solutions for the CV and the CT constant propagation problems:

- $CV_{\mathcal{A}_{CT}}(n) =_{df} \{(v, d) \in \mathbf{V} \times \mathrm{ID} \mid \mathcal{A}_{CT}(n)(v) = d \neq \perp\}$
- $CT_{\mathcal{A}_{CT}}(n) =_{df} \{(t, d) \in \mathbf{T} \times \mathrm{ID} \mid \mathcal{A}_{CT}(n)(t) = d \neq \perp\}$

Chapter 5.3

Simple Constants

Chapter 5.3.1

DFA States, DFA Lattice

From Data Domains to DFA Lattices (1)

...domain extension.

Let

- $\mathbb{D}$ be the data domain of interest (e.g. the set of natural numbers $\mathbb{N}$, the set of integers $\mathbb{Z}$, the set of Boolean truth values $\mathbb{B}$, etc.) with a distinguished element $\bot$ representing the value *undefined*.

We extend $\mathbb{D}$ by adding

- a new element $\top$ not in $\mathbb{D}$, i.e., $\top \notin \mathbb{D}$.

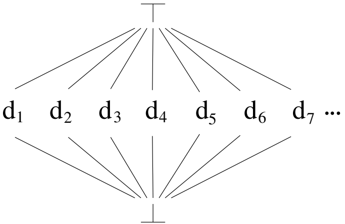
We denote the extended domain by

- $\mathbb{D}' =_{df} \mathbb{D} \cup \{\top\}$.

From Data Domains to DFA Lattices (2)

...lattice construction.

Given an extended data domain $\mathbb{D}'$, we construct the flat lattice $\mathcal{FL}_{\mathbb{D}'}$ (cf. Appendix A.4)



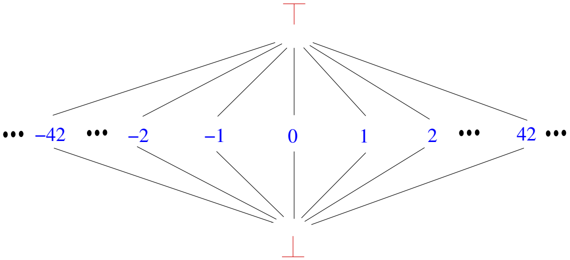
which constitutes the basic DFA lattice of the CP analysis.

Intuitively

- $\top$ represents complete but inconsistent information.
- d_i, $i \geq 1$, represents precise information.
- $\perp$ represents no information, the empty information.

The Basic DFA Lattice over $\mathbb{Z}$

...is given by $\mathcal{FL}_{\mathbb{Z}}$



...leading to the class of simple constants over $\mathbb{Z}$.

Adapting the Notion of States: DFA States

Definition 5.3.1.1 (DFA States)

A DFA state $\sigma : \mathbf{V} \to \mathbb{ID}'$ is a total mapping, which maps every variable to a datum $d \in \mathbb{ID}'$.

The set of all DFA states is denoted by

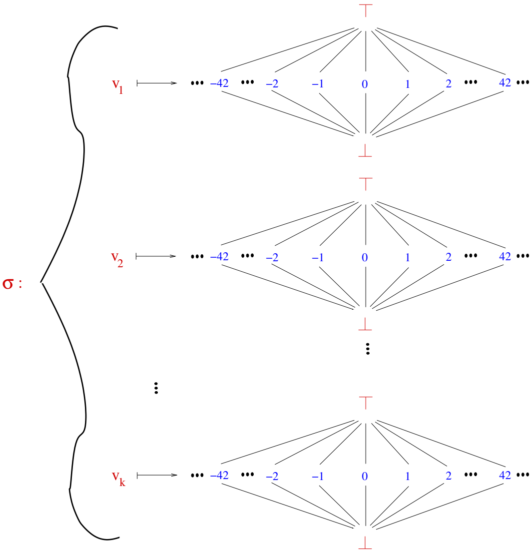
$$\Sigma' =_{df} \{ \sigma \mid \sigma : \mathbf{V} \to \mathbb{ID}' \}$$

$\sigma_\perp$ and $\sigma_\top$ denote two distinguished states of Σ' defined by

$$\forall\, v \in \mathbf{V}.\ \sigma_\perp(v) = \perp,\ \sigma_\top(v) = \top$$

respectively.

Illustrating a DFA State σ over $\mathbb{Z}$



Initial DFA States

For an initial DFA state, we require that no variable is mapped to the special value $\top$, i.e, we require to either have precise information of the value of a variable, when entering a procedure, or no information at all. We define:

Definition 5.3.1.2 (Initial DFA States over ID′)

The set of initial DFA states is defined by

$$\Sigma'_{Init} =_{df} \{ \sigma \in \Sigma' \mid \forall v \in \mathbf{V}.\ \sigma(v) \neq \top \}$$

Note: The set of initial DFA states Σ'_{Init} coincides with the set of (program) states Σ of Definition 5.2.3, i.e., $\Sigma'_{Init} = \Sigma$.

Extending the Interpretation to $\top \in \mathrm{ID}'$

Definition 5.3.1.3 (Extending the Interpretation)

Let $I =_{df} (\ \mathrm{ID}, I_0\ )$ be an interpretation of constant and operator symbols over the data domain ID.

Then $I' =_{df} (\ \mathrm{ID}', I'_0\ )$ extends I to an interpretation over ID' by defining

- $I'_0(c) =_{df} I_0(c)$ for every constant symbol $c \in \mathbf{C}$
- $I'_0(op) : \mathrm{ID}'^n \to \mathrm{ID}'$ for every operator symbol $op \in \mathbf{O}$ by

$$\forall (d_1, \ldots, d_k) \in \mathrm{ID}'^n.\ I'_0(op)(d_1, \ldots, d_k) =_{df}$$

$$\begin{cases} I_0(op)(d_1, \ldots, d_k) & \text{if } d_i = \bot \text{ for some } 1 \leq i \leq k \text{, or} \\ & \quad d_j \neq \top,\ 1 \leq j \leq k \\ \top & \text{if } d_i \neq \bot,\ 1 \leq i \leq k \text{, and} \\ & \quad d_j = \top \text{ for some } 1 \leq j \leq k \end{cases}$$

Adapting the Semantics of Terms to ID'

Definition 5.3.1.4 (Adapted Semantics of Terms)

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

$$\mathcal{E}' : \mathbf{T} \to (\Sigma' \to \mathsf{ID}')$$

defined by

$$\forall t \in \mathbf{T}\ \forall \sigma \in \Sigma'.\ \mathcal{E}'(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ I_0'(c) & \text{if } t \equiv c \in \mathbf{C} \\ I_0'(op)(\mathcal{E}'(t_1)(\sigma), \ldots, \mathcal{E}'(t_k)(\sigma)) & \\ \qquad\qquad \text{if } t \equiv (op, t_1, \ldots, t_k) & \end{cases}$$

Adapting the Semantics of Instructions to Σ'

Definition 5.3.1.5 (Adapted Semantics of Instruc's)

- Let $\iota \equiv x := t$ be an assignment instruction. The semantics of ι is defined by the extended state transformer $\theta'_\iota : \Sigma' \to \Sigma'$ defined by

$$\forall \sigma \in \Sigma' \; \forall y \in \mathbf{V}. \; \theta'_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}'(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- Let $\iota \equiv skip$ be the empty instruction. The semantics of ι is defined by the extended identical state transformer $Id_{\Sigma'}$, i.e., $\theta'_\iota =_{df} Id_{\Sigma'}$, where $Id_{\Sigma'} : \Sigma' \to \Sigma'$ is defined by $\forall \sigma \in \Sigma'. \; Id_{\Sigma'}(\sigma) =_{df} \sigma$.

The DFA Lattice for Simple Constants

The set of DFA states together with the pointwise ordering of states, $\sqsubseteq_{\Sigma'}$, constitutes a complete lattice (cf. Appendix A.4):

$$\forall\, \sigma, \sigma' \in \Sigma'.\ \sigma \sqsubseteq_{\Sigma'} \sigma' \ \text{ iff } \ \forall\, v \in \mathbf{V}.\ \sigma(v) \sqsubseteq_{\mathcal{FL}_{\mathbb{D}'}} \sigma'(v)$$

Lemma 5.3.1.6 (Lattice of DFA States)

$\widehat{\Sigma}' =_{df} (\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_\perp, \sigma_\top)$ is a complete lattice with

- least element $\sigma_\perp$, greatest element $\sigma_\top$,
- pointwise meet $\sqcap_{\Sigma'}$ and join $\sqcup_{\Sigma'}$ as meet and join operation, respectively.

Chapter 5.3.2

Simple Constants: Specification

Simple Constants over $\mathbb{Z}$: DFA Specification

DFA Specification

- DFA lattice
  $$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_\bot, \sigma_\top) = \widehat{\Sigma'}$$
  with Σ' set of DFA states over $\mathbb{Z}$.
- DFA functional
  $[\![\ ]\!]_{sc} : E \to (\Sigma' \to \Sigma')$ where $\forall\, e \in E.\ [\![e]\!]_{sc} =_{df} \theta'_{\iota_e}$
- Initial information: $\sigma_{\mathbf{s}} \in \Sigma'_{Init}$
- Direction of information flow: forward

Simple Constants Specification

- Specification: $\mathcal{S}_G^{sc} = (\widehat{\Sigma'}, [\![\ ]\!]_{sc}, \sigma_{\mathbf{s}}, fw)$

Chapter 5.3.3

Termination, Safety, and Coincidence

Towards Safety and Termination

Lemma 5.3.3.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.3.3.2 (Monotonicity)

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

Lemma 5.3.3.3 (Non-Distributivity)

$[\![\ ]\!]_{sc}$ is not distributive.

Termination and Safety/Conservativity

Theorem 5.3.3.4 (Termination)

Applied to $\mathcal{S}_G^{sc} = (\widehat{\Sigma}', [\![\ ]\!]_{sc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{sc}$.

Proof. Immediately with Lemma 5.3.3.1, Lemma 5.3.3.2, and Termination Theorem 3.4.4.

Theorem 5.3.3.5 (Safety/Conservativity)

Applied to $\mathcal{S}_G^{sc} = (\widehat{\Sigma}', [\![\ ]\!]_{sc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 is *MOP* conservative for $\mathcal{S}_G^{sc}$ (i.e., it terminates with a lower approximation of the *MOP* solution of $\mathcal{S}_G^{sc}$).

Proof. Immediately with Lemma 5.3.3.2, Safety Theorem 3.5.1, and Termination Theorem 5.3.3.4.

Non-Coincidence

Theorem 5.3.3.6 (Non-Coincidence/Non-Opt.)

Applied to $\mathcal{S}_G^{sc} = (\widehat{\Sigma}', [\![\ ]\!]_{sc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 is in general not *MOP* optimal for $\mathcal{S}_G^{sc}$ (i.e., it terminates with a properly lower approximation of the *MOP* solution of $\mathcal{S}_G^{sc}$).

Proof. Immediately with Lemma 5.3.3.3, Coincidence Theorem 3.5.2, and Termination Theorem 5.3.3.4.

Corollary 5.3.3.7 (Safety, Non-Coincidence)

The *MaxFP* solution for $\mathcal{S}_G^{sc}$, is always a safe approximation of the *MOP* solution of $\mathcal{S}_G^{sc}$. In general, the *MOP* solution and the *MaxFP* solution of $\mathcal{S}_G^{sc}$ do not coincide.

Chapter 5.3.4

Soundness and Completeness

Soundness and Completeness of $MOP_{\mathcal{S}_G^{sc}}$

Theorem 5.3.4.1 (Soundness and Completeness)

The *MOP* solution of $\mathcal{S}_G^{sc}$ is

1. sound and complete for the variable constant propagation problem CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) = MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{sc}}(n)$$

2. sound but not complete for the term constant propagation problem CT, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MOP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{sc}}(n)$$

In general, the inclusion is a proper inclusion.

Soundness and Completeness of $MaxFP_{\mathcal{S}_G^{sc}}$

Corollary 5.3.4.2 (Soundness and Completeness)

The *MaxFP* solution of $\mathcal{S}_G^{sc}$ is

1. sound but not complete for CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MaxFP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{sc}}(n)$$

2. sound but not complete for CT, i.e.

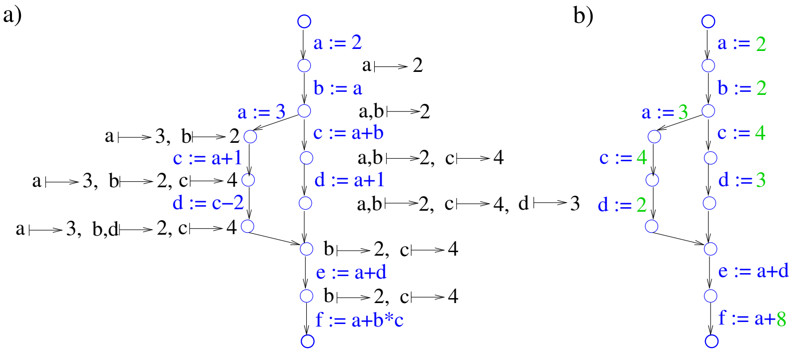
$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MaxFP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{sc}}(n)$$

In general, both inclusions are proper inclusions.

Chapter 5.3.5

Illustrating Example

Simple Constants over ℤ: Illustrating Example



...all terms except of $a + d$ and $a + 8$ are simple constants.

Note: The term $a + d$ is a constant of value 5, though not a simple constant; the term $a + 8$ is not a (non-deterministic) constant.

Chapter 5.4

Linear Constants

Chapter 5.4.1

DFA States, DFA Lattice

DFA States, DFA Lattice, and More

The linear constants (LC) analysis shares the

- extended data domain ID'
- basic flat DFA lattice $\mathcal{FL}_{\mathsf{ID}'}$
- set of DFA states Σ'
- extended interpretation $I' =_{df} (\mathsf{ID}', I'_0)$

with the simple constants (SC) analysis.

LC-specific are

- the adaption of the semantics of terms to ID'
- the adaption of the semantics of instructions to Σ'

LC-specific Term Semantics Adaption

Definition 5.4.1.1 (Adapted Semantics of Terms)

The LC-specific semantics of terms $t \in \mathbf{T}$ is defined by the extended evaluation function

$$\mathcal{E}_{lc} \colon \mathbf{T} \to (\Sigma' \to \mathrm{ID}')$$

defined by

$$\forall t \in \mathbf{T}\ \forall \sigma \in \Sigma'.\ \mathcal{E}_{lc}(t)(\sigma) =_{df}$$

$$\begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ I'_0(c) & \text{if } t \equiv c \in \mathbf{C} \\ I'_0(\oplus)(\mathcal{E}_{lc}(*, c, x)(\sigma), \mathcal{E}_{lc}(d)(\sigma)) & \\ & \text{if } t \equiv (\oplus, (*, c, x), d) \equiv c * x \oplus d, \\ & \qquad c, d \in \mathbf{C},\ \oplus \in \{+, -\} \\ \perp & \text{otherwise} \end{cases}$$

LC-specific Instructions Semantics Adaption

Definition 5.4.1.2 (Adapted Semantics of Instruc's)

- Let $\iota \equiv x := t$ be an assignment instruction. The semantics of ι is defined by the LC-specific state transformer $\theta_{\iota}^{lc} : \Sigma' \to \Sigma'$ defined by

$$\forall\, \sigma \in \Sigma'\ \forall\, y \in \mathbf{V}.\ \theta_{\iota}^{lc}(\sigma)(y) =_{df} \begin{cases} \mathcal{E}_{lc}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- Let $\iota \equiv skip$ be the empty instruction. The semantics of ι is defined by the extended identical state transformer $Id_{\Sigma'}$, i.e., $\theta_{\iota}^{lc} =_{df} Id_{\Sigma'}$, where $Id_{\Sigma'} : \Sigma' \to \Sigma'$ is defined by $\forall\, \sigma \in \Sigma'.\ Id_{\Sigma'}(\sigma) =_{df} \sigma$.

Chapter 5.4.2

Linear Constants: Specification

Linear Constants over $\mathbb{Z}$: DFA Specification

DFA Specification

- DFA lattice
  $$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_\perp, \sigma_\top) = \widehat{\Sigma'}$$
  with Σ' set of DFA states over $\mathbb{Z}$.
- DFA functional
  $[\![\ ]\!]_{lc} : E \to (\Sigma' \to \Sigma')$ where $\forall\, e \in E.\ [\![\, e\,]\!]_{lc} =_{df} \theta^{lc}_{\iota_e}$
- Initial information: $\sigma_{\mathbf{s}} \in \Sigma'_{Init}$
- Direction of information flow: forward

Linear Constants Specification

- Specification: $\mathcal{S}^{lc}_G = (\widehat{\Sigma'}, [\![\ ]\!]_{lc}, \sigma_{\mathbf{s}}, fw)$

Chapter 5.4.3

Termination, Safety, and Coincidence

Towards Coincidence and Termination

Lemma 5.4.3.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.4.3.2 (Distributivity)

$[\![\ ]\!]_{lc}$ is distributive.

Corollary 5.4.3.3 (Monotonicity)

$[\![\ ]\!]_{lc}$ is monotonic.

Termination and Coincidence/Optimality

Theorem 5.4.3.4 (Termination)

Applied to $\mathcal{S}_G^{lc} = (\widehat{\Sigma}', [\![\ ]\!]_{lc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{lc}$.

Proof. Immediately with Lemma 5.4.3.1, Lemma 5.4.3.3, and Termination Theorem 3.4.4.

Theorem 5.4.3.5 (Coincidence/Optimality)

Applied to $\mathcal{S}_G^{lc} = (\widehat{\Sigma}', [\![\ ]\!]_{lc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{lc}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{lc}$).

Proof. Immediately with Lemma 5.4.3.2, Coincidence Theorem 3.5.2, and Termination Theorem 5.4.3.4.

Chapter 5.4.4

Soundness and Completeness

Soundness, Non-Completeness of $MOP_{\mathcal{S}_G^{lc}}$

Theorem 5.4.4.1 (Soundness, Non-Completeness)

The *MOP* solution of $\mathcal{S}_G^{lc}$ is

1. sound but not complete for the variable constant propagation problem CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in {\Sigma'}_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{lc}}(n)$$

2. sound but not complete for the term constant propagation problem CT, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in {\Sigma'}_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MOP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{lc}}(n)$$

In general, both inclusions are proper inclusions.

Soundness, Non-Completeness of $MaxFP_{\mathcal{S}_G^{lc}}$

Corollary 5.4.4.2 (Soundness, Non-Completeness)

The *MaxFP* solution of $\mathcal{S}_G^{lc}$ is

1. sound but not complete for CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MaxFP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{lc}}(n)$$

2. sound but not complete for CT, i.e.

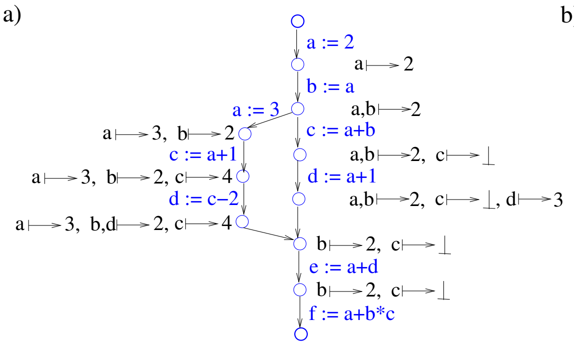
$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MaxFP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{lc}}(n)$$

In general, both inclusions are proper inclusions.

Chapter 5.4.5

Illustrating Example

Linear Constants over $\mathbb{Z}$: Illustrating Example



...the terms $a + b$, $a + d$, $b * c$, and $a + b * c$ are not linear constants, though they are simple constants (except of $a + 8 \equiv a + b * c$).

Chapter 5.5

Copy Constants

Chapter 5.5.1

DFA States, DFA Lattice

DFA States, DFA Lattice, and More

The copy constants (CpC) analysis shares the

- extended data domain ID'
- basic flat DFA lattice $\mathcal{FL}_{\mathsf{ID}'}$
- set of DFA states Σ'
- extended interpretation $I' =_{df} (\mathsf{ID}', I'_0)$

with the simple constants (SC) and the linear constants (LC) analysis.

CpC-specific are

- the adaption of the semantics of terms to ID'
- the adaption of the semantics of instructions to Σ'

CpC-specific Term Semantics Adaption

Definition 5.5.1.1 (Adapted Semantics of Terms)

The CpC-specific semantics of terms $t \in \mathbf{T}$ is defined by the extended evaluation function

$$\mathcal{E}_{cpc} \colon \mathbf{T} \to (\Sigma' \to \mathrm{I\!D}')$$

defined by

$$\forall t \in \mathbf{T}\ \forall \sigma \in \Sigma'.\ \mathcal{E}_{cpc}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ I'_0(c) & \text{if } t \equiv c \in \mathbf{C} \\ \perp & \text{otherwise} \end{cases}$$

CpC-specific Instructions Semantics Adaption

Definition 5.5.1.2 (Adapted Semantics of Instruc's)

- Let $\iota \equiv x := t$ be an assignment instruction. The semantics of ι is defined by the CpC-specific state transformer $\theta_\iota^{cpc} : \Sigma' \to \Sigma'$ defined by

$$\forall\, \sigma \in \Sigma'\ \forall\, y \in \mathbf{V}.\ \theta_\iota^{cpc}(\sigma)(y) =_{df} \begin{cases} \mathcal{E}_{cpc}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

- Let $\iota \equiv skip$ be the empty instruction. The semantics of ι is defined by the extended identical state transformer $Id_{\Sigma'}$, i.e., $\theta_\iota^{cpc} =_{df} Id_{\Sigma'}$, where $Id_{\Sigma'} : \Sigma' \to \Sigma'$ is defined by $\forall\, \sigma \in \Sigma'.\ Id_{\Sigma'}(\sigma) =_{df} \sigma$.

Chapter 5.5.2

Copy Constants: Specification

Copy Constants over $\mathbb{Z}$: DFA Specification

DFA Specification

- DFA lattice
  $$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\Sigma', \sqcap_{\Sigma'}, \sqcup_{\Sigma'}, \sqsubseteq_{\Sigma'}, \sigma_\perp, \sigma_\top) = \widehat{\Sigma}'$$
  with Σ' set of DFA states over $\mathbb{Z}$.
- DFA functional
  $[\![\ ]\!]_{cpc} : E \to (\Sigma' \to \Sigma')$ where $\forall\, e \in E.\ [\![\, e\,]\!]_{cpc} =_{df} \theta^{cpc}_{\iota_e}$
- Initial information: $\sigma_\mathbf{s} \in \Sigma'_{Init}$
- Direction of information flow: forward

Copy Constants Specification

- Specification: $\mathcal{S}^{cpc}_G = (\widehat{\Sigma}', [\![\ ]\!]_{cpc}, \sigma_\mathbf{s}, fw)$

Chapter 5.5.3

Termination, Safety, and Coincidence

Towards Coincidence and Termination

Lemma 5.5.3.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.5.3.2 (Distributivity)

$[\![\ ]\!]_{cpc}$ is distributive.

Corollary 5.5.3.3 (Monotonicity)

$[\![\ ]\!]_{cpc}$ is monotonic.

Termination and Coincidence/Optimality

Theorem 5.5.3.4 (Termination)

Applied to $\mathcal{S}_G^{cpc} = (\widehat{\Sigma}', [\![\ ]\!]_{cpc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{cpc}$.

Proof. Immediately with Lemma 5.5.3.1, Lemma 5.5.3.3, and Termination Theorem 3.4.4.

Theorem 5.5.3.5 (Coincidence/Optimality)

Applied to $\mathcal{S}_G^{cpc} = (\widehat{\Sigma}', [\![\ ]\!]_{cpc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{cpc}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{cpc}$).

Proof. Immediately with Lemma 5.5.3.2, Coincidence Theorem 3.5.2, and Termination Theorem 5.5.3.4.

Chapter 5.5.4

Soundness and Completeness

Soundness, Non-Completeness of $MOP_{\mathcal{S}_G^{cpc}}$

Theorem 5.5.4.1 (Soundness, Non-Completeness)

The MOP solution of $\mathcal{S}_G^{cpc}$ is

1. sound but not complete for the variable constant propagation problem CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{cpc}}(n)$$

2. sound but not complete for the term constant propagation problem CT, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MOP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{cpc}}(n)$$

In general, both inclusions are proper inclusions.

Soundness, Non-Completeness of $MaxFP_{\mathcal{S}_G^{cpc}}$

Corollary 5.4.4.2 (Soundness, Non-Completeness)

The *MaxFP* solution of $\mathcal{S}_G^{cpc}$ is

1. sound but not complete for CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MaxFP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{cpc}}(n)$$

2. sound but not complete for CT, i.e.

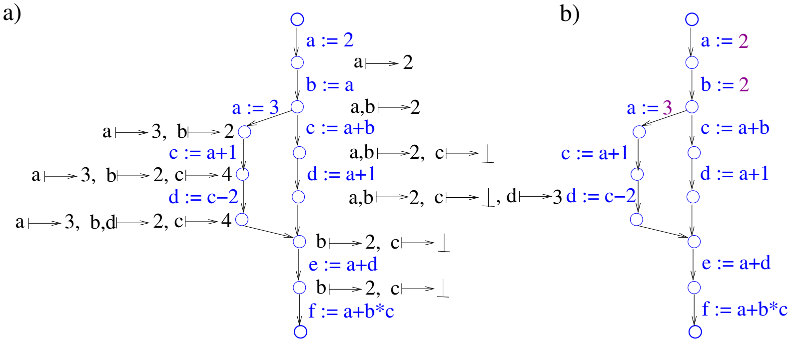
$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MaxFP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{cpc}}(n)$$

In general, both inclusions are proper inclusions.

Chapter 5.5.5

Illustrating Example

Copy Constants over $\mathbb{Z}$: Illustrating Example



...only the right-hand side terms 2, 3, and a, are copy constants, though many of the other terms are linear or simple constants.

Chapter 5.6

Q Constants

Chapter 5.6.1

Background and Motivation

Background and Motivation (1)

...the *MOP* solution as the specifying solution of a DFA problem is not decidable.

Theorem 3.3.3 (Undecidability, Kam&Ullman 1977) – recalled

There is no algorithm A satisfying:

- The input of A are
  - a DFA specification $\mathcal{S}_G = (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, fw)$
  - algorithms for the computation of the meet, the equality test, and the application of monotonic functions on the elements of a complete lattice
- The output of A is the *MOP* solution of $\mathcal{S}_G$.

Background and Motivation (2)

...for monotonic DFA problems, the *MaxFP* solution as their computable solution is generally a proper approximation of their *MOP* solution only.

Theorem 3.5.1 (Safety) – recalled

The *MaxFP* solution of $\mathcal{S}_G = (\widehat{\mathcal{C}}, [\![\ ]\!], c_s, fw)$ is a safe (i.e., lower) approximation of the *MOP* solution of $\mathcal{S}_G$, i.e.,

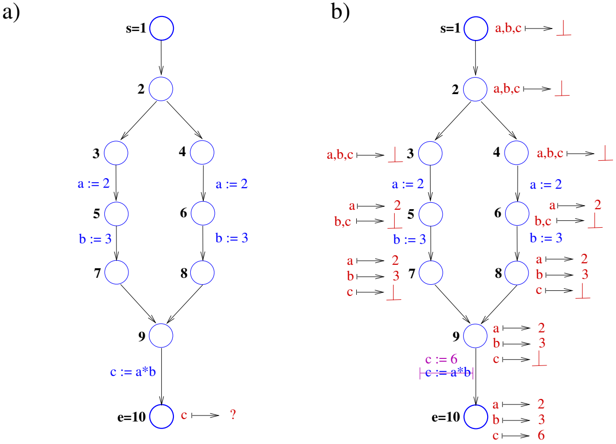
$$\forall\, n \in N.\ MaxFP_{\mathcal{S}_G}(n) \sqsubseteq MOP_{\mathcal{S}_G}(n)$$

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

The Impact of Monotonicity on SCs (1)

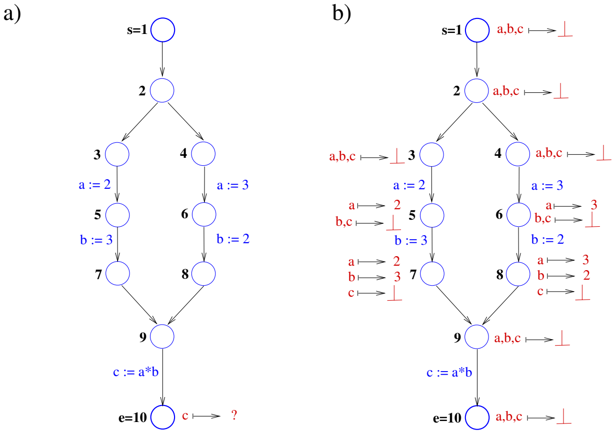
...SCs, a (non-distributive) monotonic DFA problem.

While $a * b$ is a simple constant in the example below...



The Impact of Monotonicity on SCs (2)

...it is not in the only slightly modified example below:



The Proposal of Kam and Ullman

To improve on this situation

- Kam and Ullman propose using a slightly modified fixed point approach to cope with (non-distributive) monotonic DFA problems.

We call this approach the

- Kam/Ullman *MaxFP* approach (or Q-*MaxFP* approach).

Chapter 5.6.2

The Q-*MaxFP* Approach

Preliminaries

Following Kam and Ullman, we start considering a node-labelled SI flow graph[1]

- $G = (N, E, \mathbf{s}, \mathbf{e})$

Let

- $\mathcal{S}_G = (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathbf{s}}, fw)$

be a monotonic (non-distributive) DFA specification with

$$[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$$

[1] We will adapt the Q approach to edge-labelled flow graphs later.

The *MaxFP* Approach (N-labelled Graphs)

...adapted for node-labelled SI graphs.

Equation System 5.6.2.1 (*MaxFP* EQS)

$$N\text{-}inf(n) = \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \prod\{X\text{-}inf(m) \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

$$X\text{-}inf(n) = [\![n]\!](N\text{-}inf(n))$$

The Q-*MaxFP* Approach (N-labelled Graphs)

Equation System 5.6.2.2 (Q-*MaxFP* EQS)

$$NQ\text{-}inf(n) = \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigsqcap\{XQ\text{-}inf(m) \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

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

...essential: delaying to joining (by $\sqcap$) information.

MaxFP Approach vs. Q-*MaxFP* Approach

MaxFP Approach – joining information early (eagerly):

Equation System 5.6.2.1′ (*MaxFP* EQS)

$$N\text{-}inf(n) \;=\; \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigsqcap\{X\text{-}inf(m) \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

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

Q-*MaxFP* Approach – joining information late (lazily):

Equation System 5.6.2.2 (Q-*MaxFP* EQS)

$$NQ\text{-}inf(n) \;=\; \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigsqcap\{XQ\text{-}inf(m) \mid m \in pred(n)\} & \text{otherwise} \end{cases}$$

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

Eager and Lazy Fixed Point Approach

The Equation Systems 5.6.2.1′ and 5.6.2.2 give rise to consider the

- *MaxFP* approach
- Q-*MaxFP* approach

the

- eager (eagerly joining)
- lazy (lazily joining)

fixed point approach, respectively.

MaxFP, Q-*MaxFP* Solutions (N-lab'ed Graphs)

Definition 5.6.2.3 (*MaxFP*, Q-*MaxFP* Solution)

For every node $n \in N$, the *MaxFP* and Q-*MaxFP* Solutions of $\mathcal{S}_G$ are defined by

- $N\text{-}MaxFP_{\mathcal{S}_G}(n) =_{df} N\text{-}inf^\star(n)$
  $X\text{-}MaxFP_{\mathcal{S}_G}(n) =_{df} X\text{-}inf^\star(n)$
- $NQ\text{-}MaxFP_{\mathcal{S}_G}(n) =_{df} NQ\text{-}inf^\star(n)$
  $XQ\text{-}MaxFP_{\mathcal{S}_G}(n) =_{df} XQ\text{-}inf^\star(n)$

where

- $N\text{-}inf^\star(n),\ X\text{-}inf^\star(n) : N \rightarrow \mathcal{C}$
- $NQ\text{-}inf^\star(n),\ XQ\text{-}inf^\star(n) : N \rightarrow \mathcal{C}$

denote the greatest solutions of Equation System 5.6.2.1′ and Equation System 5.6.2.2, respectively.

Q Approach: Improvement (N-lab'ed Graphs)

We have:

Lemma 5.6.2.4 (Q Improvement Lemma)

For every node $n \in N$, we have:

- $NQ\text{-}MaxFP_{\mathcal{S}_G}(n) \sqsupseteq N\text{-}MaxFP_{\mathcal{S}_G}(n)$
- $XQ\text{-}MaxFP_{\mathcal{S}_G}(n) \sqsupseteq X\text{-}MaxFP_{\mathcal{S}_G}(n)$

In general, all inclusions are proper inclusions.

Computing *MaxFP* and Q-*MaxFP* Solution: Pragmatics (N-labelled Graphs)

Note

- *XQ-inf** and *X-inf** can be computed without referring to (approximations of) *NQ-inf** and *N-inf**, respectively.

Once

- *XQ-inf** and *X-inf** have been computed *NQ-inf** and *N-inf** can be computed by visiting each node once. No further fixed point computation or iteration is required.

Computing *MaxFP* and Q-*MaxFP* Solution: Algorithms (N-labelled Graphs)

The greatest solutions of

- Equation System 5.6.2.1′
- Equation System 5.6.2.2

can be computed in the same fashion as the greatest solution of Equation System 3.4.1.

We denote these algorithms, which are generic straightforward adaptions of Algorithm 3.4.3, by

- *MaxFP* Algorithm 5.6.2.5[2]
- Q-*MaxFP* Algorithm 5.6.2.6[2]

respectively.

[2] We omit presenting the algorithms explicitly.

Q Approach: Main Results (N-lab'ed Graphs)

Theorem 5.6.2.7 (Termination)

The generic *MaxFP* Algorithm 5.6.2.5 and the generic Q-*MaxFP* Algorithm 5.6.2.6 terminate with the *MaxFP* solution and the Q-*MaxFP* solution of $\mathcal{S}_G$, respectively, if (1) the DFA lattice $\widehat{\mathcal{C}}$ satisfies the descending chain condition, and (2) the DFA functional $[\![\ ]\!]$ is monotonic.

Theorem 5.6.2.8 (Safety)

The *MaxFP* solution and the Q-*MaxFP* solution of $\mathcal{S}_G$ are safe (i.e., lower) approximations of the *MOP* solution of $\mathcal{S}_G$ satisfying for every node $n \in N$:

- $N\text{-}MOP_{\mathcal{S}_G}(n) \sqsupseteq NQ\text{-}MaxFP_{\mathcal{S}_G}(n) \sqsupseteq N\text{-}MaxFP_{\mathcal{S}_G}(n)$
- $X\text{-}MOP_{\mathcal{S}_G}(n) \sqsupseteq XQ\text{-}MaxFP_{\mathcal{S}_G}(n) \sqsupseteq X\text{-}MaxFP_{\mathcal{S}_G}(n)$

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

In general, all inclusions are proper inclusions.

Q Approach: From N to E-labelled Graphs

...focusing on node exits of node-labelled SI flow graphs

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

yields the key for adapting the Q-*MaxFP* approach to edge-labelled SI flow graphs.

The Q-*MaxFP* Approach (E-labelled Graphs)

...adapted to edge-labelled SI graphs.

Equation System 5.6.2.9 (Q-*MaxFP* EQS)

$$Q\text{-}inf(e) = \begin{cases} [\![\, e \,]\!](c_{\mathbf{s}}) & \text{if } start(e) = \mathbf{s} \\ \bigsqcap \{\, [\![\, e \,]\!](Q\text{-}inf(f)) \mid f \in pred(e) \,\} & \text{otherwise} \end{cases}$$

where $pred(e) =_{df} \{ f \mid end(f) = start(e) \}$.

Recall and compare with:

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

Q-*MaxFP* Solution (E-labelled Graphs)

Definition 5.6.2.10 (Q-*MaxFP* Solution)

For every edge $e \in E$, the Q-*MaxFP* Solution of $\mathcal{S}_G$ is defined by

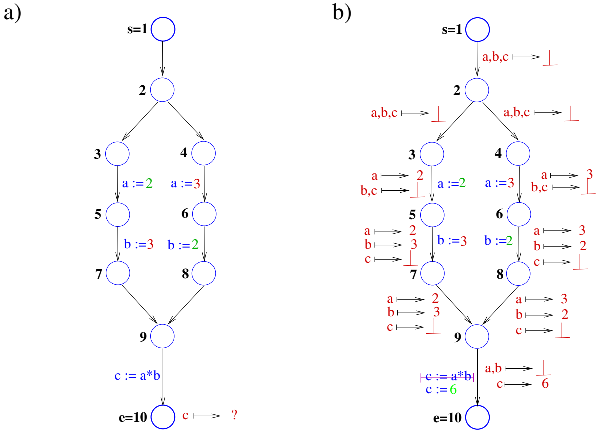
- $Q\text{-}MaxFP_{\mathcal{S}_G}(e) =_{df} Q\text{-}inf^{\star}(e)$

where

- $Q\text{-}inf^{\star} : E \to \mathcal{C}$

denotes the greatest solution of Equation System 5.6.2.9.

Illustrating the Q-*MaxFP* Approach and the Q-*MaxFP* Solution (E-lab'ed Graphs)



Q-*MaxFP* Solution (E-lab'ed Graphs): Induced Node Annotation

The greatest solution of Equation System 5.6.2.9, $Q\text{-}inf^\star(e)$, $e \in E$, induces an annotation of the nodes of G as follows:

Definition 5.6.2.11 (Induced Node Annotation)

For every node $n \in N$, we define:

$$Q\text{-}MaxFP(n) =_{df} Q\text{-}inf^\star(n) =_{df}$$

$$\begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \prod \{\, Q\text{-}inf^\star(e) \mid end(e) = n \,\} & \text{otherwise} \end{cases}$$

Note: There is no fixed point computation involved in computing $Q\text{-}inf^\star(n)$, $n \in N$.

Q Approach: Improvement (E-lab'ed Graphs)

We have:

Lemma 5.6.2.12 (Q Improvement Lemma)

For every node $n \in N$, for every $e \in E$ with $end(e) = n$, we have:

- $Q\text{-}MaxFP_{\mathcal{S}_G}(e) \sqsupseteq Q\text{-}MaxFP_{\mathcal{S}_G}(n) \sqsupseteq MaxFP_{\mathcal{S}_G}(n)$

In general, all inclusions are proper inclusions.

Computing the Q-*MaxFP* Solution: Algorithm (E-labelled Graphs)

The greatest solution of

- Equation System 5.6.2.9

can be computed in the same fashion as the greatest solution of Equation System 3.4.1.

We denote this algorithm, which is a generic straightforward adaption of Algorithm 3.4.3, by

- Q-*MaxFP*-Algorithm 5.6.2.13[3]

[3] We omit presenting the algorithm explicitly.

Q Approach: Main Results (E-lab'ed Graphs)

Theorem 5.6.2.14 (Termination)

The generic Q-*MaxFP* Algorithm 5.6.2.13 terminates with the Q-*MaxFP* solution of $\mathcal{S}_G$, if (1) the DFA lattice $\widehat{\mathcal{C}}$ satisfies the descending chain condition, and (2) the DFA functional $[\![\ ]\!]$ is monotonic.

Theorem 5.6.2.15 (Safety)

The Q-*MaxFP* solution is a safe (i.e., lower) approximation of the *MOP* solution of $\mathcal{S}_G$ satisfying for every node $n \in N$, for every $e \in E$ with $end(e) = n$:

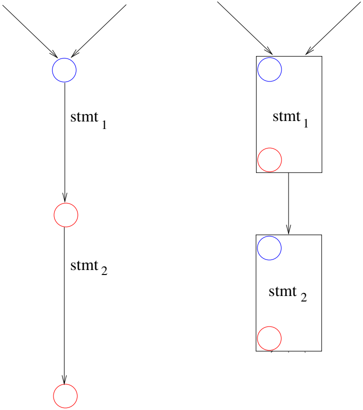
- $$\begin{aligned} MOP_{\mathcal{S}_G}(n) &\sqsupseteq Q\text{-}MaxFP_{\mathcal{S}_G}(e) \\ &\sqsupseteq Q\text{-}MaxFP_{\mathcal{S}_G}(n) \sqsupseteq MaxFP_{\mathcal{S}_G}(n) \end{aligned}$$

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

In general, all inclusions are proper inclusions.

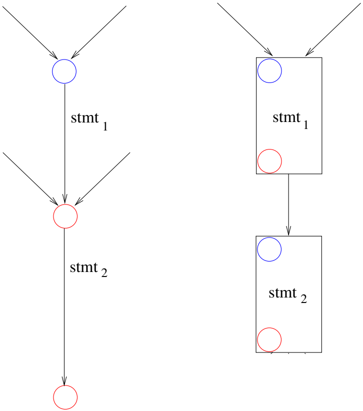
Q Approach: Choosing N or E-labelled Graphs

...at first sight, both variants appear equally suited.



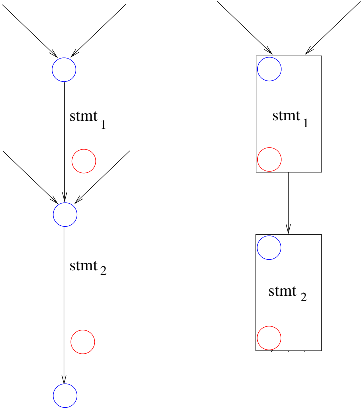
Q Approach: Choosing N or E-labelled Graphs

However, a closer look reveals...



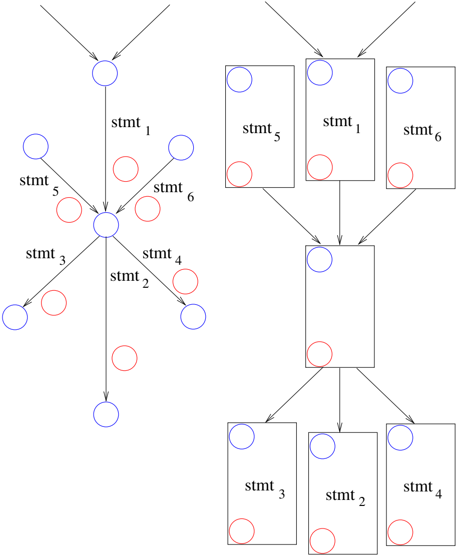
Q Approach: Choosing N or E-labelled Graphs

...edge-labelled graphs are more compact



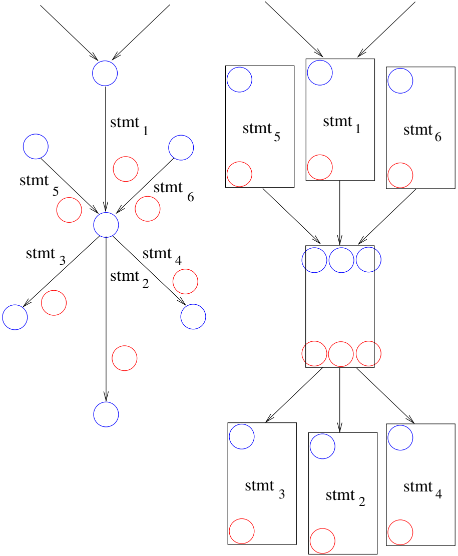
Q Approach: Choosing N or E-labelled Graphs

...making them more appropriate



Q Approach: Choosing N or E-labelled Graphs

...for taking advantage of the Q heuristics.



Q Approach: Choosing N or E-labelled Graphs

The Q approach

- applies to both node and edge-labelled flow graphs.

The heuristics of the Q approach, however,

- is more effective on edge-labelled flow graphs than on node-labelled ones because of their greater compactness.

Chapter 5.6.3

Q Constants: The Specification

Q Constants over $\mathbb{Z}$: DFA Specification

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be an edge-labelled SI flow graph.

Q Constants Specification

- Specification: $\mathcal{S}_G^{qc} =_{df} \mathcal{S}_G^{sc} = (\widehat{\Sigma'}, [\![\ ]\!]_{sc}, \sigma_{\mathbf{s}}, fw)$

Note

- Q constants (QCs) and simple constants (SCs) share the same specification.
- The only difference between the QC and SC problem is that $\mathcal{S}_G^{sc}$ and $\mathcal{S}_G^{qc}$ are fed into and solved by the *MaxFP* and Q-*MaxFP* approach, respectively.

Chapter 5.6.4

Termination, Safety, and Coincidence

Towards Safety and Termination

Because of $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', [\![\ ]\!]_{qc}, \sigma_{\mathbf{s}}, fw) = (\widehat{\Sigma}', [\![\ ]\!]_{sc}, \sigma_{\mathbf{s}}, fw) = \mathcal{S}_G^{sc}$ we get as corollaries of Lemma 5.3.3.1, 5.3.3.2, and 5.3.3.3:

Corollary 5.6.4.1 (Descending Chain Condition)

$\widehat{\Sigma}'$ satisfies the descending chain condition.

Corollary 5.6.4.2 (Monotonicity)

$[\![\ ]\!]_{qc} (= [\![\ ]\!]_{sc})$ is monotonic.

Corollary 5.6.4.3 (Non-Distributivity)

$[\![\ ]\!]_{qc} (= [\![\ ]\!]_{sc})$ is not distributive.

Termination and Safety/Conservativity

Theorem 5.6.4.4 (Termination)

Applied to $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', [\![\ ]\!]_{qc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 5.6.2.13 terminates with the Q-*MaxFP* solution of $\mathcal{S}_G^{qc}$.

Proof. Immediately with Corollary 5.6.4.1, Corollary 5.6.4.2, and Termination Theorem 5.6.2.14.

Theorem 5.6.4.5 (Safety/Conservativity)

Applied to $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', [\![\ ]\!]_{qc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 5.6.2.13 is *MOP* conservative for $\mathcal{S}_G^{qc}$ (i.e., it terminates with a lower approximation of the *MOP* solution of $\mathcal{S}_G^{qc}$).

Proof. Immediately with Corollary 5.6.4.2, Safety Theorem 5.6.2.15, and Termination Theorem 5.6.4.4.

Non-Coincidence

Theorem 5.6.4.6 (Non-Coincidence/Non-Opt.)

Applied to $\mathcal{S}_G^{qc} = (\widehat{\Sigma}', [\![\ ]\!]_{qc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 5.6.2.13 is in general not *MOP* optimal for $\mathcal{S}_G^{qc}$ (i.e., it terminates with a properly lower approximation of the *MOP* solution of $\mathcal{S}_G^{qc}$).

Proof. Immediately with Corollary 5.6.4.3, Safety Theorem 5.6.2.15, and Termination Theorem 5.6.4.4.

Corollary 5.6.4.7 (Safety, Non-Coincidence)

The *Q-MaxFP* solution for $\mathcal{S}_G^{qc}$, is always a safe approximation of the *MOP* solution of $\mathcal{S}_G^{qc}$. In general, the *MOP* solution and the *Q-MaxFP* solution of $\mathcal{S}_G^{qc}$ do not coincide.

Chapter 5.6.5

Soundness and Completeness

Soundness and Completeness of $MOP_{\mathcal{S}_G^{qc}}$

Because of $\mathcal{S}_G^{qc} = \mathcal{S}_G^{sc}$ we have $MOP_{\mathcal{S}_G^{qc}} = MOP_{\mathcal{S}_G^{sc}}$, and thus can conclude as an immediate corollary of Theorem 5.3.4.1:

Corollary 5.6.5.1 (Soundness and Completeness)

The *MOP* solution of $\mathcal{S}_G^{qc}$ is

1. sound and complete for the variable constant propagation problem CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) = MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{qc}}(n)$$

2. sound but not complete for the term constant propagation problem CT, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MOP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{qc}}(n)$$

In general, the inclusion is a proper inclusion.

Soundness and Completeness of Q-$MaxFP_{\mathcal{S}_G^{qc}}$

Corollary 5.6.5.2 (Soundness and Completeness)

The Q-*MaxFP* solution of $\mathcal{S}_G^{qc}$ is

1. sound but not complete for CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{V}}_{CV_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} Q\text{-}MaxFP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{sc}}(n)$$

2. sound but not complete for CT, i.e.

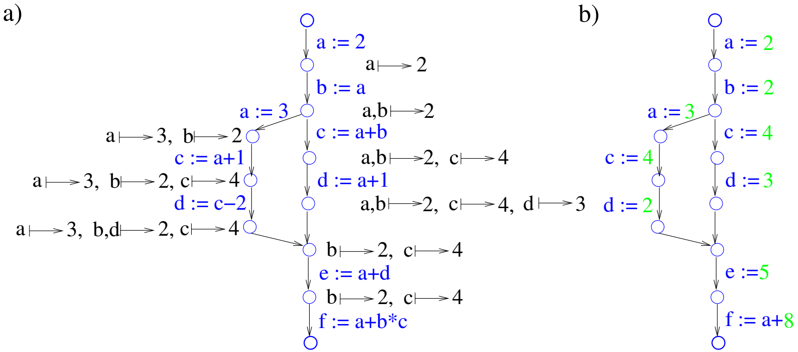
$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} Q\text{-}MaxFP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{qc}}(n)$$

In general, both inclusions are proper inclusions.

Chapter 5.6.6

Illustrating Example

Q Constants over ℤ: Illustrating Example



...all terms are Q constants except of $a + 8$, which, however, is not a (non-deterministic) constant at all.

Chapter 5.6.7

Summary

The Essence of the Q Approach

The Q approach is

- a heuristic approach to cope with the information loss caused by "early (eagerly)" joining information in the *MaxFP* approach for (non-distributive) monotonic DFA problems.

Intuitively

- the Q approach accomplishes "a look-ahead of one edge" by joining information "late (lazily)" avoiding thereby the loss of information in part.

Benefits and Limitations of the Q Approach

Benefits

Introduced and proposed with an application to constant propagation (i.e., Q constants)

- the Q approach can beneficially be used for every monotonic DFA problem at (in practice) almost no additional costs compared to the standard *MaxFP* approach.

Limitations

- In practice, the impact of the Q approach on improving the precision of analysis results will be limited because its look-ahead heuristics is limited to one edge.
- Avoiding the loss of information by joining information completely, the look-ahead would need to be arbitrarily large in general; there is no finite upper limit on the required look-ahead for avoiding information loss.

Chapter 5.7

Finite Constants

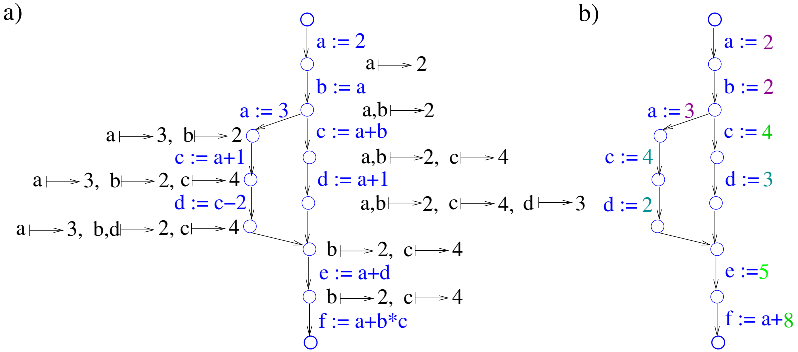
Chapter 5.7.1

Background and Movitation

CP by Q Constants: Everything Alright?

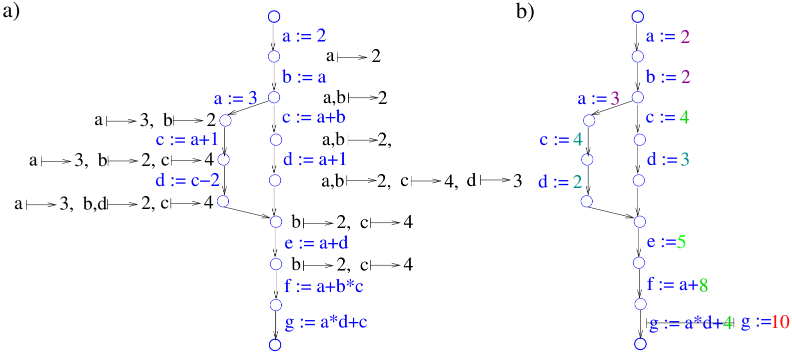
Note

- All terms except of $a + 8$, which is not a (non-deterministic) constant, are Q constants.



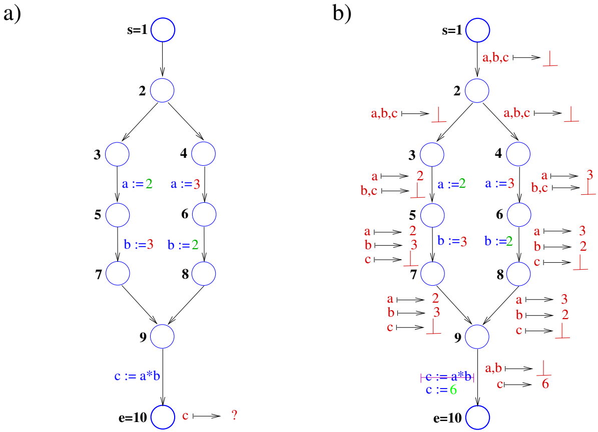
Unfortunately not

...$a * d$ and g are constants of value 6 and 10, respectively, however, they are no Q constants.



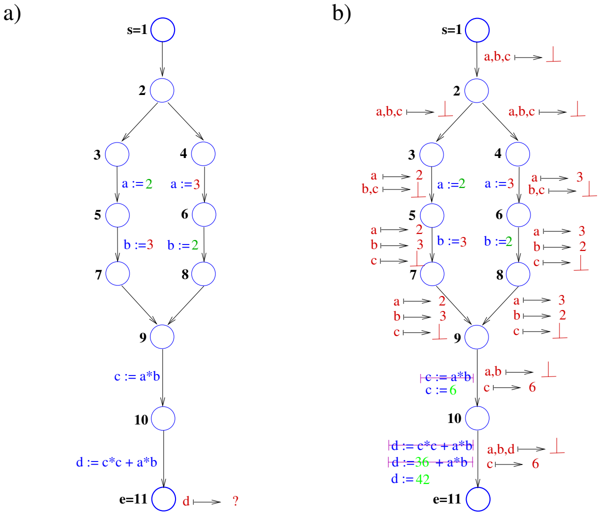
Achievement of the Q Constants Heuristics

...joining information "lazily" accomplishes a "look-ahead" of 1 edge after a join node:



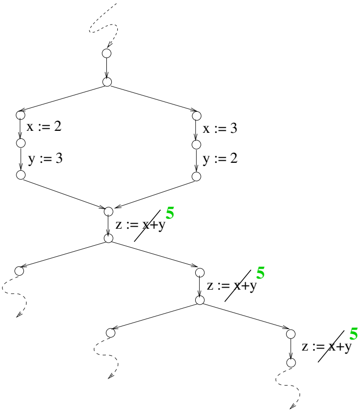
Limitations of the Q Constants Heuristics

...but not of 2 as required here (or even more in general):



The Look-Ahead Challenge

...a need for a look-ahead of unlimited length in general:



After Finite Constants Propagtion

...the approach for finite constants deals systematically with this challenge.

Chapter 5.7.2

Finite Constants: The Very Idea

Finite Constants: The Very Idea

Intuitively

- finite constants achieve a look-ahead of arbitrary but finite depth

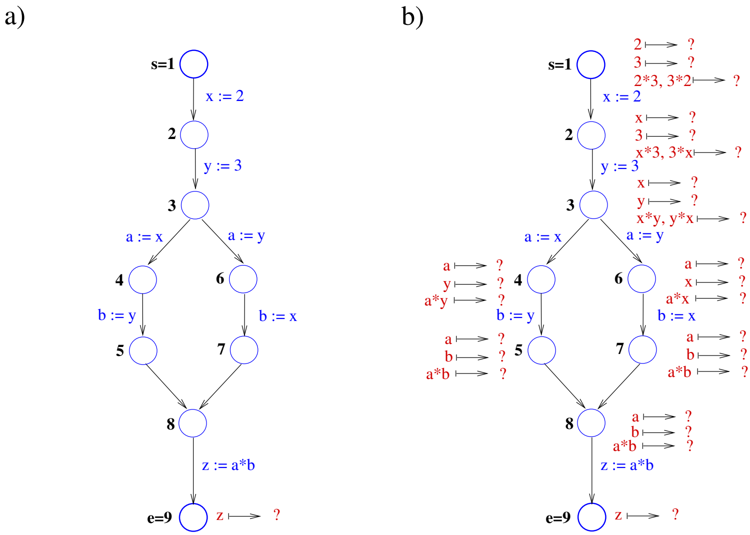
Technically, this is achieved

- by pre-computing for every program point a finite set of interesting terms and
- focusing the analysis at every program point to this set of terms instead of the program variables only.

Hence

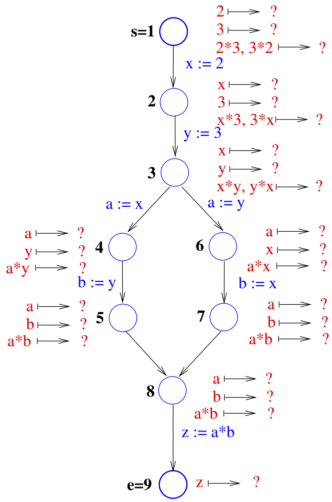
- unlike the other CP algorithms, the CP algorithm for finite constants is a CT algorithm, not a CV algorithm.

Finite Constants: Pre-Computing Term Sets

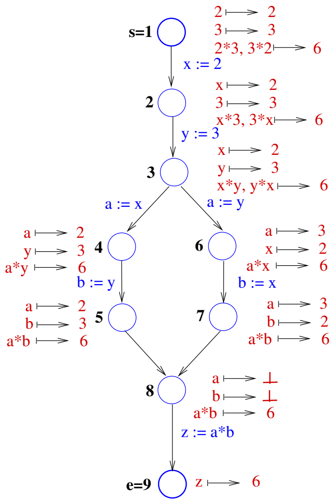


Finite Constants: The Analysis (Conceptually)

a)



b)



Chapter 5.7.3

Finite Operational Constants

Backward Substitution for Instructions

Definition 5.7.3.1 (Backward Substitution δ)

- Let $\iota \equiv x := t$ be an assignment instruction. The backward substitution of ι is the function $\delta_\iota : \mathbf{T} \to \mathbf{T}$ defined by
$$\forall\, t' \in \mathbf{T}.\ \delta_\iota(t') =_{df} t'[t/x]$$
- Let $\iota \equiv \mathit{skip}$ be the empty instruction. The backward substitution of ι is the identical mapping on the set of terms $\mathbf{T}$, i.e., $\delta_\iota =_{df} Id_{\mathbf{T}}$, where $Id_{\mathbf{T}} : \mathbf{T} \to \mathbf{T}$ is defined by $\forall\, t \in \mathbf{T}.\ Id_{\mathbf{T}}(t) =_{df} t$.

Backward Substitution for Paths

Definition 5.7.3.2 (Extending δ to Paths)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph. The backward substitutions δ_{ι_e} of instructions at edges $e \in E$ are extended onto paths $p = \langle e_1, e_2, \ldots, e_q \rangle$ in G by defining:

$$\delta_p =_{df} \begin{cases} Id_{\mathbf{T}} & \text{if } q < 1 \\ \delta_{\langle e_1, \ldots, e_{q-1} \rangle} \circ \delta_{\iota_{e_q}} & \text{otherwise} \end{cases}$$

Substitution Lemma: Relating δ and θ

Lemma 5.7.3.3 (Substitution Lemma for Edges)

$$\forall\, t \in \mathbf{T}.\ \forall\, e \in E.\ \forall\, \sigma \in \Sigma.\ \mathcal{E}(\delta_{\iota_e}(t))(\sigma) = \mathcal{E}(t)(\theta_{\iota_e}(\sigma))$$

Lemma 5.7.3.4 (Substitution Lemma for Paths)

$$\forall\, t \in \mathbf{T}.\ \forall\, n \in N.\ \forall\, \sigma \in \Sigma.\ \forall\, p \in \mathbf{P}[m, n].$$

$$\mathcal{E}(\delta_p(t))(\sigma) = \mathcal{E}(t)(\theta_p(\sigma))$$

Corollary 5.7.3.5 (Substitution L. for Paths from $\mathbf{s}$)

$$\forall\, t \in \mathbf{T}.\ \forall\, n \in N.\ \forall\, \sigma \in \Sigma.\ \forall\, p \in \mathbf{P}[\mathbf{s}, n].$$

$$\mathcal{E}(\delta_p(t))(\sigma_{\mathbf{s}}) = \mathcal{E}(t)(\theta_p(\sigma_{\mathbf{s}}))$$

t-Associated Paths

Definition 5.7.3.6 (t-Associated Path)

Let $p = \langle e_1, e_2, \ldots, e_q \rangle$ be a path, and let $t \in \mathbf{T}$ be a term. The t-associated path p_t for p is defined by

$$p_t = \langle (t_1, e_1), (t_2, e_2), \ldots, (t_q, e_q) \rangle$$

with $t_q = \delta_{\iota_{e_q}}(t)$ and $t_j = \delta_{\iota_e}(t_{j+1})$ for all $1 \leq j < q$.

Corollary 5.7.3.7 (Subst. L. for t-assoc. Paths)

$$\forall\, t \in \mathbf{T}.\ \forall\, n \in N.\ \forall\, \sigma \in \Sigma.\ \forall\, p \in \mathbf{P}[\mathbf{s}, n].$$

$$\mathcal{E}(t_{\mathbf{s}})(\sigma_{\mathbf{s}}) = \mathcal{E}(t)(\theta_p(\sigma_{\mathbf{s}}))$$

where $p_t = \langle (t_1, e_1), (t_2, e_2), \ldots, (t_q, e_q) \rangle$ is the t-associated path for p and $t_{\mathbf{s}} \equiv t_1$.

Relevant Paths

Definition 5.7.3.8 (Relevant Path of length k)

A t-associated path $p_t = \langle (t_1, e_1), (t_2, e_2), \ldots, (t_q, e_q) \rangle$ is called a relevant path of length (at most) k for t and n, iff

- $dst(e_q) = n \;\wedge\; t_q = \delta_{\iota_{e_q}}(t)$
- $q = k \;\vee\; (q < k \;\wedge\; src(e_1) = \mathbf{s})$
- $\forall\, i, j \in \{1, \ldots, q\}.\ (t_i, e_i) = (t_j, e_j) \Rightarrow i = j$

The set of all relevant paths of length (at most) k for t and n is denoted by $\mathbf{RP}_k(t, n)$.

Definition 5.7.3.9 (Relevant Paths from $\mathbf{s}$ to n)

The set of all relevant paths from the start node $\mathbf{s}$ to node n is denoted by $\mathbf{RP}_{\mathbb{N}}(t, n)$.

Finite Operational Constants

Definition 5.7.3.10 (Finite Operational Constants)

Let $k \in \mathbb{N} \cup \{\mathbb{N}\}$, $d \in \mathbb{D} \backslash \{\perp\}$, $n \in N$, $\sigma_{\mathbf{s}} \in \Sigma_{Init}$. Then:

1. t is a k-constant of value d at node n for $\sigma_{\mathbf{s}}$, $t \in C_k^{\sigma_{\mathbf{s}}}(n, d)$, iff

$$\forall\, p_t = \langle (t_1, e_1), (t_2, e_2), \ldots, (t_q, e_q) \rangle \in \mathbf{RP}_k(t, n).$$

$$\mathcal{E}(t_1)(\sigma_{\mathbf{s}}) = d$$

2. the set of finite operational constants of value d at node n for $\sigma_{\mathbf{s}}$, $C_{fop}^{\sigma_{\mathbf{s}}}(n, d)$, by

$$C_{fop}^{\sigma_{\mathbf{s}}}(n, d) =_{df} \bigcup \{\, C_k^{\sigma_{\mathbf{s}}}(n, d) \mid k \in \mathbb{N} \}$$

3. the set of operational constants of value d at node n for $\sigma_{\mathbf{s}}$, $C_{op}^{\sigma_{\mathbf{s}}}(n, d)$, by

$$C_{op}^{\sigma_{\mathbf{s}}}(n, d) =_{df} \bigcup \{\, C_k^{\sigma_{\mathbf{s}}}(n, d) \mid k \in \mathbb{N} \cup \{\mathbb{N}\} \}$$

Finite Operational Constants: Main Result

For $n \in N$ and $\sigma_{\mathbf{s}} \in \Sigma_{Init}$, let

$$C_{op}^{\sigma_{\mathbf{s}}}(n) =_{df} \bigcup \{ C_{op}^{\sigma_{\mathbf{s}}}(n, d) \mid d \in \mathbb{D} \backslash \{\perp\} \}$$

Theorem 5.7.3.11 (Main Result)

Let $n \in N$, $\sigma_{\mathbf{s}} \in \Sigma_{Init}$, and $d \in \mathbb{D} \backslash \{\perp\}$. Then we have:

1. $C_{op}^{\sigma_{\mathbf{s}}}(n) = CT_{G}^{\sigma_{\mathbf{s}}}(n)$
2. $\exists\, k \in \mathbb{N}.\ C_{fop}^{\sigma_{\mathbf{s}}}(n, d) = C_{k}^{\sigma_{\mathbf{s}}}(n, d)$

Chapter 5.7.4

Finite Denotational Constants

Partitions

Definition 5.7.4.1 (Partitions)

Let $T \subseteq \mathbf{T}$ be a set of terms. Then we define:

- $Part(T)$ denotes the set of all partitions of T.
- $Part =_{df} \bigcup \{Part(T) \mid T \subseteq \mathbf{T}\}$
- $CSet(p) =_{df} \{t \mid t \text{ lies in a class of } p\}$, $p \in Part$, denotes the carrier set of partition p.

Note

- Partitions can be viewed as equivalence relations on their carrier sets.
- This allows us to define a meet and a join operation on *Part* in terms of the set theoretical intersection and union of the equivalence relations corresponding to the partitions, respectively.

Induced Partitions

The evaluation function $\mathcal{E}$ induces for every set of terms $T \subseteq \mathbf{T}$ and every initial state $\sigma_\mathbf{s} \in \Sigma_{Init}$ a unique partition $Part_\mathcal{E}^{\sigma_\mathbf{s}}(T)$ with carrier set T.

Definition 5.7.4.2 (Induced Partitions)

Let $T \subseteq \mathbf{T}$, and $\sigma_\mathbf{s} \in \Sigma_{Init}$. Then we define:

- $\forall\, t_1, t_2 \in T.\ (t_1, t_2) \in Part_\mathcal{E}^{\sigma_\mathbf{s}}(T)$
$$\iff \mathcal{E}(t_1)(\sigma_\mathbf{s}) = \mathcal{E}(t_2)(\sigma_\mathbf{s})$$
- $Part_\mathcal{E}^{\sigma_\mathbf{s}} =_{df} \{Part_\mathcal{E}^{\sigma_\mathbf{s}}(T) \mid T \subseteq \mathbf{T}\}$ denotes the set of initial partitions induced by $\sigma_\mathbf{s}$.

Complete Lattice of the Set of Partitions

Lemma 5.7.4.3 (Complete Lattice)

The quintuple

$$\widehat{Part} =_{df} (Part, \sqcap, \sqcup, \sqsubseteq, \{\{t\} \mid t \in \mathbf{T}\}, \{\mathbf{T}\})$$

is a complete lattice, where $\sqcap$, $\sqcup$, and $\sqsubseteq$ are given by the set theoretical intersection, union, and subset relation of the equivalence relations represented by the partitions, respectively.

Local Abstract Semantics

Definition 5.7.4.4 (Local Abstract Semantics)

The semantic functional

$$[\![\ ]\!] : E \to (Part \to Part)$$

defines a local abstract semantics (for edges) by

$$\forall\, e \in E.\ \forall\, p \in Part.\ [\![\, e\,]\!](p) =_{df} \{(r,s) \mid (\delta_{\iota_e}(r), \delta_{\iota_e}(s)) \in p\}$$

Lemma 5.7.4.5 (Distributivity)

The local semantic functions $[\![\, e\,]\!]$, $e \in E$, are distributive.

Finite Denotational Constants

Definition 5.7.4.6 (Finite Denotational Constants)

Let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$, let $p_{\mathbf{s}} \in Part_{\mathcal{E}}^{\sigma_{\mathbf{s}}}$, let $d \in \mathrm{ID} \backslash \{\perp\}$, and let $n \in N$. Then we define:

- t is a $p_{\mathbf{s}}$-constant of value d at node n, $t \in C_{p_{\mathbf{s}}}(n, d)$, iff $(t, d) \in inf^{\star}_{p_{\mathbf{s}}}(n)$, where $inf^{\star}_{p_{\mathbf{s}}}$ denotes the greatest solution of the *MaxFP* Equation System 3.4.1.
- the set of finite denotational constants of value d at node n for $\sigma_{\mathbf{s}}$, $C^{\sigma_{\mathbf{s}}}_{fden}(n, d)$, by

$$C^{\sigma_{\mathbf{s}}}_{fden}(n, d) =_{df} \bigcup \{ C_p(n, d) \mid p \in Part_{\mathcal{E}}^{\sigma_{\mathbf{s}}} \wedge |CSet(p)| \in \mathrm{IN} \}$$

- the set of denotational constants of value d at node n for $\sigma_{\mathbf{s}}$, $C^{\sigma_{\mathbf{s}}}_{den}(n, d)$, by

$$C^{\sigma_{\mathbf{s}}}_{den}(n, d) =_{df} \bigcup \{ C_p(n, d) \mid p \in Part_{\mathcal{E}}^{\sigma_{\mathbf{s}}} \}$$

Computability of $inf^{\star}_{p_s}$

Theorem 5.7.4.7 ($inf^{\star}_{p_s}$-Theorem)

Let $\sigma_s \in \Sigma_{Init}$, and let $p_s \in Part^{\sigma_s}_{\mathcal{E}}$ be an initial partition with finite carrier set $CSet(p_s)$, i.e., $|CSet(p_s)| \in \mathbb{N}$. Then we have:

The *MaxFP* Algorithm 3.4.3 terminates with the greatest solution of Equation System 3.4.1, hence effectively computing

$$inf^{\star}_{p_s}(n),\ \ n \in N$$

In particular, Algorithm 3.4.3 computes for every node $n \in N$ and value $d \in \mathbb{D} \backslash \{\perp\}$ the set

$$C_{p_s}(n, d)$$

of p_s-constants of value d at node n.

Finite Denotational Constants: Main Result

For $n \in N$ and $\sigma_\mathbf{s} \in \Sigma_{Init}$, let:

$$C^{\sigma_\mathbf{s}}_{den}(n) =_{df} \bigcup \{\, C^{\sigma_\mathbf{s}}_{den}(n,d) \mid d \in \mathrm{ID} \backslash \{\bot\} \}$$

Theorem 5.7.4.8 (Main Result)

Let $n \in N$, $\sigma_\mathbf{s} \in \Sigma_{Init}$, and $d \in \mathrm{ID} \backslash \{\bot\}$. Then we have:

1. $C^{\sigma_\mathbf{s}}_{den}(n) = CT^{\sigma_\mathbf{s}}_{G}(n)$
2. $(\forall\, T \subseteq \mathbf{T}.\ |T| \in \mathrm{IN}).\ (\exists\, p_{fdc} \in Part^{\sigma_\mathbf{s}}_{\mathcal{E}}.$
   $|CSet(p_{fdc})| \in \mathrm{IN} \ \wedge\ C^{\sigma_\mathbf{s}}_{fden}(n,d) \cap T \subseteq C_{p_{fdc}}(n,d)\,)$

Chapter 5.7.5

Finite Constants

Equivalence

... of (finite) operational and (finite) denotational constants.

Theorem 5.7.5.1 (Equivalence)

Let $n \in N$, $\sigma_{\mathbf{s}} \in \Sigma_{Init}$, and $d \in \mathrm{ID} \backslash \{\perp\}$. Then we have:

1. $C^{\sigma_{\mathbf{s}}}_{op}(n,d) = \{ t \mid (t,d) \in CT^{\sigma_{\mathbf{s}}}_{G}(n) \} = C^{\sigma_{\mathbf{s}}}_{den}(n,d)$
2. $C^{\sigma_{\mathbf{s}}}_{fop}(n,d) = C^{\sigma_{\mathbf{s}}}_{fden}(n,d)$

Corollary 5.7.5.2 (Equivalence)

Let $n \in N$, and let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$. Then we have:

1. $C^{\sigma_{\mathbf{s}}}_{op}(n) = CT^{\sigma_{\mathbf{s}}}_{G}(n) = C^{\sigma_{\mathbf{s}}}_{den}(n)$
2. $C^{\sigma_{\mathbf{s}}}_{fop}(n) = C^{\sigma_{\mathbf{s}}}_{fden}(n)$

Finite Constants

Definition 5.7.5.3 (Finite Constants)

Let $n \in N$ and $\sigma_{\mathbf{s}} \in \Sigma_{Init}$. Then the set of finite constants is defined by

$$\begin{array}{rcl} C_{fc}^{\sigma_{\mathbf{s}}}(n) & =_{df} & \bigcup \{ C_{fop}^{\sigma_{\mathbf{s}}}(n,d) \mid d \in \mathbb{D} \backslash \{\perp\} \} \\ & = & \bigcup \{ C_{fden}^{p_{\mathbf{s}}}(n,d) \mid d \in \mathbb{D} \backslash \{\perp\} \} \end{array}$$

Optimality of Finite Constants

...for acyclic control flow.

Theorem 5.7.5.4 (Optimality for Acyclic Graphs)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be an acyclic flow graph, let $n \in N$ and $\sigma_{\mathbf{s}} \in \Sigma_{Init}$. Then we have:

$$C_{fop}^{\sigma_{\mathbf{s}}}(n) = CT_G^{\sigma_{\mathbf{s}}}(n) = C_{fden}^{\sigma_{\mathbf{s}}}(n)$$

Chapter 5.7.6

Deciding Finite Constants: Algorithm Sketch

The Decision Algorithm for Finite Constants

The decision algorithm for finite constants is essentially a two-stage procedure, sketched below:

Algorithm 5.7.6.1 (Decision Algorithm Sketch)

Let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$ be an initial state, let $t \in \mathbf{T}$, and let $n \in N$.

1. Algorithm 5.7.6.2:
   Compute a finite subset $TS_G(n,t) \subseteq \mathbf{T}$ such that all finite subsets $T_{finite} \subseteq \mathbf{T}$ satisfy:

   $$\forall\, t \in \mathbf{T}.\ \forall\, d \in \mathbb{D} \backslash \{\perp\}.$$
   $$t \in C_{Part_{\mathcal{E}}^{\sigma_{\mathbf{s}}}(T_{finite})}(n,d) \Rightarrow t \in C_{Part_{\mathcal{E}}^{\sigma_{\mathbf{s}}}(TS_G(n,t))}(n,d)$$

2. *MaxFP* Algorithm 3.4.3:
   Compute $C_{Part_{\mathcal{E}}^{\sigma_{\mathbf{s}}}(TS_G(n,t))}(n,d)$ for all values $d \in \mathbb{D} \backslash \{\perp\}$.

Initial Partition: Carrier Set Computation (1)

Algorithm 5.7.6.2 (Computing the Carrier Set of the Initial Start Partition for t at n, i.e., $TS_G(n,t)$)

1. Transform G by adding a new node n' to N such that
   - n' represents the same assignment as n: $\iota_{n'} = \iota_n$
   - n' has the same set of predecessors as n: $pred(n') = pred(n)$
   - n' has no successors: $succ(n') = \emptyset$

   Let $N' =_{df} N \cup \{n'\}$ and E' denote the set of resulting of nodes and edges, respectively.
2. Construct a regular expression ρ over N' representing the set of paths $\mathbf{P}[\mathbf{s}, n']$ (e.g., using the algorithm of Tarjan, 1981).

   (Note: "+" stands for non-deterministic branching, ";" for sequential composition, and "*" for indefinite looping).

Initial Partition: Carrier Set Computation (2)

3. Replace indefinite looping, *, by bounded looping, ^{k}, where k is the number of variables which occur on the left hand side of an assignment in the corresponding subexpression of ρ, to arrive at the ($*$-free) regular expression $\rho_{bounded}$.
4. Evaluate the functional $\Delta_\rho : \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T})$, which is inductively defined by

$$\Delta_\rho(T) =_{df} \begin{cases} \{\delta_{\iota_\rho}(s) \mid s \in T\} & \text{if } \rho \in E' \\ \Delta_{\rho_1}(\Delta_{\rho_2}(T)) & \text{if } \rho = \rho_1 ; \rho_2 \\ \Delta_{\rho_1}(T) \cup \Delta_{\rho_2}(T) & \text{if } \rho = \rho_1 + \rho_2 \\ \bigcup \{\Delta^j_{\rho_1}(T) \mid j \in \{1, \ldots, k\}\} & \text{if } \rho_1^k \end{cases}$$

for $\rho_{bounded}$ and $\{t\} \in \mathcal{P}(\mathbf{T})$, i.e., evaluate $\Delta_{\rho_{bounded}}(\{t\})$.

(Note: $\Delta^0_\rho =_{df} Id_{\mathcal{P}(\mathbf{T})}$ and $\Delta^j_\rho =_{df} \Delta^{j-1}_\rho \circ \Delta_\rho$, $j \geq 1$).

Initial Partition: Carrier Set Computation (3)

5. Finally set:

$$TS_G^{\sigma_s}(n,t) =_{df} \{t' \in \Delta_{\rho_{bounded}}(\{t\}) \,|\, \mathcal{E}(t')(\sigma_s) \neq \bot\} \cup$$
$$\{d \in \mathrm{ID} \,|\, \exists\, t' \in \Delta_{\rho_{bounded}}(\{t\}).\ \mathcal{E}(t')(\sigma_s) = d\}$$

Decidability of Finite Constants (1)

Theorem 5.7.6.3 (Decidability)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let $n \in N$ be a node, let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$ be an initial state, and let $t \in \mathbf{T}$ be a term. Then we have:

Algorithm 5.7.6.1 determines whether t is a finite constant at node n, i.e., whether

$$t \in \bigcup \{ C_{fin}^{\sigma_{\mathbf{s}}}(n, d) \mid d \in \mathrm{ID} \backslash \{\perp\} \}$$

In the positive case, Algorithm 5.7.6.1 determines additionally the value d of t at node n.

Decidability of Finite Constants (2)

Corollary 5.7.6.4 (Decidability of Finite Constants)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, let $n \in N$ be a node, let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$ be an initial state, and let $T_{finite} \subseteq \mathbf{T}$ be a finite set of terms. Then we have:

$$C_{fin}^{\sigma_{\mathbf{s}}}(n) \cap T_{finite}$$

is algorithmically decidable.

Note: The set of terms occuring in a program is finite. In particular, the set of terms occurring in an instruction at an edge are finite. Hence, the set of program terms, which are finite constants, can algorithmically be decided.

Chapter 5.7.7

Finite Constants: Specification

Finite Constants over $\mathbb{Z}$: DFA Specification

DFA Specification

- DFA lattice
  $$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (Part, \cap, \cup, \subseteq, \{\{t\} \mid t \in \mathbf{T}\}, \{\mathbf{T}\}) = \widehat{Part}$$
- DFA functional
  $[\![\ ]\!]_{fc} : E \to (Part \to Part)$ where
  $$\forall\, e \in E.\, \forall\, p \in Part.\, [\![\, e \,]\!]_{fc}(p) =_{df} \{(r,s) \mid (\delta_{\iota_e}(r), \delta_{\iota_e}(s)) \in p\}$$
- Initial information: $p_{fc} \in Part^{\sigma_s}_{\mathcal{E}}(T^{\sigma_s}_{fc})$ for $\sigma_{\mathbf{s}} \in \Sigma'_{Init}$ and $T^{\sigma_s}_{fc}$ finite $\subseteq \mathbf{T}$ computed using Algorithm 5.7.6.2.
- Direction of information flow: forward

Finite Constants Specification

- Specification: $\mathcal{S}^{fc}_{G} = (\widehat{Part}, [\![\ ]\!]_{fc}, p_{fc}, fw)$

Chapter 5.7.8

Termination, Safety, and Coincidence

Towards Safety and Termination

Lemma 5.7.8.1 (Descending Chain Condition)

The "*MaxFP* relevant" part $\widehat{Part}_{p_{fc}}$ of $\widehat{Part}$ satisfies the descending chain condition.

Note. The carrier set of the initial partition p_{fc} is finite.

Lemma 5.7.8.2 (Distributivity)

$[\![\ ]\!]_{fc}$ is distributive.

Corollary 5.7.8.3 (Monotonicity)

$[\![\ ]\!]_{fc}$ is monotonic.

Termination and Coincidence/Optimality

Theorem 5.7.8.4 (Termination)

Applied to $\mathcal{S}_G^{fc} = (\widehat{Part}, [\![\ ]\!]_{fc}, p_{fc}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{fc}$.

Proof. Immediately with Lemma 5.7.8.1, Lemma 5.7.8.3, Theorem 5.7.4.7, and Termination Theorem 3.4.4.

Theorem 5.7.8.5 (Coincidence/Optimality)

Applied to $\mathcal{S}_G^{fc} = (\widehat{Part}, [\![\ ]\!]_{fc}, p_{fc}, fw)$, Algorithm 3.4.3 is *MOP* optimal for $\mathcal{S}_G^{fc}$ (i.e., it terminates with the *MOP* solution of $\mathcal{S}_G^{fc}$).

Proof. Immediately with Lemma 5.7.8.2, Coincidence Theorem 3.5.2, and Termination Theorem 5.7.8.4.

Chapter 5.7.9

Soundness and Completeness

Soundness and Completeness of $MOP_{\mathcal{S}_G^{fc}}$

Theorem 5.7.9.1 (Soundness and Completeness)

The *MOP* solution of $\mathcal{S}_G^{fc}$ is

1. sound and complete for the term constant propagation problem CT, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ \ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) = MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{fc}}(n)$$

   if G is a flow graph with acyclic control flow.

2. sound but not complete for the term and variable constant propagation problems CT and CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.$$
$$C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} C^{\mathbf{V}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{fc}}(n)$$

   if G is a flow graph with arbitrary, possibly cyclic, control flow.

Soundness and Completeness $MaxFP_{\mathcal{S}_G^{fc}}$

Corollary 5.7.9.2 (Soundness and Completeness)

The *MaxFP* solution of $\mathcal{S}_G^{fc}$ is

1. sound and complete for the term constant propagation problem CT, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.\ C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) = MaxFP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{fc}}(n)$$

   if G is a flow graph with acyclic control flow.

2. sound but not complete for the term and variable constant propagation problems CT and CV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma'_{Init}.$$
$$C^{\mathbf{T}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} C^{\mathbf{V}}_{CT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma'} MaxFP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{fc}}(n)$$

   if G is a flow graph with arbitrary, possibly cyclic, control flow.

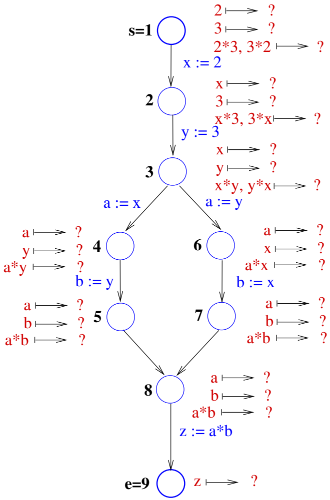
Chapter 5.7.10

Illustrating Example

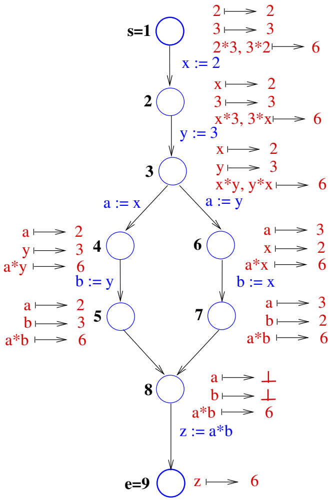
Finite Constants: Illustrating Example

...all terms are finite constants.

a)



b)



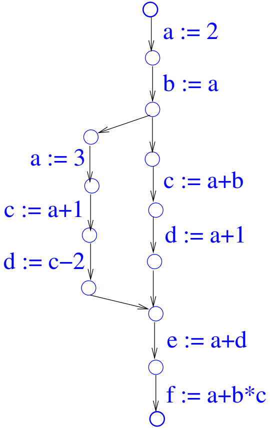
Chapter 5.8

Conditional Constants

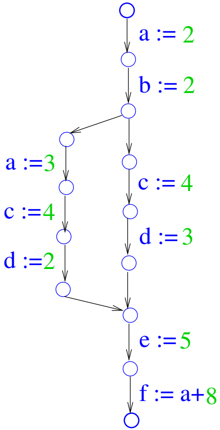
Motivating Example

Up to now: Branches were non-deterministically interpreted.

a)



b)



...unfortunately, a+8 is not a constant, if branches are non-deterministically interpreted.

As a Matter of Fact

In the preceding example

- a+8 is not a (non-deterministic) constant.

Consequently

- a+8 is neither a simple constant nor a Q constant nor a finite constant.

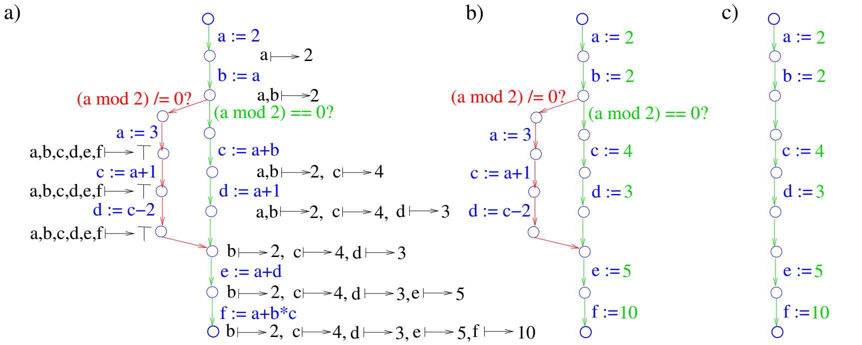
However

- a+8 could be a constant, if branching conditions were taken into account.

Note: Interpreting branches non-deterministically in DFA

- is done to avoid intricacies due to the undecidability of constant propagation,
- is counter-intuitive, however, for constant propagation itself.

Interpreting Branches Deterministically



...the term a+8 is a (deterministic) constant of value 10.

Conditional Constants

Conditional constants

- are an efficiently decidable subset of the set of deterministic constants.
- build on and generalize the notion of simple constants.
- allow the optimizations shown in Figure b) and in Figure c), when applied to the flow graph shown in Figure a) of the previous slide.

Chapter 5.8.1

Preliminaries, Problem Definition

From Simple to Conditional Constants

What needs to be extended and adapted?

- the notion of terms: Relators, logical constants and operators
- the notion of interpretation: Relators, logical constants and operators
- the semantics of terms: Boolean terms
- The semantics of instructions: Conditionals
- The (basic) DFA lattice for constant propagation: Truth values

We do not need to extend

- the notion of states

since we stay with arithmetical variables and do not introduce Boolean variables.

Introducing Relators and Logical Operators

We introduce

- **LC** $=_{df} \{true, false\}$: the set of logical (or Boolean) constant symbols,
- **R** $=_{df} \{==, /=, <, >, <=, >=, \ldots\}$: a set of binary relator symbols (or relators),
- **LO** $=_{df} \{\wedge, \vee, \neg\}$: the set of logical operator symbols.

We assume that

- **V**, **C**, **O** (for arithmetical terms), **LC**, **R**, and **LO** (for Boolean terms) are all pairwise disjoint.

Introducing Boolean Terms

Definition 5.8.1.1 (Boolean Terms)

1. Every constant symbol $b \in \mathbf{LC}$ is a Boolean term.
2. If $rel \in \mathbf{R}$ is a binary relator and t_1 and t_2 are (arithmetical) terms, then $t_1 \ rel \ t_2$ is a Boolean term.
3. if b_1 and b_2 are Boolean terms, then $b_1 \wedge b_2$, $b_1 \vee b_2$, and $\neg b_1$ are Boolean terms.
4. There are no Boolean terms other than those which can be constructed by means of the above three rules.

We denote the set of all Boolean terms by $\mathbf{T_B}$.

Arithmetical and Boolean Terms

We denote by

- $\mathbf{T_{AB}} =_{df} \mathbf{T_A} \cup \mathbf{T_B}$ the set of all terms

where

- $\mathbf{T_A}$ denotes the set of arithmetical terms
- $\mathbf{T_B}$ denotes the set of Boolean terms.

Note: $\mathbf{T_A}$ equals $\mathbf{T}$ as introduced and used in the previous sections of Chapter 5.

Towards the Semantics of Boolean Terms

We need to extend

- the data domain $\mathbb{D}$ by adding the set of Boolean truth values $\mathbb{B} =_{df} \{True, False\}$,
- the interpretation from the (arithmetical) constant and operator symbols over $\mathbb{D}$ to the Boolean constant symbols, relators, and logical operator symbols over $\mathbb{B}$.

Extending the Data Domain

Let

- $\mathbb{D}_{\mathbb{B}} =_{df} \mathbb{D} \mathbin{\dot\cup} \mathbb{B}$ be the extended new data domain of interest.

  (with $\mathbb{D}$ e.g., the set of natural numbers $\mathbb{N}$, the set of integers $\mathbb{Z}$, etc.).

As before, we call the

- elements of $\mathbb{D}_{\mathbb{B}}$ (data) values.

Moreover, we assume that

- $\mathbb{D}_{\mathbb{B}}$ includes a distinguished element $\perp$ representing the value *undefined* and a distinguished element $\top$ representing "universal" information.

Interpreting Relators and Logical Operators

Definition 5.8.1.2 (Extended Interpretation)

An interpretation $I =_{df} (\, \mathbb{ID}_{\mathbb{B}}, I_0 \,)$ of $\mathbf{C}$, $\mathbf{O}$, $\mathbf{LC}$, $\mathbf{R}$, and $\mathbf{LO}$ is a pair, where I_0 is a function, which maps every

- every constant symbol $c \in \mathbf{C}$ to a datum $I_0(c) \in \mathbb{ID}$,
- every k-ary operator symbol $op \in \mathbf{O}$ to a total strict function $I_0(op) : \mathbb{ID}^k \to \mathbb{ID}$,
- the logical constant symbols *true* and *false* to the Boolean constants *True* and *False*, respectively,
- the relator symbols $rel \in \mathbf{R} =_{df} \{==, /=, <, >, \ldots\}$ to the strict relations equal, not equal, less, etc., on $\mathbb{ID} \times \mathbb{ID}$,
- the logical operator symbols $\wedge$, $\vee$, and $\neg$ to the strict logical operations and, or, and not on $\mathbb{B} \times \mathbb{B}$ and $\mathbb{B}$.

and satisfies $I_0(o)(d_1, \ldots, d_k) = \top$, $o \in \mathbf{O} \cup \mathbf{R} \cup \mathbf{LO}$, if $(\forall\, i \in \{1, \ldots, k\}.\ d_i \neq \bot) \ \wedge\ (\exists\, i \in \{1, \ldots, k\}.\ d_i = \top)$.

States, Initial States

We do not introduce Boolean variables. The notion of states thus remains essentially unchanged.

Definition 5.8.1.3 (States, Initial States over ID)

- A state $\sigma : \mathbf{V} \to \mathrm{ID}$ is a total mapping, which maps every (arithmetical) variable to a data value $d \in \mathrm{ID}$.

  We denote the set of all states by

  $$\Sigma =_{df} \{\, \sigma \,|\, \sigma : \mathbf{V} \to \mathrm{ID} \,\}$$

- $\sigma_\perp$ and $\sigma_\top$ denote two distinguished states of Σ defined by

  $$\forall\, v \in \mathbf{V}.\ \sigma_\perp(v) = \perp,\ \sigma_\top(v) = \top$$

  respectively.
- $\Sigma_{Init} =_{df} \{\, \sigma \in \Sigma \,|\, \forall\, v \in \mathbf{V}.\ \sigma(v) \neq \top \,\}$ denotes the set of initial states.

Semantics of Terms

Definition 5.8.1.4 (Semantics of Terms)

The semantics of terms $t \in \mathbf{T_{AB}}$ is defined by the evaluation function

$$\mathcal{E} : \mathbf{T_{AB}} \to (\Sigma \to \mathbb{D}_{\mathbb{B}})$$

defined by

$$\forall t \in \mathbf{T_{AB}}\ \forall \sigma \in \Sigma.\ \mathcal{E}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t \equiv x \in \mathbf{V} \\ I_0(c) & \text{if } t \equiv c \in \mathbf{C} \cup \mathbf{LC} \\ I_0(o)(\mathcal{E}(t_1)(\sigma), \ldots, \mathcal{E}(t_k)(\sigma)) & \\ & \text{if } t \equiv (o, t_1, \ldots, t_k), \\ & \quad o \in \mathbf{O} \cup \mathbf{R} \cup \mathbf{LO} \end{cases}$$

Semantics of Instructions

Definition 5.8.1.5 (Semantics of Instructions)

- Let $\iota \equiv x := t$, $t \in \mathbf{T_A}$, be an assignment instruction. The semantics of ι is defined by the state transformation function (or state transformer) $\theta_\iota : \Sigma \to \Sigma$ defined by $\theta_\iota(\sigma_\top) =_{df} \sigma_\top$ and
$$\forall\, \sigma \in \Sigma \backslash \{\sigma_\top\}\ \forall\, y \in \mathbf{V}.\ \theta_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$
- Let $\iota \equiv$ *skip* be the empty instruction. The semantics of ι is defined by the identical state transformation function (or state transformer) Id_Σ, i.e., $\theta_\iota =_{df} Id_\Sigma$.
- Let $\iota \equiv t$, $t \in \mathbf{T_B}$, be a conditional expression. The semantics of ι is defined by $\theta_\iota(\sigma_\top) =_{df} \sigma_\top$ and
$$\forall\, \sigma \in \Sigma \backslash \{\sigma_\top\}.\ \theta_\iota(\sigma)(y) =_{df} \begin{cases} \sigma_\top & \text{if } \mathcal{E}(t)(\sigma) \in \{\mathit{False}, \top\} \\ \sigma & \text{otherwise} \end{cases}$$

State Transformer Lemma for $\sigma_\top$

Lemma 5.8.1.6 (State Transformer Lemma for $\sigma_\top$)

Let ι be an assignment instruction, the empty statement, or a conditional expression. Then we have:

$$\theta_\iota(\sigma_\top) = \sigma_\top$$

Remarks on the Impact of Lemma 5.8.1.6 (1)

- Lemma 5.8.1.6 ensures that the distinguished state $\sigma_\top$ is left invariant by every state transformer.
- This guarantees that the state transformers of conditional expressions act as filters that prevent propagating information alongside branches whose guarding conditional expressions are known to be violated (i.e., $\mathcal{E}(t)(\sigma) = \mathit{False}$) or can not yet be evaluated (i.e., $\mathcal{E}(t)(\sigma) = \top$).
- Conversely, the filters let information pass and propagate it further if the values of the guarding conditional expressions are known to be satisfied (i.e., $\mathcal{E}(t)(\sigma) = \mathit{True}$) or dubious (i.e., $\mathcal{E}(t)(\sigma) = \bot$).
- Overall, this causes the semantics to be deterministic and the fixed point analysis based on it to behave deterministically, too.

Remarks on the Impact of Lemma 5.8.1.6 (2)

Note

- The case "can not yet be evaluated (i.e., $\theta_\iota(\sigma)(y) = \top$)" can only occur in the actual fixed point program analysis by picking an edge carrying a conditional expression "too early" from the workset.

  The case can not occur and is irrelevant for the pathwise characterization of deterministic constants (the general problem, Chapter 5.8.1) and conditional constants (the computed class of constants, Chapter 5.8.3).

Extending State Transformers to Paths

Nothing has to be changed. For convenience, we recall:

Definition 5.2.6 (Extending θ from Edges to Paths) – recalled

The state transformers θ_{ι_e}, $e \in E$, are extended onto paths $p = \langle e_1, e_2, \ldots, e_q \rangle$ in G by defining:

$$\theta_p =_{df} \begin{cases} Id_\Sigma & \text{if } q < 1 \\ \theta_{\langle e_2, \ldots, e_q \rangle} \circ \theta_{\iota_{e_1}} & \text{otherwise} \end{cases}$$

...where $G = (N, E, \mathbf{s}, \mathbf{e})$ denotes the flow graph of interest, and ι_e the instruction at edge e, $e \in E$.

Semantics of Programs: Det. Collecting Sem.

Defining the deterministic collecting semantics requires (only) to take care of and remove the special state $\sigma_\top$:

Definition 5.8.1.7 (Deterministic Collecting Sem.)

- The deterministic collecting semantics of G is defined by:

$$\mathcal{DCS}_G : \Sigma_{Init} \to N \to \mathcal{P}(\Sigma)$$

$$\forall\, n \in N.\, \forall\, \sigma \in \Sigma_{Init}.\, \mathcal{DCS}_G(n) =_{df} \{\, \theta_p(\sigma) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\} \setminus \{\sigma_\top\}$$

- The deterministic collecting semantics of G with respect to a fixed initial state $\sigma_\mathbf{s} \in \Sigma_{Init}$ is defined by

$$\mathcal{DCS}_G^{\sigma_\mathbf{s}} : N \to \mathcal{P}(\Sigma)$$

$$\forall\, n \in N.\, \mathcal{DCS}_G^{\sigma_\mathbf{s}}(n) =_{df} \{\, \theta_p(\sigma_\mathbf{s}) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\} \setminus \{\sigma_\top\}$$

Unreachable Nodes

Note

- If $\mathcal{DCS}_G^{\sigma_s}(n) = \emptyset$ for some node $n \in N$, this means that node n is not reachable, when branching conditions are deterministically interpreted.

Deterministic Constants

Let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$ be an initial state, let $t \in \mathbf{T_{AB}}$ be a term, and let $d \in \mathrm{ID_{IB}}$ be a data value with $d \notin \{\perp, \top\}$.

Definition 5.8.1.8 (Deterministic Constant)

t is a deterministic constant at node n for $\sigma_{\mathbf{s}}$, i.e., the value of t at node n, $n \in N$, is a constant, if

$$\forall\, \sigma, \sigma' \in \mathcal{DCS}_G^{\sigma_{\mathbf{s}}}(n).\ \mathcal{E}(t)(\sigma) = \mathcal{E}(t)(\sigma') \neq \perp$$

Definition 5.8.1.9 (Det. Constant of Value d)

t is a deterministic constant of value d for $\sigma_{\mathbf{s}}$ at node n, $n \in N$, if

$$\{\mathcal{E}(t)(\sigma) \mid \sigma \in \mathcal{DCS}_G^{\sigma_{\mathbf{s}}}(n)\} = \{d\}$$

Deterministic Constant Terms & Variables (1)

Let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$ be an initial state, let $d \in \mathrm{ID}_{\mathrm{IB}} \backslash \{\perp, \top\}$ be a data value, and let n be a node of G.

Definition 5.8.1.10 (Det. Const. Terms & Variables)

The set of terms and variables being (deterministic) constants of some value at n are given by the sets:

- $DCT_G^{\sigma_{\mathbf{s}}}(n) =_{df}$
  $$\{(t, d) \in \mathbf{T_{AB}} \times \mathrm{ID} \mid t \text{ is a deterministic constant of value } d \text{ for } \sigma_{\mathbf{s}} \text{ at } n\}$$
- $DCV_G^{\sigma_{\mathbf{s}}}(n) =_{df}$
  $$\{(v, d) \in \mathbf{V} \times \mathrm{ID} \mid v \text{ is a deterministic constant of value } d \text{ for } \sigma_{\mathbf{s}} \text{ at } n\}$$

Deterministic Constant Terms & Variables (2)

The sets $DCT_G^{\sigma_\mathbf{s}}$ and $DCV_G^{\sigma_\mathbf{s}}$ induce (state-like) functions $DC^{\mathbf{T_{AB}}}_{DCT_G^{\sigma_\mathbf{s}}}$ and $DC^{\mathbf{V}}_{DCV_G^{\sigma_\mathbf{s}}}$, $DC^{\mathbf{T_{AB}}}_{DCV_G^{\sigma_\mathbf{s}}}$, respectively, which we use alternatively to the sets $DCT_G^{\sigma_\mathbf{s}}$ and $DCV_G^{\sigma_\mathbf{s}}$:

- $DC^{\mathbf{T_{AB}}}_{DCT_G^{\sigma_\mathbf{s}}} : N \to \mathbf{T_{AB}} \to \mathrm{ID}_{\mathrm{IB}}$ defined by

$$DC^{\mathbf{T_{AB}}}_{DCT_G^{\sigma_\mathbf{s}}}(n)(t) =_{df} \begin{cases} d & \text{if } (t,d) \in DCT_G^{\sigma_\mathbf{s}}(n) \\ \perp & \text{otherwise} \end{cases}$$

- $DC^{\mathbf{V}}_{DCV_G^{\sigma_\mathbf{s}}} : N \to \mathbf{V} \to \mathrm{ID}$ defined by

$$DC^{\mathbf{V}}_{DCV_G^{\sigma_\mathbf{s}}}(n)(v) =_{df} \begin{cases} d & \text{if } (v,d) \in DCV_G^{\sigma_\mathbf{s}}(n) \\ \perp & \text{otherwise} \end{cases}$$

  which itself induces the (state-like) function on terms

  $DC^{\mathbf{T_{AB}}}_{DCV_G^{\sigma_\mathbf{s}}} : N \to \mathbf{T_{AB}} \to \mathrm{ID}_{\mathrm{IB}}$ defined by

$$DC^{\mathbf{T_{AB}}}_{DCV_G^{\sigma_\mathbf{s}}}(n)(t) = \mathcal{E}(t)(DC^{\mathbf{V}}_{DCV_G^{\sigma_\mathbf{s}}}(n))$$

Deterministic Constant Terms & Variables (3)

We have:

Lemma 5.8.1.11 (Equivalence)

$\forall\, n \in N\ \forall\, t \in \mathbf{T_{AB}}\ \forall\, v \in \mathbf{V}\ \forall\, d \in \mathbb{D}.$

- $(t, d) \in DCT_G^{\sigma_{\mathbf{s}}}(n)$ iff $DC^{\mathbf{T_{AB}}}_{DCT_G^{\sigma_{\mathbf{s}}}}(n)(t) = d$
- $(v, d) \in DCV_G^{\sigma_{\mathbf{s}}}(n)$ iff $DC^{\mathbf{V}}_{DCV_G^{\sigma_{\mathbf{s}}}}(n)(v) = d$

Deterministic Constant Propagation Problem

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph, and let $\sigma_{\mathbf{s}} \in \Sigma_{Init}$ be an initial state.

The deterministic constant propagation problems for terms (DCT) and variables (DCV) are defined by:

Definition 5.8.1.12 (Det. CP Problem for Terms)

The deterministic term constant propagation problem, DCT, is to determine for every node $n \in N$ of G the set $DCT_G^{\sigma_{\mathbf{s}}}(n)$.

Definition 5.8.1.13 (Det. CP Problem for Variables)

The deterministic variable constant propagation problem, DCV, is to determine for every node $n \in N$ of G the set $DCV_G^{\sigma_{\mathbf{s}}}(n)$.

Equival't Charact'ion of the Det. CP Problem

The Equivalence Lemma 5.8.1.11 yields:

Lemma 5.8.1.14 (Problem Equivalence)

Solving the deterministic

- term constant propagation problem DCT
- variable constant propagation problem DCV

is equivalent to computing the (state-like) functions

- $DC^{\mathbf{T_{AB}}}_{DCT^{\sigma_{\mathbf{s}}}_{G}} : N \to \mathbf{T_{AB}} \to \mathbb{ID}_{\mathbb{B}}$
- $DC^{\mathbf{V}}_{DCV^{\sigma_{\mathbf{s}}}_{G}} : N \to \mathbf{V} \to \mathbb{ID}$

respectively.

Det. CP Algorithms: Soundness, Completeness

Let A be a deterministic constant propagation algorithm for DCT (DCV), and let $A_{DCT_G^{\sigma_s}}(n) \subseteq \mathbf{T_{AB}} \times \mathsf{ID}_{\mathsf{IB}}$ and $A_{DCV_G^{\sigma_s}}(n) \subseteq \mathbf{V} \times \mathsf{ID}$ denote the sets of terms and variables discovered by A to be constant at node n, respectively.

Definition 5.8.1.15 (Soundness of DCP Algorithms)

A is sound for DCT (DCV) if

$$\forall\, n \in N.\ \forall\, \sigma_s \in \Sigma_{Init}.\ DCT_G^{\sigma_s}(n) \supseteq A_{DCT_G^{\sigma_s}}(n)$$

$$(\, DCV_G^{\sigma_s}(n) \supseteq A_{DCV_G^{\sigma_s}}(n)\,)$$

Definition 5.8.1.16 (Completeness of DCP Alg's)

A is complete for DCT (DCV) if

$$\forall\, n \in N.\ \forall\, \sigma_s \in \Sigma_{Init}.\ DCT_G^{\sigma_s}(n) \subseteq A_{DCT_G^{\sigma_s}}(n)$$

$$(\, DCV_G^{\sigma_s}(n) \subseteq A_{DCV_G^{\sigma_s}}(n)\,)$$

DCP Algorithms: Conservativity, Optimality

Definition 5.8.1.17 (Conservativity of DCP Alg's)

A deterministic constant propagation algorithm A is conservative for DCT (DCV), if it is sound for DCT (DCV), i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma_{Init}.\ DCT_G^{\sigma_{\mathbf{s}}}(n) \supseteq A_{DCT_G^{\sigma_{\mathbf{s}}}}(n)$$

$$(\, DCV_G^{\sigma_{\mathbf{s}}}(n) \supseteq A_{DCV_G^{\sigma_{\mathbf{s}}}}(n)\,)$$

Definition 5.8.1.18 (Optimality of DCP Algorithms)

A deterministic constant propagation algorithm A is optimal for DCT (DCV), if it is sound and complete for DCT (DCV), i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma_{Init}.\ DCT_G^{\sigma_{\mathbf{s}}}(n) = A_{DCT_G^{\sigma_{\mathbf{s}}}}(n)$$

$$(\, DCV_G^{\sigma_{\mathbf{s}}}(n) = A_{DCV_G^{\sigma_{\mathbf{s}}}}(n)\,)$$

Relating $DCV_G^{\sigma_s}$ and $DCT_G^{\sigma_s}$ (1)

Note:

The solution $DCV_G^{\sigma_s}$ of the DCV constant propagation problem induces for every node n of G a state $\sigma^n_{DCV_G^{\sigma_s}} \in \Sigma$ defined by

$$\forall\, n \in N.\, \forall\, v \in \mathbf{V}.\; \sigma^n_{DCV_G^{\sigma_s}}(v) =_{df} \begin{cases} d & \text{if } (v,d) \in DCV_G^{\sigma_s}(n) \\ \perp & \text{otherwise} \end{cases}$$

Then, the states $\sigma^n_{DCV_G^{\sigma_s}}$ induce a solution $DCT_{DCV_G^{\sigma_s}}$ for the DCT constant propagation problem:

$$DCT_{DCV_G^{\sigma_s}}(n) =_{df} \{(t,d) \in \mathbf{T} \times \mathsf{ID_{IB}} \mid \mathcal{E}(t)(\sigma^n_{DCV_G^{\sigma_s}}) = d \neq \perp\}$$

Relating $DCV_G^{\sigma_s}$ and $DCT_G^{\sigma_s}$ (2)

We have:

Lemma 5.8.1.19 (Approximation Lemma)

$$\forall\, n \in N.\ \forall\, \sigma_s \in \Sigma_{Init}.\ DCT_G^{\sigma_s}(n) \supseteq DCT_{DCV_G^{\sigma_s}}(n)$$

In general, this inclusion is a proper inclusion.

Interpretation, Conclusions (1)

Intuitively

- The Approximation Lemma 5.8.1.19 states that a term can be a deterministic constant at some node n without that all of its variables are (deterministic) constants at n.

Hence

- Any sound algorithm for the DCV constant propagation problem is in general conservative and suboptimal for the DCT constant propagation problem.
- This holds even for a (hypothetical) optimal algorithm (cf. Theorem 5.1.1) for the DCV constant propagation problem.

Interpretation, Conclusions (2)

As a matter of fact

- The Undecidability Theorem 5.1.1 rules out the possibility and existence of DCT and DCV optimal constant propagation algorithms.

Hence

- The best we can hope for are conservative DCT and DCV constant propagation algorithms trading optimality for decidability (and efficiency, scalability).

Conditional Constants

The algorithm for conditional constants $\mathcal{A}_{CC}$ is a

- singleton DCV algorithm computing a variable valuation function

$$\mathcal{A}_{CC_{DCV}} : N \to \mathbf{V} \to \mathrm{ID}$$

as the result of the analysis.

Induced Term Valuation Function of $\mathcal{A}_{CC}$

The variable valuation function

- $\mathcal{A}_{CC_{DCV}} : N \to \mathbf{V} \to \mathbb{D}$

of the conditional constants algorithm $\mathcal{A}_{CC}$ induces a term valuation function

- $\mathcal{A}^{\mathbf{T_{AB}}}_{CC_{DCV}} : N \to \mathbf{T_{AB}} \to \mathbb{D}_{\mathbb{B}}$

defined by

$$\forall\, n \in N\ \forall\, t \in \mathbf{T_{AB}}.\ \mathcal{A}_{CC_{DCV}}(n)(t) =_{df}$$

$$\begin{cases} d & \text{if } \mathcal{E}(t)(\mathcal{A}_{CC_{DCV}}(n)) = d \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

DCV/DCT Solutions induced by $\mathcal{A}_{CC}$

The variable and term valuation functions

- $\mathcal{A}_{CC_{DCV}} : N \to \mathbf{V} \to \mathsf{ID}$
- $\mathcal{A}^{\mathbf{T_{AB}}}_{CC_{DCV}} : N \to \mathbf{T_{AB}} \to \mathsf{ID}_{\mathsf{IB}}$

of $\mathcal{A}_{CC}$ induce solutions for the DCV and the DCT constant propagation problems:

- $DCV_{\mathcal{A}_{CC_{DCV}}}(n) =_{df}$
$$\{(v,d) \in \mathbf{V} \times \mathsf{ID} \mid \mathcal{A}_{CC_{DCV}}(n)(v) = d \neq \perp\}$$
- $DCT_{\mathcal{A}_{CC_{DCV}}}(n) =_{df}$
$$\{(t,d) \in \mathbf{T_{AB}} \times \mathsf{ID}_{\mathsf{IB}} \mid \mathcal{A}^{\mathbf{T_{AB}}}_{CC_{DCV}}(n)(t) = d \neq \perp\}$$

Chapter 5.8.2

DFA States, DFA Lattice

From Data Domains to DFA Lattices

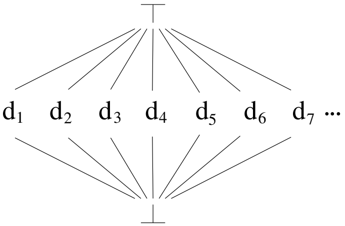
Given

- the data domain $\mathbb{D}_{\mathbb{B}} =_{df} \mathbb{D} \,\dot{\cup}\, \mathbb{B}$ of interest with $\mathbb{D}$ e.g., the set of natural numbers $\mathbb{N}$, the set of integers $\mathbb{Z}$, etc., including a distinguished element $\perp$ representing the value *undefined* and a distinguished element $\top$ representing "universal" information.

...it remains to arrange the elements of data domain $\mathbb{D}_{\mathbb{B}}$ to a flat DFA lattice (as shown next).

From Data Domains to DFA Lattices (2)

Given the data domain $\mathbb{ID}_{\mathbb{B}}$, the flat lattice $\mathcal{FL}_{\mathbb{ID}_{\mathbb{B}}}$
(cf. Appen- dix A.4)



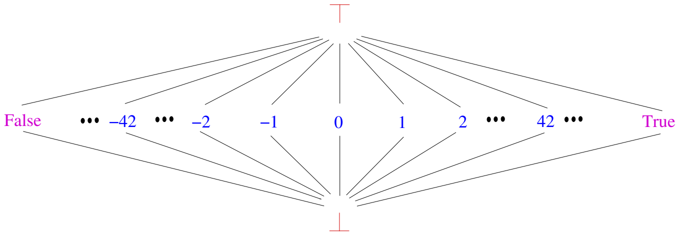
...constitutes the basic DFA lattice for conditional constant propagation.

Intuitively

- $\top$ represents complete but inconsistent information.
- d_i, $i \geq 1$, represents precise information.
- $\perp$ represents no information, the empty information.

The Basic DFA Lattice over $\mathbb{Z}$ and $\mathbb{B}$

...is given by $\mathcal{FL}_{\mathbf{Z}_\mathbb{B}}$



...leading to the class of conditional constants over $\mathbb{Z}$ and $\mathbb{B}$.

DFA States, Semantics of Instructions, etc.

Unlike to Chapter 5.3.1, we do not need to extend the notions of

- interpretation of terms
- states and initial states
- semantics of terms
- semantics of instructions

to cope with the distinguished element $\top$ for the constant propagation analysis. This has been done to full extent in the corresponding definitions of Chapter 5.8.1, which can now directly be used for defining the conditional constants propagation analysis.

The DFA Lattice for Conditional Constants

The set of (DFA) states Σ together with the pointwise ordering of states, $\sqsubseteq_\Sigma$, constitutes a complete lattice (cf. Appendix A.4):

$$\forall\, \sigma, \sigma' \in \Sigma.\ \sigma \sqsubseteq_\Sigma \sigma' \ \text{ iff } \ \forall\, v \in \mathbf{V}.\ \sigma(v) \sqsubseteq_{\mathcal{FL}_{\mathbb{D}_\mathbb{B}}} \sigma'(v)$$

Lemma 5.8.2.1 (Lattice of DFA States)

$\widehat{\Sigma} =_{df} (\Sigma, \sqcap_\Sigma, \sqcup_\Sigma, \sqsubseteq_\Sigma, \sigma_\perp, \sigma_\top)$ is a complete lattice with

- least element $\sigma_\perp$, greatest element $\sigma_\top$,
- pointwise meet $\sqcap_\Sigma$ and join $\sqcup_\Sigma$ as meet and join operation, respectively.

Chapter 5.8.3

Conditional Constants: Specification

Conditional Constants over $\mathbb{Z}_{\mathbb{B}}$: DFA Specific.

DFA Specification

- DFA lattice
  $$\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\Sigma, \sqcap_\Sigma, \sqcup_\Sigma, \sqsubseteq_\Sigma, \sigma_\bot, \sigma_\top) = \widehat{\Sigma}$$
  with Σ set of DFA states over $\mathbb{Z}_{\mathbb{B}}$.
- DFA functional
  $[\![\ ]\!]_{cc} : E \to (\Sigma \to \Sigma)$ where $\forall\, e \in E.\ [\![\, e\,]\!]_{cc} =_{df} \theta_{\iota_e}$
- Initial information: $\sigma_{\mathbf{s}} \in \Sigma_{Init}$
- Direction of information flow: forward

Conditional Constants Specification

- Specification: $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, [\![\ ]\!]_{cc}, \sigma_{\mathbf{s}}, fw)$

Chapter 5.8.4

Termination, Safety, and Coincidence

Towards Safety and Termination

Lemma 5.8.4.1 (Descending Chain Condition)

$\widehat{\Sigma}$ satisfies the descending chain condition.

Note. The set of variables occurring in a program is finite.

Lemma 5.8.4.2 (Monotonicity)

$[\![\ ]\!]_{cc}$ is monotonic.

Lemma 5.8.4.3 (Non-Distributivity)

$[\![\ ]\!]_{cc}$ is not distributive.

Termination and Safety/Conservativity

Theorem 5.8.4.4 (Termination)

Applied to $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, [\![\ ]\!]_{cc}, \sigma_{\mathsf{s}}, fw)$, Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_G^{cc}$.

Proof. Immediately with Lemma 5.8.4.1, Lemma 5.8.4.2, and Termination Theorem 3.4.4.

Theorem 5.8.4.5 (Safety/Conservativity)

Applied to $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, [\![\ ]\!]_{cc}, \sigma_{\mathsf{s}}, fw)$, Algorithm 3.4.3 is *MOP* conservative for $\mathcal{S}_G^{cc}$ (i.e., it terminates with a lower approximation of the *MOP* solution of $\mathcal{S}_G^{cc}$).

Proof. Immediately with Lemma 5.8.4.2, Safety Theorem 3.5.1, and Termination Theorem 5.8.4.4.

Non-Coincidence

Theorem 5.8.4.6 (Non-Coincidence/Non-Opt.)

Applied to $\mathcal{S}_G^{cc} = (\widehat{\Sigma}, [\![\ ]\!]_{cc}, \sigma_{\mathbf{s}}, fw)$, Algorithm 3.4.3 is in general not *MOP* optimal for $\mathcal{S}_G^{cc}$ (i.e., it terminates with a properly lower approximation of the *MOP* solution of $\mathcal{S}_G^{cc}$).

Proof. Immediately with Lemma 5.8.4.3, Coincidence Theorem 3.5.2, and Termination Theorem 5.8.4.4.

Corollary 5.8.4.7 (Safety, Non-Coincidence)

The *MaxFP* solution for $\mathcal{S}_G^{cc}$, is always a safe approximation of the *MOP* solution of $\mathcal{S}_G^{cc}$. In general, the *MOP* solution and the *MaxFP* solution of $\mathcal{S}_G^{cc}$ do not coincide.

Chapter 5.8.5

Soundness and Completeness

Soundness and Completeness of $MOP_{\mathcal{S}_G^{cc}}$

Theorem 5.8.5.1 (Soundness and Completeness)

The *MOP* solution of $\mathcal{S}_G^{cc}$ is

1. sound and complete for the variable constant propagation problem DCV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma_{Init}.\ DC^{\mathbf{V}}_{DCV_G^{\sigma_{\mathbf{s}}}}(n) = MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{cc}}(n)$$

2. sound but not complete for the term constant propagation problem DCT, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma_{Init}.\ DC^{\mathbf{T}}_{DCT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma} {MOP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{cc}}}^{\mathbf{T}}(n)$$

In general, the inclusion is a proper inclusion.

Soundness and Completeness of $MaxFP_{\mathcal{S}_G^{cc}}$

Corollary 5.8.5.2 (Soundness and Completeness)

The *MaxFP* solution of $\mathcal{S}_G^{cc}$ is

1. sound but not complete for DCV, i.e.

$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma_{Init}.\ DC^{\mathbf{V}}_{DCV_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma} MaxFP^{\sigma_{\mathbf{s}}}_{\mathcal{S}_G^{cc}}(n)$$

2. sound but not complete for DCT, i.e.

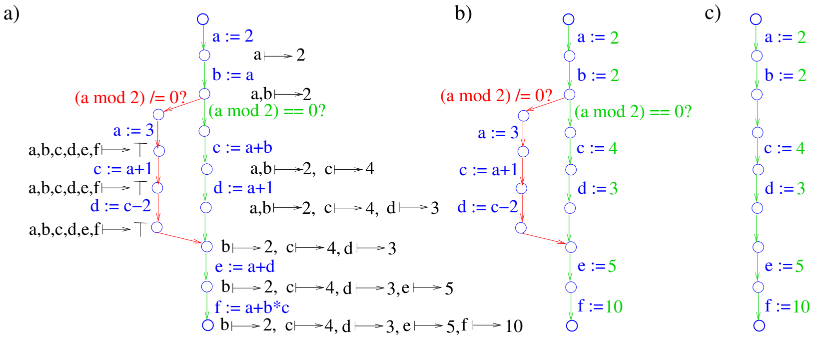
$$\forall\, n \in N.\ \forall\, \sigma_{\mathbf{s}} \in \Sigma_{Init}.\ DC^{\mathbf{T}}_{DCT_G^{\sigma_{\mathbf{s}}}}(n) \sqsupseteq_{\Sigma} MaxFP^{\sigma_{\mathbf{s}}\,\mathbf{T}}_{\mathcal{S}_G^{cc}}(n)$$

In general, both inclusions are proper inclusions.

Chapter 5.8.6

Illustrating Example

Conditional Constants over $\mathbb{Z}_{\mathbb{B}}$: Illustrating Example



...all terms are conditional constants.

Remark

The original algorithm of Wegman and Zadeck for conditional constants is technically different and makes use of

- executable flags for nodes/edges

in order to filter information for propagation.

Going Beyond Conditional Constants

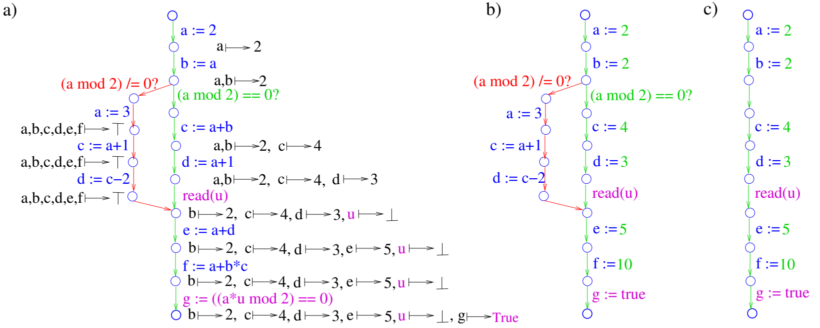
...is possible.

Conditional expressions like

- `x==1?` can be treated like an assignment on the “true” branch, even if their truth value can not be decided at analysis time.
- `x>0?` can also be propagated along the “true” branch and beneficially be exploited for evaluating other program terms; similarly, this holds for the negation of this expression `x<=0?` along the “false” branch.
- `(a mod 2)==0?` can also be progagated along the “true” branch and beneficially be exploited for evaluating other program terms.
- ...

Beyond Conditional Constants over $\mathbb{Z}_{\mathbb{B}}$

Heuristic extensions of the conditional constants analysis as sketched before would allow to detect that the (Boolean) variable g is a constant of value True, even though the value of variable u can not be figured out at compile time.



Chapter 5.9

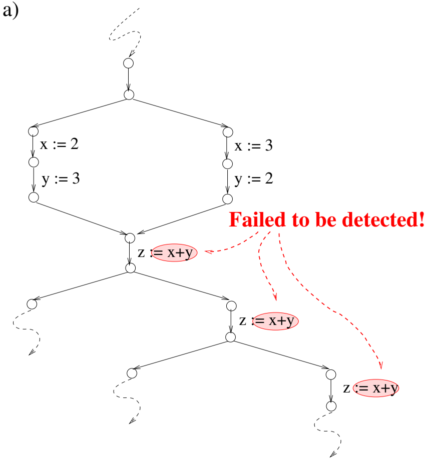
VG Constants

Chapter 5.9.1

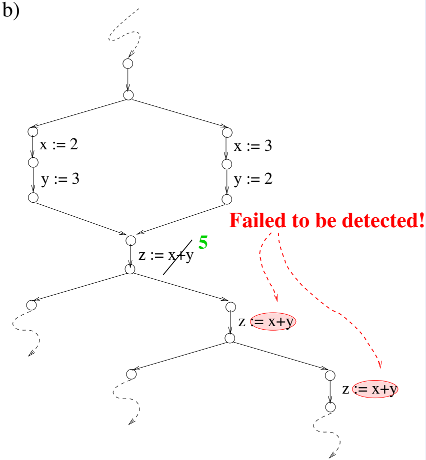
Motivation

Simple Constants and Q Constants

Recalling the limitations of simple and Q constants:



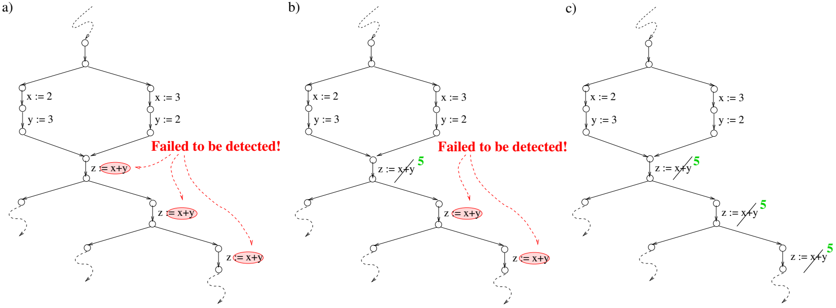
After simple constant propagation
(No effect at all!)



After Q constants propagation

The Look-Ahead Challenge

...there is a need for a look-ahead of unlimited length:



After simple constant propagation
(Note: No effect at all!)

After Q constants propagation

After VG_{Φ} constants propagation

Constant Propagation on the Value Graph (1)

In this chapter, we are going to present

- the VG_ϕ algorithm addressing this challenge.

As the algorithm for finite constants, the VG_ϕ algorithm

- extends the look-ahead of 1 heuristics of the Q approach systematically.

Compared to the algorithm for finite constants, however,

- the VG_ϕ algorithm balances analysis power and computational complexity differently giving more weight to performance .

Constant Propagation on the Value Graph (2)

...comes in two variants:

- The VG_{sc} Approach: The basic algorithm

  ...computes simple constants.

- The VG_{ϕ} Approach: The full algorithm

  ...mimicks the look-ahead heuristics for a finite but long range analysis.

Constant Propagation on the Value Graph (3)

Technically, both the basic VG_{sc} and the full VG_{ϕ} algorithm

- work on the value graph of Alpern, Wegman, and Zadeck (POPL'88) of a program that is derived from the static single assignment (SSA) representation of a program.

and proceed in 5 steps:

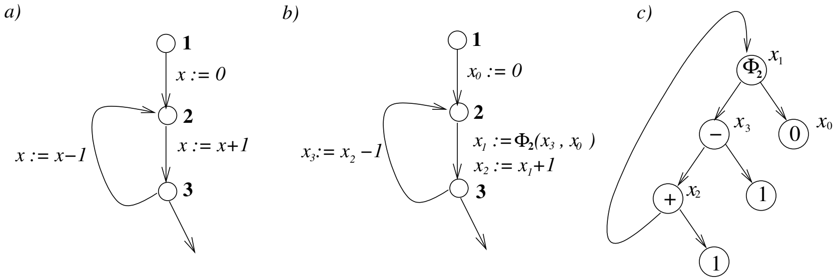
- Construct the static single assignment form (SSA) form G_{ssa} of a program G.
- Construct the value graph (VG) of G_{ssa}.
- Analyse the value graph to detect constant terms.
- Apply the analysis results to optimize the SSA form of G_{ssa} of G, obtaining $G_{ssa_{opt}}$.
- Construct the optimized flow graph G_{opt} from $G_{ssa_{opt}}$.

Chapter 5.9.2

VG$_{sc}$ Constants: The Basic Approach

The Running Example

...constructing the SSA form and the VG of a program:



Original Flow Graph SSA Form Value Graph

Defining the Value Graph Formally (1)

Definition 5.9.2.1 (Value Graph)

Let G_{ssa} be the SSA form of a flow graph G. The value graph $VG_{G_{ssa}} = (V, L, A)$ is a triple, where V is a set of vertices, L a labelling function of vertices, and A a set of directed edges (or arcs).

- Vertices: For every assignment of G_{ssa} with a nontrivial right-hand side term t (i.e., t contains at least one operator), V contains an operator vertex; for every occurrence of a constant in G_{ssa}, V contains a constant vertex.[4]

[4] For the sake of simplicity we assume that all variables are initialized.

Defining the Value Graph Formally (2)

- Labels: Every vertex of V is labelled with the operator or the constant of its underlying right-hand side term or constant occurrence, respectively, and the set of variables whose value is generated by the corresponding assignment or constant.
- Arcs: Every operator vertex of V has for every of its operands an outgoing arc pointing to the vertix of V labelled with this operand.

  Every arc is labelled with a natural number denoting the position of the operand that it points to in the term it is an operand of.[5]

[5] This labelling is omitted in the examples; we assume that edges are ordered implicitly from left to right.

Value Graphs: A few Remarks

Let *VG* be a value graph. By construction, it is ensured that

- operator nodes of *VG* are always annotated with the left-hand side variable of their underlying assignment statement, also called the generating assignment.
- The left-hand side variable x of a trivial assignment
  - $x := y$, $y \in \mathbf{V}$, is attached to the vertex corresponding to the generating assignment of y
  - $x := c$, $c \in \mathbf{C}$, is attached to the constant vertex corresponding to the occurrence of the constant c.

For convenience, constant and operator annotations are written inside the circle visualizing a vertex, variable annotations outside.[6]

[6] For simplicity, we assume that ordinary term operators and ϕ operators are all binary (extensions to k-ary operators are straightforward).

The VG_{sc} Algorithm

Algorithm 5.9.2.2 (Computing VG_{sc} Constants)

Let $VG = (V, L, A)$ be a value graph. Then:

Initialization Step: For every vertex $v \in V$ initialize:

$$\mathrm{dfi}[v] = \begin{cases} I_0(c) & \text{if } v \text{ is a leaf node of VG labelled by } c \\ \top & \text{otherwise} \end{cases}$$

Iteration Step:

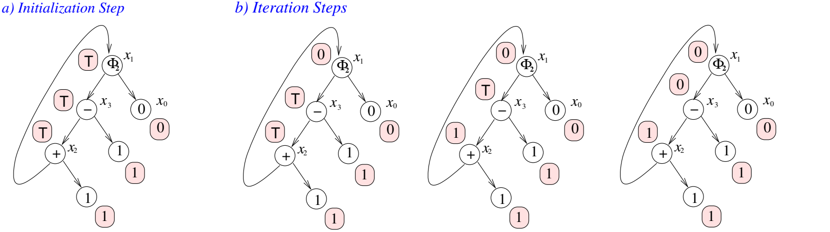
1. For every vertex $v \in V$ labelled by an ord. operator op:

   $$\mathrm{dfi}[v] = I_0(op)(\mathrm{dfi}[\mathrm{l}(v)], \mathrm{dfi}[\mathrm{r}(v)]) \quad \text{(Evaluating terms)}$$

2. For every vertex $v \in V$ labelled by a ϕ operator:

   $$\mathrm{dfi}[v] = \mathrm{dfi}[\mathrm{l}(v)] \sqcap \mathrm{dfi}[\mathrm{r}(v)] \quad \text{(Merging DFA-info's at join nodes)}$$

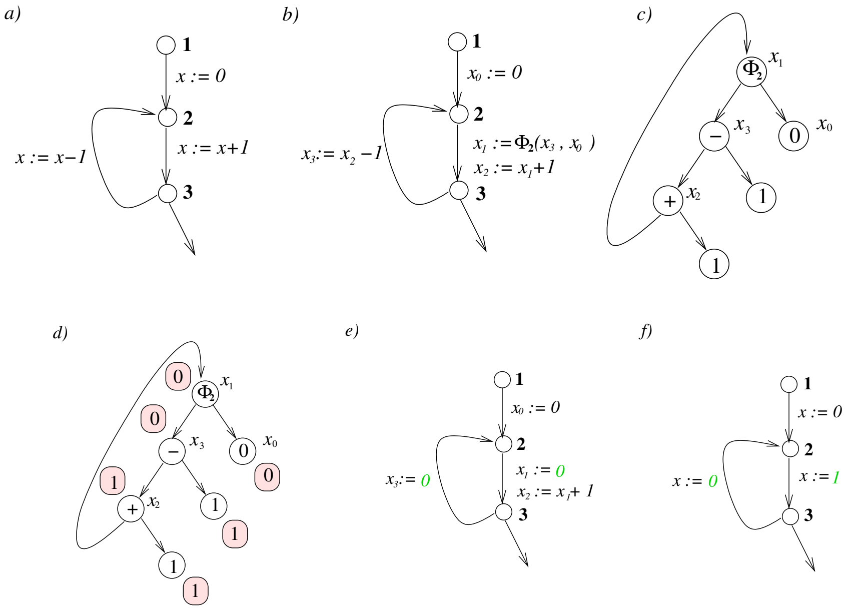
Running Example: Illustrating the VG_{sc} Alg.



After the initialization step　After the 1st iteration step　After the 2nd iteration step　After the 3rd iteration step: Stable!

Analysis result: x_0, x_1, x_2, and x_3 are VG_{sc} constants!

Running Example: The VG_{sc} Optimization

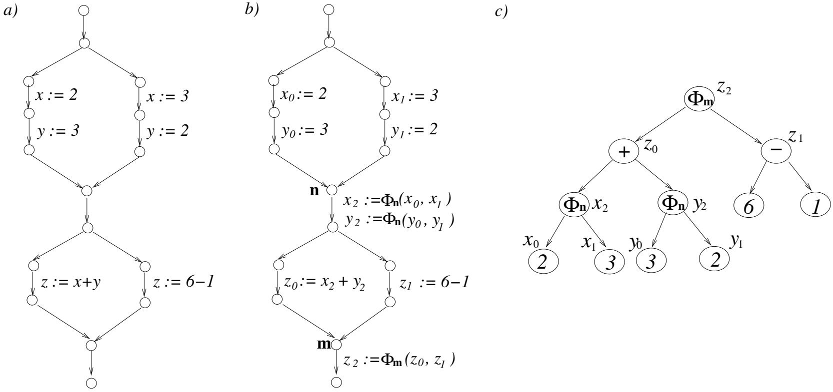


Chapter 5.9.3

VG_ϕ Constants: The Full Approach

The Running Example

...constructing the SSA form and the VG of a program:



Original Flow Graph SSA Form Value Graph

Extending the Data Domain: ϕ Constants

Intuitively, ϕ constants are expressions which are composed of constants and ϕ operators.

Definition 5.9.3.1 (ϕ Constants over ID')

The set ID^ϕ of ϕ constants over ID' is inductively defined as the smallest set satisfying:

1. $\mathsf{ID}' \subseteq \mathsf{ID}^\phi$
2. If ϕ_n is a ϕ operator occurring in the SSA form and $d_1, d_2 \in \mathsf{ID}^\phi$ such that neither d_1 nor d_2 contains ϕ_n, then $\phi_n(d_1, d_2) \in \mathsf{ID}^\phi$.

Note that ID' (cf. Chapter 5.3) and hence ID^ϕ contain the distinguished elements $\perp$ and $\top$.

The Complete Partial Order of ϕ Constants

Lemma 5.9.3.2 (CPO of ϕ Constants)

The pair $(\mathsf{ID}^{\phi}, \sqsubseteq)$, where $\sqsubseteq$ is defined by

$$\phi_n(r_1, r_2) \sqsubseteq r \iff_{df}$$

$$(r_1 \sqsubseteq r \;\vee\; r_2 \sqsubseteq r) \vee (r = \phi_n(r_3, r_4) \wedge r_1 \sqsubseteq r_3 \;\wedge\; r_2 \sqsubseteq r_4)$$

is a complete partial order.

Note: $(\mathsf{ID}^{\phi}, \sqsubseteq)$ is not a lattice since greatest lower bounds do not exist. E.g., $\phi_n(2, 3)$ and $\phi_n(3, 2)$ are incomparable lower bounds of 2 and 3, respectively.

Definition 5.9.3.3 (Evaluation Function $\mathcal{E}^+$)

The evaluation function $\mathcal{E}^+ : V_{L_{(C,Op)}} \to \mathrm{ID}^\phi$ is defined by:

1. $\mathcal{E}^+(d) = d$ if $d \in \mathrm{ID}'$
2. $$\mathcal{E}^+(\phi_n(r_1, r_2)) = \begin{cases} \bot & \text{if } \mathcal{E}^+(r_1) \text{ or } \mathcal{E}^+(r_2) \\ & \quad \text{contains } \phi_n \\ \mathcal{E}^+(r_1) & \text{if } \mathcal{E}^+(r_1) \sqsubseteq \mathcal{E}^+(r_2) \\ \mathcal{E}^+(r_2) & \text{if } \mathcal{E}^+(r_2) \sqsubseteq \mathcal{E}^+(r_1) \\ \phi_n(\mathcal{E}^+(r_1), \mathcal{E}^+(r_2)) & \text{otherwise} \end{cases}$$
3. $\mathcal{E}^+(op(r_1, r_2)) =$
$$\begin{cases} I_0(op)(r_1, r_2) & \text{if } r_1, r_2 \in \mathrm{ID}' \\ \mathcal{E}^+(\phi_n(op(r_1, r_{21}), op(r_1, r_{22}))) & \text{if } r_1 \in \mathrm{ID}', r_2 = \phi_n(r_{21}, r_{22}) \\ \mathcal{E}^+(\phi_n(op(r_{11}, r_2), op(r_{12}, r_2))) & \text{if } r_1 = \phi_n(r_{11}, r_{12}), r_2 \in \mathrm{ID}' \\ \mathcal{E}^+(\phi_n(op(r_{11}, r_{21}), op(r_{12}, r_{22}))) & \text{if } r_1 = \phi_n(r_{11}, r_{12}), \\ & \quad r_2 = \phi_n(r_{21}, r_{22}) \\ \bot & \text{otherwise} \end{cases}$$

Discussing $\mathcal{E}^+$: Intuition

Intuitively

- the evaluation function $\mathcal{E}^+$ maps vertices of the value graph depending on the operator or constant symbol they are annotated with ("inside the circle") to a ϕ constant in ID^ϕ.

Discussing $\mathcal{E}^+$: Power and Performance

Controlling analysis power:

- Def. 5.9.3.3, second item, the "otherwise" case: Here, ϕ constants are constructed, which, as operands of ordinary operators, are evaluated in a distributive fashion (cf. lines two to four of the third item, $\phi_n(\mathcal{E}^+(...))$ and $\mathcal{E}^+(\phi_n...)$)

Controlling performance:

- Def. 5.9.3.3, third item, the "otherwise" case: The evaluation of $\mathcal{E}^+$ yields $\perp$, if r_1 and r_2 are ϕ constants with different top level ϕ operators, i.e., origin from different join nodes in the program. This is in order to avoid the combinatoric explosion which reflects the co-NP-hardness of constant propagation on (even) acyclic programs.

Algorithm 5.9.3.3 (Computing VG_ϕ Constants)

Let $VG = (V, L, A)$ be a value graph. Then:

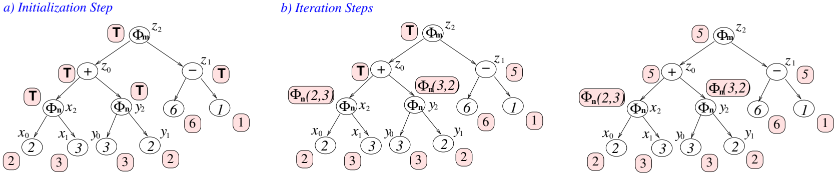
Initialization Step: For every vertex $v \in V$ initialize:

$$\mathrm{dfi}[v] = \begin{cases} \mathcal{E}^+(\mathrm{lab}[v]) & \text{if } v \text{ is a leaf node of } VG \\ \top & \text{otherwise} \end{cases}$$

Iteration Step: For every vertex $v \in V$ labelled by an ordinary or ϕ operator op:

$$\mathrm{dfi}[v] = \mathcal{E}^+(op(\mathrm{dfi}[\mathrm{l}(v)], \mathrm{dfi}[\mathrm{r}(v)]))$$

Running Example: Illustrating the VG_ϕ Alg.



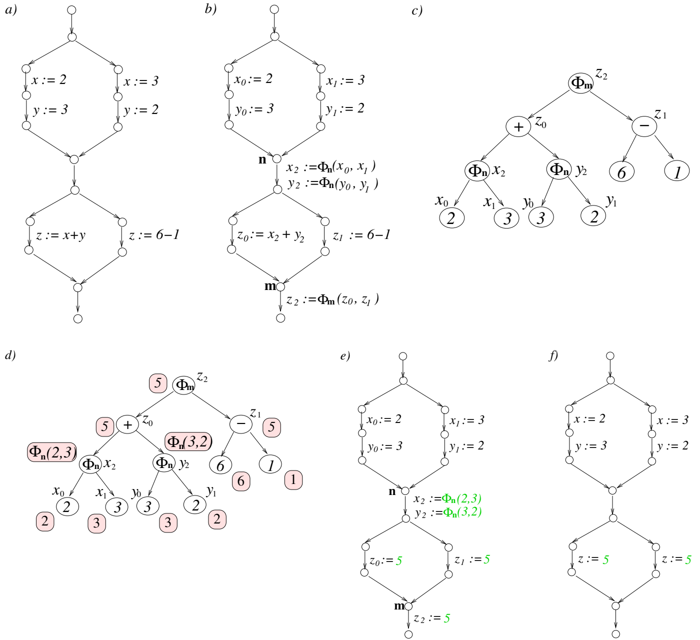
After the initialization step

After the 1st iteration step

After the 2nd iteration step: Stable!

Analysis result: x_0, x_1, y_0, y_1, z_0, z_1, and z_2 are VG_ϕ constants!

Running Example: The VG_ϕ Optimization



The Key

...to obtaining this result:

- Introducing ϕ constants.
- Extending the eval. function on VGs to ϕ constants, $\mathcal{E}^+$.
- Defining $\mathcal{E}^+$ to carefully balancing power and computational complexity of evaluating it.

Chapter 5.9.4

Main Results

Main Result for VG_{sc} and VG_ϕ

Theorem 5.9.4.1 (VG_{sc} Theorem)

The VG_{sc} algorithm computes the class of simple constants.

Theorem 5.9.4.2 (VG_ϕ Theorem)

The VG_ϕ algorithm computes

1. a superset of the set of simple constants for programs with unrestricted control flow.
2. the class of injective constants, i.e., the class of constants composed of operators, which are injective for the relevant term operands, for programs with acyclic control flow.

Overall

...the VG_ϕ algorithm

- keeps a fine balance between power and performance
- achieves a finite but long range look-ahead.

The value graph and the SSA form of a program it is derived from are fundamental for this achievement.

Note:
Beyond its usage for constant propagation on the value graph in this chapter, the SSA form of a program is

- a most widely used intermediate program representation in optimizing compilers.

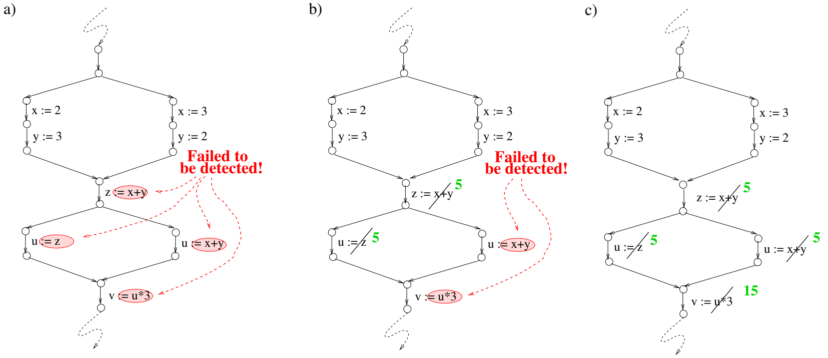
The SSA form of a program is attractive because

- lexical identical terms are ensured to be semantically equivalent, i.e., to always yield the same value, which is important to know for many analyses and optimizations.

Chapter 5.9.5

Illustrating Example

VG_ϕ Constants over $\mathbb{Z}$: Illustrating Example



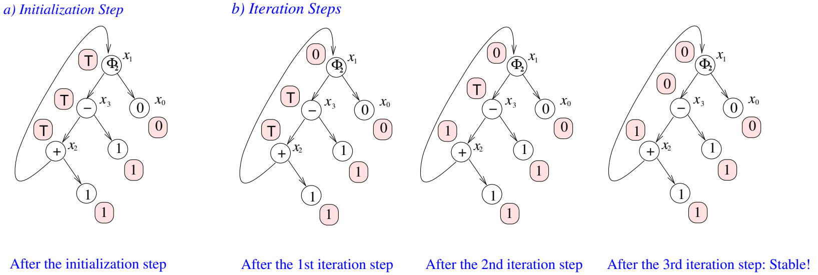
After simple constants propagation (No effect at all!)

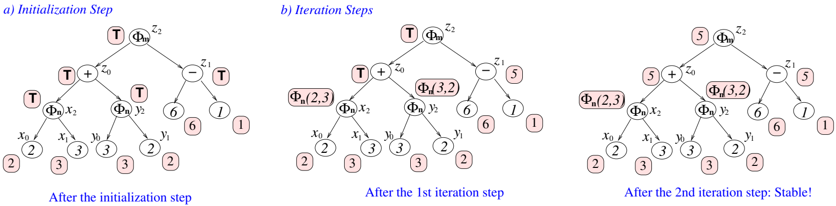
After Q constants propagation

After VG_Φ constants propagation

Constant Propagation on the Value Graph

...receives **Triple E** Rating: **E**xpressive, **E**fficient, **E**asy!





Chapter 5.10

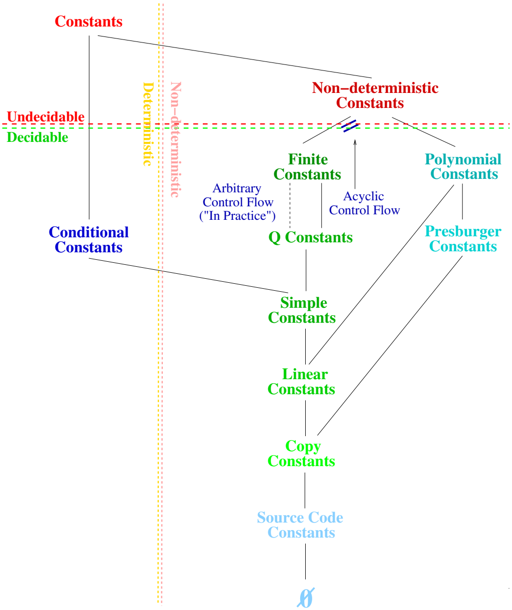
Summary, Looking Ahead

Summary

The undecidability of the general constant propation problem inspired a quest for decidable and for efficiently decidable classes of the constant propagation problem having led to

- Simple constants
- Linear constants
- Copy constants
- Q constants
- Conditional constants
- Finite constants
- VG_ϕ constants
- Presburger constants
- Polynomial constants
- ...

The Lattice of Constant Propagation Classes



Design Strategies and Achievements

- Trading generality and precision for efficiency, scalability
  - Simple constants (standard algorithm for intraprocedural CP)
  - Linear constants (relevant for interprocedural CP)
  - Copy constants (relevant for interprocedural CP)
  - Q constants (modest improvement over simple const.)
  - Conditional constants (improvement over simple constants by branch evaluation, towards deterministic CP)
- Trading generality, efficiency, and scalability for precision
  - Finite constants (arbitrary term operators, decidable for arbitrary control flow, complete for acyclic control flow, intraprocedural)
  - Presburger constants ($+$, $-$: decidable and complete for arbitrary control flow, intraprocedural)
  - Polynomial constants ($+$, $-$, $*$: decidable and complete for arbitrary control flow, intraprocedural)

The Challenges of Constant Propagation

...are nicely illustrated by an example of Markus Müller-Olm and Helmut Seidl (SAS 2002): z at node 4 is a polynomial constant of value 0 but it is not a simple/Q/finite/conditional constant .

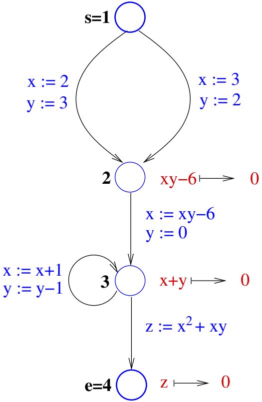
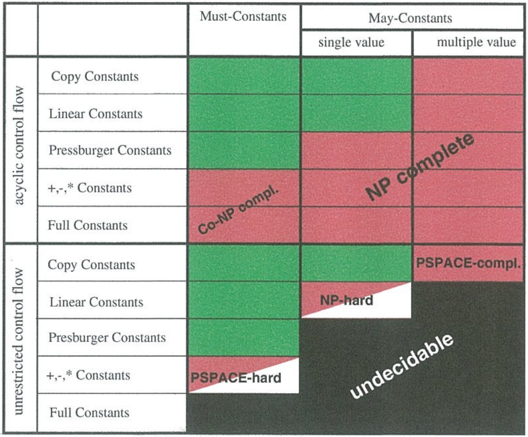


Table 2. Complexity classification of a taxonomy of CP: summarizing the results.



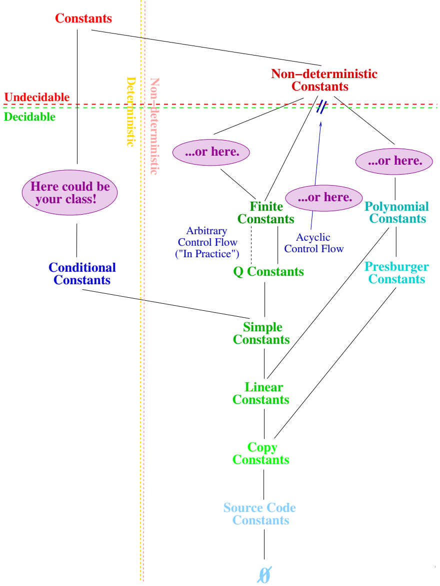
Constant Propagat'n: More than a Commodity

Constant propagation is

- among the most important and most widely used optimizations of classical optimization
- indispensible for designing and engineering safety-critical real-time systems, (e.g., for worst-case execution time analysis (loop bounds computation, recursion depths analysis, etc.) of such systems.

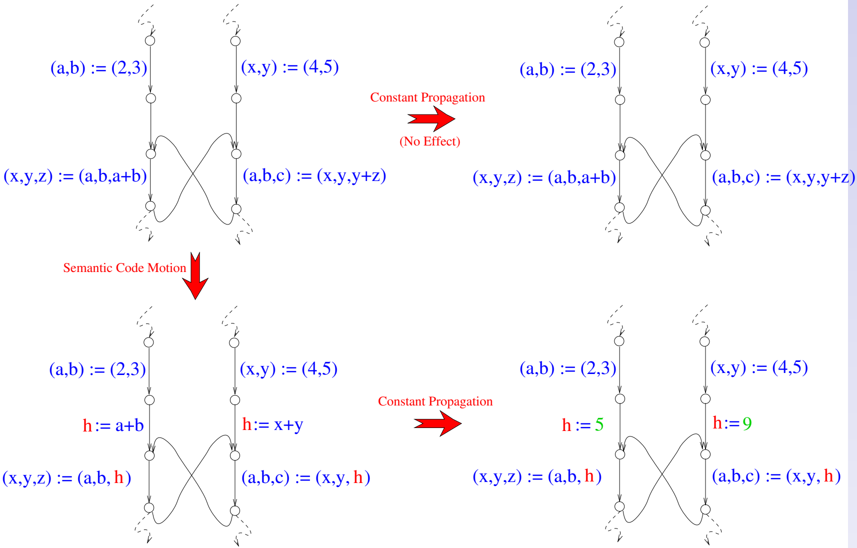
Looking Ahead: Constant Propagation

...a field of challenging Master (and Bachelor) theses.



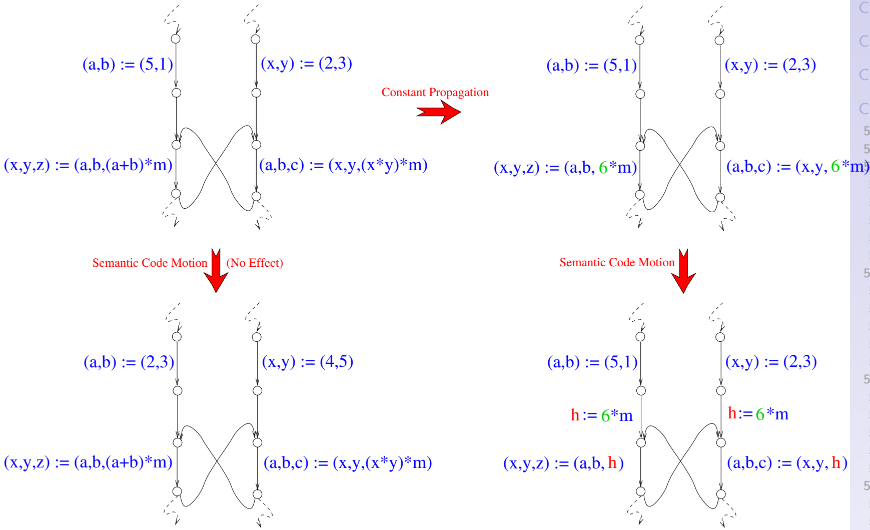
Looking Ahead: Topics for Theses (1)

...combining analyses and optimizations, implement, evaluate:



Looking Ahead: Topics for Theses (2)

...combining analyses and optimizations, implement, evaluate:



Chapter 5.11

References, Further Reading

Further Reading for Chapter 5 (1)

Constant Propagation in Textbooks

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 8.5.4, The Use of Algebraic Identities; Chapter 9.4, Constant Propagation)
- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 4.4.3, Constant Propagation)
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 1, Overview of Compilation; Chapter 8, Introduction to Code Optimization; Chapter 9, Data Flow Analysis)

Further Reading for Chapter 5 (2)

- Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 4.2.3, Constant Propagation; Chapter 4.2.4, Variants of Constant Propagation - Conditional Constant Propagation, Copy Constant Propagation)
- Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. (Chapter 8.3, Constant Propagation)
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 8.14, More Ambitious Analyses)

Further Reading for Chapter 5 (3)

Pioneering Works on Constant Propagation

Simple Constants

Gary A. Kildall. *A Unified Approach to Global Program Optimization*. In Conference Record of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73), 194-206, 1973.

Simple Constants and (Slightly) Beyond: Q Constants

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

Further Reading for Chapter 5 (4)

Undecidability of Constant Propagation, Constant Propagation on the Global Value Graph

- John H. Reif, Harry R. Lewis. *Symbolic Evaluation and the Global Value Graph*. In Conference Record of the 4th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), 104-118, 1977.
- John H. Reif, Harry R. Lewis. *Efficient Symbolic Analysis of Programs*. Aiken Computation Laboratory, Harvard University, TR-37-82, 1982.

Further Reading for Chapter 5 (5)

Complexity of Constant Propagation

- Markus Müller-Olm. *The Complexity of Copy Constant Detection in Parallel Programs*. In Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2001), Springer-V., LNCS 2010, 490-501, 2001.
- Markus Müller-Olm. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*. Springer-V., LNCS 3800, 2006.
- Markus Müller-Olm, Oliver Rüthing. *The Complexity of Constant Propagation*. In Proceedings of the European Symposium on Programming (ESOP 2001), Springer-V., LNCS 2028, 190-205, 2001.

Further Reading for Chapter 5 (6)

Conditional Constants

- Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. In Conference Record of the 12th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'85), 291-299, 1985.

- Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. ACM Transactions on Programming Languages and Systems 13(2):181-210, 1991.

Linear Constants

- Mooly Sagiv, Tom Reps, Susan Horwitz. *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation*. Theoretical Computer Science 167(1-2):131-170, 1996.

Contents
Chap. 1
Chap. 2
Chap. 3
Chap. 4
Chap. 5
5.1
5.2
5.3
5.3.1
5.3.2
5.3.3
5.3.4
5.3.5
5.4
5.4.1
5.4.2
5.4.3
5.4.4
5.4.5
5.5
5.5.1
5.5.2
5.5.3
5.5.4
5.5.5
5.6
5.6.1
5.6.2
5.6.3

Further Reading for Chapter 5 (7)

Polynomial Constants and Finite Constants

- Markus Müller-Olm, Helmut Seidl. *Polynomial Constants are Decidable*. In Proceedings of the 9th Static Analysis Symposium (SAS 2002), Springer-V., LNCS 2477, 4-19, 2002.
- Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. In Proceedings of the 14th International Conference on Mathematical Foundations of Computer Science (MFCS'89), Springer-V., LNCS 379, 481-490, 1989.
- Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. Theoretical Computer Science 80(2):303-318, 1991.

Further Reading for Chapter 5 (8)

Constant Propagation in Specific Settings

- Kathleen Knobe, Vivek Sarkar. *Conditional Constant Propagation of Scalar and Array References Using Array SSA Form*. In Proceedings of the 5th Static Analysis Symposium (SAS'98), Springer-V., LNCS 1503, 33-56, 1998.
- Jens Knoop. *Parallel Constant Propagation*. In Proceedings of the 4th European Conference on Parallel Processing (Euro-Par'98), Springer-V., LNCS 1470, 445-455, 1998.
- Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. Journal of Universal Computer Science 9(8):829-850, 2003.

Further Reading for Chapter 5 (9)

- Samuel P. Midkiff, José E. Moreira, Marc Snir. *A Constant Propagation Algorithm for Explicitly Parallel Programs*. International Journal of Computer Science 26(5):563-589, 1998.

Constant Propagation on the Value Graph

- Jens Knoop, Oliver Rüthing. *Constant Propagation on the Value Graph: Simple Constants and Beyond*. In Proceedings of the 9th International Conference on Compiler Construction (CC 2000), Springer-V., LNCS 1781, 94-109, 2000.
- Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. Journal of Universal Computer Science 9(8):829-850, 2003. (Special issue for SBLP'03).

Further Reading for Chapter 5 (10)

- Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. In Proceedings of the 7th Brazilian Symposium on Programming Languages (SBLP 2003), 135-148, 2003.

Constructing SSA Form

- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Ken Zadeck. *An Efficient Method of Computing Static Single Assignment Form*. In Conference Record of the 16th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89), 25-35, 1989.

Further Reading for Chapter 5 (11)

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Ken Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. ACM Transactions on Programming Languages and Systems 13(4):451-490, 1991.

Constructing the Value Graph

Bowen Alpern, Mark N. Wegman, F. Ken Zadeck. *Detecting Equality of Variables in Programs*. In Conference Record of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 1-11, 1988.

Further Reading for Chapter 5 (12)

Miscellaneous

- Yuri V. Matijasevic. *Enumerable Sets are Diophantine (In Russian)*. Dodl. Akad. Nauk SSSR 191, 279-282, 1970.
- Yuri V. Matijasevic. *What Should We Do Having Proved a Decision Problem to be Unsolvable?* Algorithms in Modern Mathematics and Computer Science 1979:441-448, 1979.
- Yuri V. Matijasevic. *Hilbert's Tenth Problem*. MIT Press, 1993.
- Robert E. Tarjan. *Fast Algorithms for Solving Path Problems*. Journal of the ACM 28(3):594-614, 1981.

Chapter 6

Partial Redundancy Elimination

Chapter 6.1

Motivation

Motivation: Looking Back and Ahead

Looking back

- Classical Gen/Kill Data Flow Analyses
  - Focus: Proving soundness and completeness of DFAs for selected program properties (availability, liveness, very busyness, etc.); no program optimization involved (cf. Chapter 2, 3, and 4).
- Constant Propagation
  - Focus: Proving soundness and (relative) completeness of DFAs for non-deterministic and deterministic constant propagation (SCs, LCs, CpCs, QCs, FCs, CCs); program optimizations involved but trivial (cf. Chapter 3 and 5).

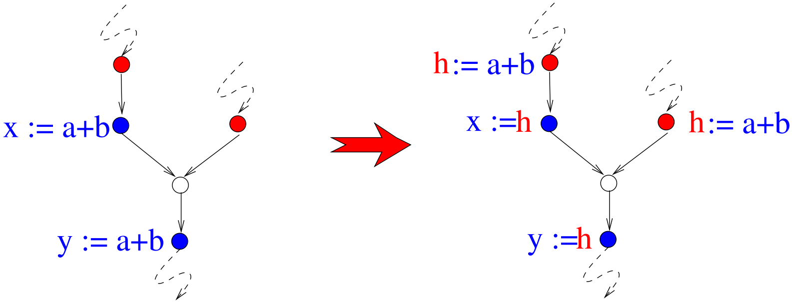
Looking ahead

- Partial Redundancy Elimination
  - Focus: Proving optimality of several non-trivial program optimizations (busy code motion, lazy code motion, sparse code motion) (cf. Chapter 7, 8, and 9).

Partial Redundancy Elimination (PRE)

What's it all about?

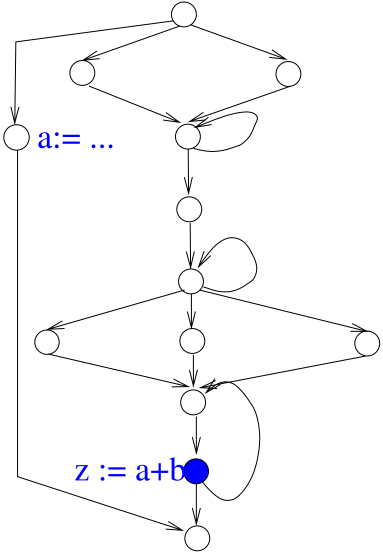
...avoiding multiple (re-) computations of the same value!



Chapter 6.2

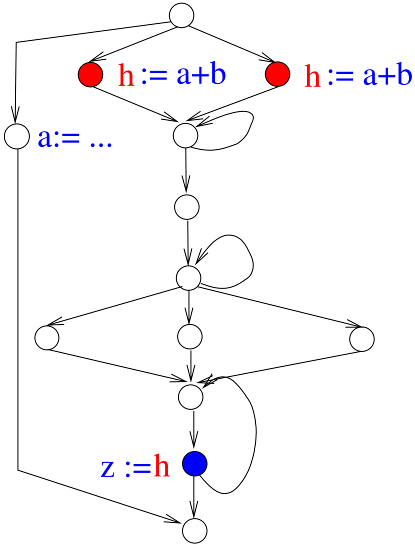
PRE: Essence and Objectives

PRE – Particularly Striking for Loops

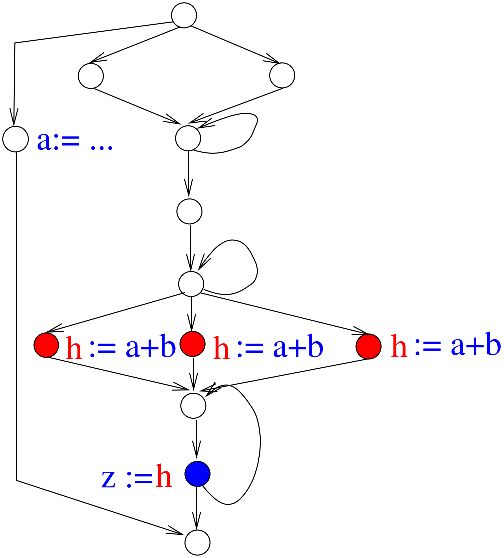


A Computationally Optimal Program

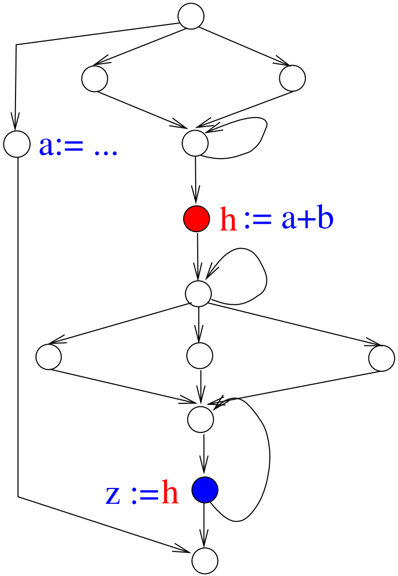
...w/out any redundancy at all!



Often there is more than one!



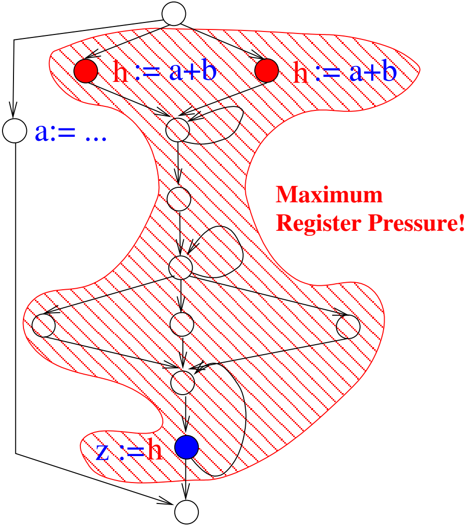
Which one shall PRE deliver?



The (Optimization) Goals make the Difference!

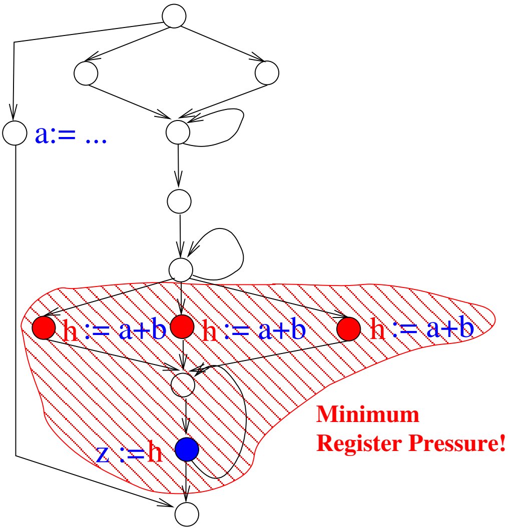
Transformation I

...no redundancies but maximum register pressure!



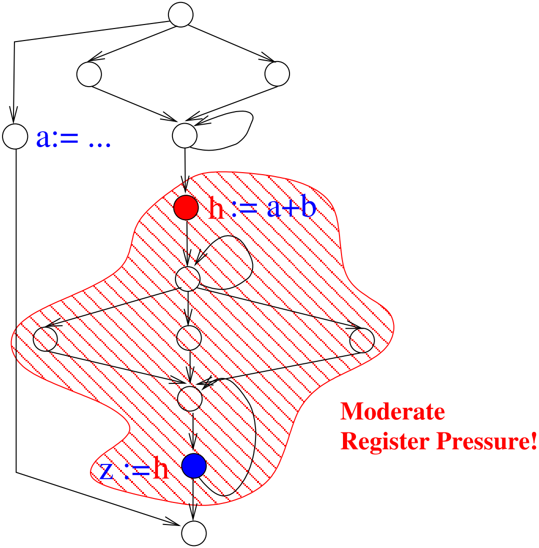
Transformation II

...no redundancies but minimum register pressure!



Transformation III

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



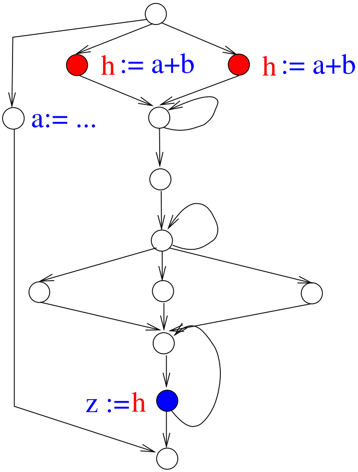
The (Optimization) Goals make the Difference!

In our running example:

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

Transformation I: Busy Code Motion

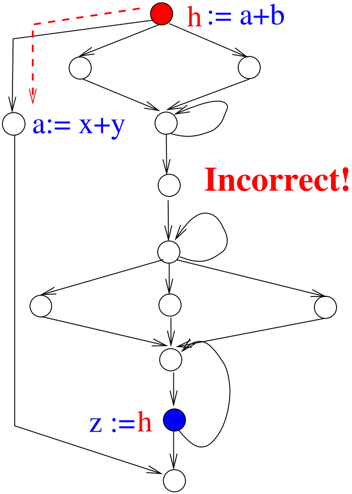
...placing computations as early as possible!



...yields computationally optimal programs.

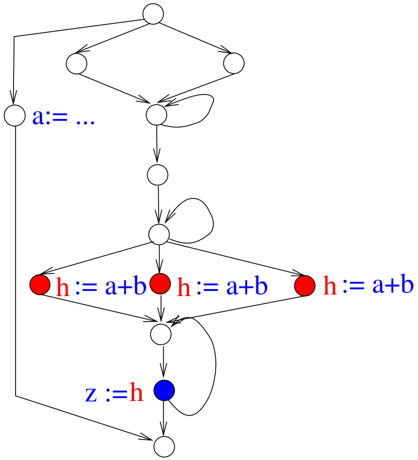
Note: As Early as Possible

...means earliest but not earlier.



Transformation II: Lazy Code Motion

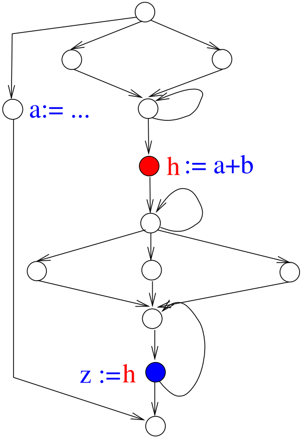
...placing computations as late as possible!



...yields computationally and lifetime optimal programs.

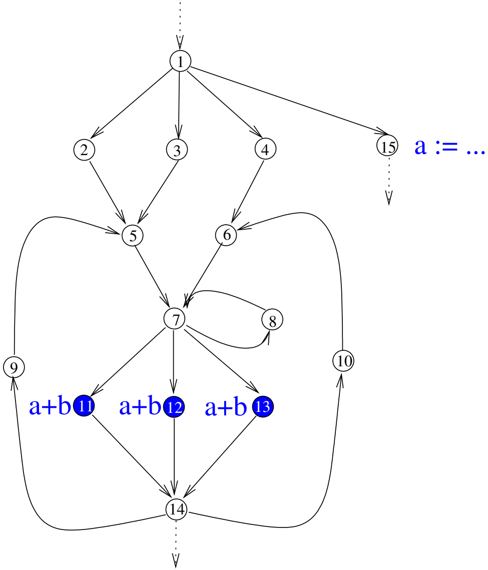
Transformation III: Sparse Code Motion

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

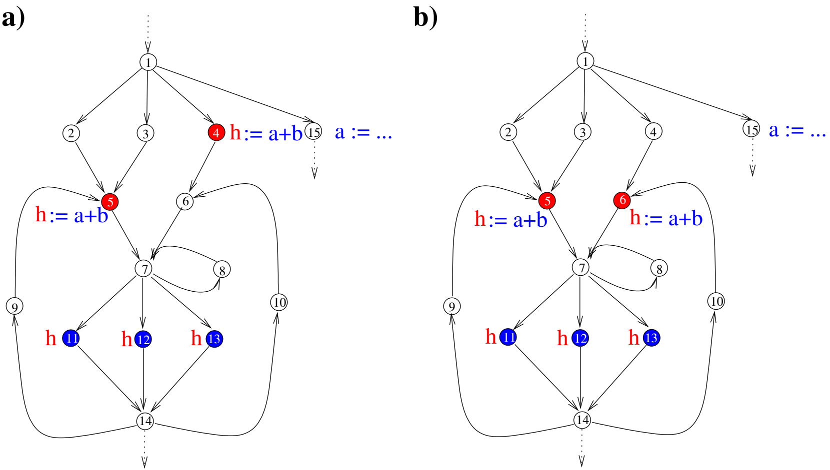


...yields comp. and lifetime best code-size optimal programs.

Illustrating PRE: A More Complex Example (1)

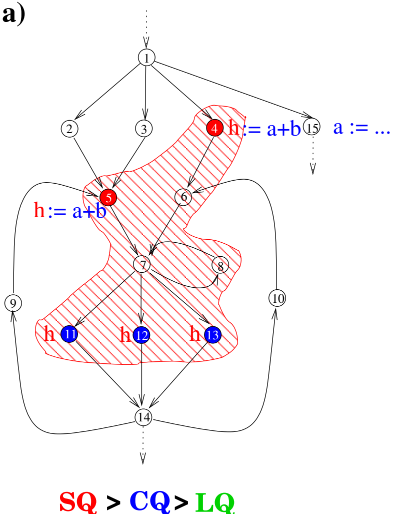


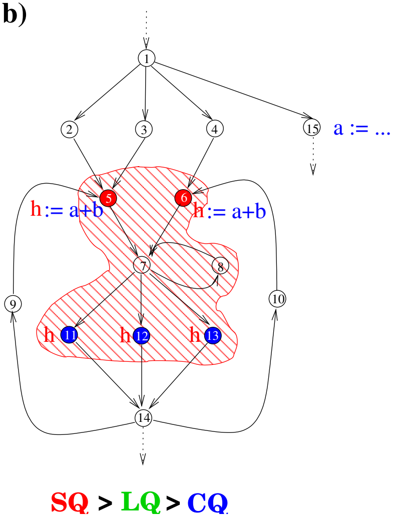
Illustrating PRE: A More Complex Example (2)



Two Code–size Optimal Programs

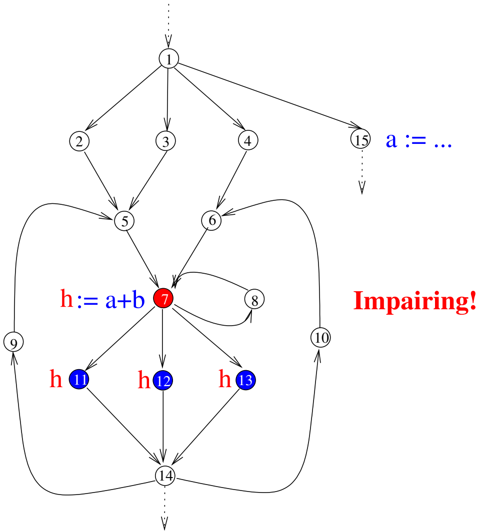
Illustrating PRE: A More Complex Example (3)





Illustrating PRE: A More Complex Example (4)

Note: The below transformation is not desired!



Summing up

The previous examples demonstrate that in general we can not achieve

- computational optimality
- lifetime optimality
- space optimality

at the same time.

However, given a

- prioritization of computational/lifetime/space optimality

we can deliver a program that is

- optimal with respect to the requested prioritization

of these goals.

Chapter 6.3

The Groundbreaking PRE Algorithm of Morel and Renvoise

The Groundbreaking Algorithm for PRE

PRE (or Code Motion (CM)) is intrinsically tied to Etienne Morel and Claude Renvoise.

Conceptually

- The PRE algorithm of Morel and Renvoise presented in 1979 can be considered the *prime father* of all code motion (CM) algorithms
- continued to be the “state of the art” CM algorithm until the early 1990s.

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

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

The PRE Algorithm of Morel & Renvoise (1)

The PRE Analyses

I Availability

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

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

II Partial Availability

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

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

The PRE Algorithm of Morel & Renvoise (2)

III Very Busyness (Anticipability)

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

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

The PRE Algorithm of Morel & Renvoise (3)

IV Placement Possible

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

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

where

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

The PRE Algorithm of Morel & Renvoise (4)

The PRE Transformation

Initializing temporaries

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

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

Replacing original computations by references to temporaries

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

Achievements, Merits

...of Morel and Renvoise's PRE algorithm:

- First algorithm for global PRE

  Before 1979: PRE restricted to

  - Basic blocks: Value numbering
  - Program loops: Loop invariant code motion
- Computationally optimal results
- State-of-the-art algorithm for global PRE for about 15 years

Shortcomings, Limitations

...of Morel and Renvoise's PRE algorithm:

- Conceptually
  - Computational optimality

    $\leadsto$ Achieved if critical edges are split (not part of the original algorithm formulation)
  - Lifetime optimality

    $\leadsto$ Register pressure is heuristically dealt with, no optimality
  - Code-size optimality

    $\leadsto$ Not addressed, no objective
- Technically
  - Bi-directional

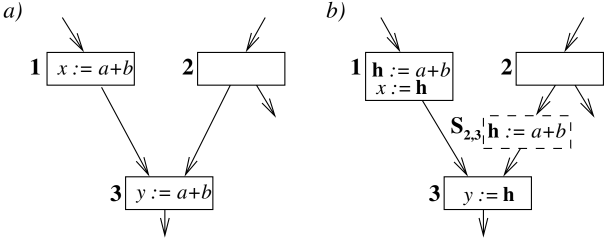
    $\leadsto$ conceptually and computationally more complex than uni-directional analyses

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

Critical Edges

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

Illustration:



...splitting the critical edge from node **2** to node **3** by inserting the synthetic node $\mathbf{S}_{2,3}$ allows PRE to eliminate the partially redundant computation of $a + b$ at node **3**, which would not safely be possible otherwise.

Chapter 6.4

References, Further Reading

Further Reading for Chapter 6 (1)

The Groundbreaking PRE Algorithm of Morel and Renvoise

- Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.

Variations and Improvements on Morel/Renvoise's Algorithm

- 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.
- Karl-Heinz Drechsler, Manfred P. Stadel. *A Solution to a Problem with Morel and Renvoise's "Global Optimization by Suppression of Partial Redundancies"*. ACM Transactions on Programming Languages and Systems 10(4):635-640, 1988, Technical Correspondence.

Further Reading for Chapter 6 (2)

Textbook Presentations of PRE

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 9.5, Partial-Redundancy Elimination)
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 8.6, Global Redundancy Elimination)
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 13, Redundancy Elimination)

Miscellaneous

- Andrei P. Ershov. *On Programming of Arithmetic Operations*. Communications of the ACM 1(8):3-6, 1958.

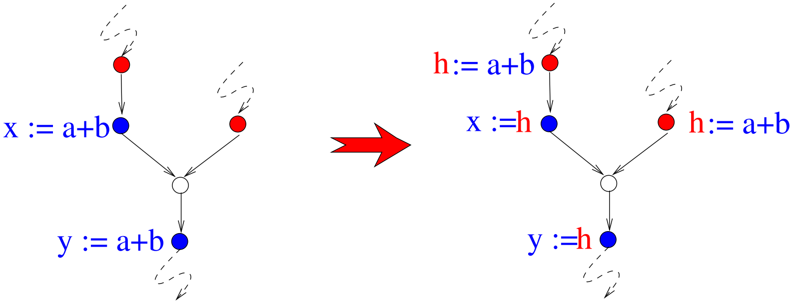
Chapter 7

Busy Code Motion

Code Motion: Recalling the Very Idea

Code Motion (CM) – often synonymously denoted as Partial Redundancy Elimination (PRE) – aims at:

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



Chapter 7.1

Preliminaries, Problem Definition

Work Plan

In the following we will introduce and formally define:

- The set of CM transformations CM
- The set of admissible CM transformations $\mathcal{CM}_{Adm}$
- The set of computationally optimal CM transformations $\mathcal{CM}_{CmpOpt}$
- The *BCM* transformation as one specific computationally optimal CM transformation

Before, however, we will recall useful notations and common assumptions.

Useful Notations

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph. Then

- $pred(n) =_{df} \{m \mid (m, n) \in E\}$ denote the set of all predecessors
- $succ(n) =_{df} \{m \mid (n, m) \in E\}$ denote the set of all successors
- $source(e)$, $dest(e)$ denote the start node and end node of an edge
- a sequence of edges $(e_1, \ldots, e_k)$ with $dest(e_i) = source(e_{i+1})$ for all $1 \leq i < k$ denotes a finite path.

  Note: We also consider sequences of nodes as paths, if appropriate.

Useful Notations (Cont'd)

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

Assumptions on Flow Graphs

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph. As common and w/out losing generality we assume:

Common assumptions in program analysis and optimization

- G is a node-labelled SI graph
- Every node of G lies on a path from **s** to **e**

  Intuitively: Unreachable parts of G are removed.

CM specific assumption

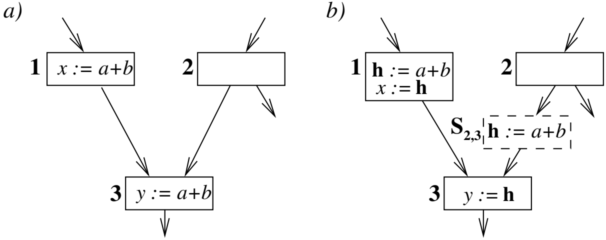
- Critical edges of G are split by inserting new so-called synthetic nodes

  Note: Splitting critical edges is required to enable computationally optimal transformation results.

Recalling the Splitting of Critical Edges

...edges connecting a branch node with a join node are crucial for code motion and are thus considered critical:

Illustration:



...the critical edge $(2,3)$ connecting branch node 2 and join node 3 is split by introducing the synthetic node $\mathbf{S}_{2,3}$ and allows us to remove the partially redundant computation of $a+b$ at 3.

Splitting Critical Edges vs. Join Edges

Splitting critical edges

- Computationally optimal CM results can be achieved, if just critical edges in a flow graph are split.
- CM algorithms need to store results of computations for later reuse at both node entries (N-initializations) and at node exits (X-Initializations).
- Algorithmically, this is not a problem at all.

Splitting join edges

- Splitting all edges leading to a join node (and not just critical ones) simplifies (the presentation of) code motion.
- Computationally optimal CM results can be achieved by storing the results of computations for later resuse uniformly at node entries (N-initializations).

Final Assumption: Join Edges are Split

In the following we thus assume

- Every edge in G leading to a join node is split by inserting a synthetic node.

  Note: Synthetic nodes, where no instruction will be placed, can be removed after the transformation in a final cleaning step.

Example

- Join edges like the one connecting node **1** and node **3** in the example illustrating the splitting of critical edges are assumed to be split by inserting a synthethic node $\mathbf{S}_{1,3}$.

Chapter 7.1.1

Code Motion

CM: The General Transformation Pattern

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a (node-labelled SI) flow graph, and let $t \in \mathbf{T}$ be a term, the so-called candidate expression for code motion.

Definition 7.1.1.1 (CM Transformation)

A CM transformation for t, CM_t, consists of two steps:

- Inserting at (the entry of) some nodes of G the instruction $h := t$, where h is a new variable.
- Replacing some of the original occurrences of t by h.

The set of CM transformations for t is denoted by $\mathcal{CM}_t$.

Specifying CM Transformations

A CM transformation CM_t is completely specified by means of two predicates (defined on nodes)

- $Insert_{CM_t} : N \rightarrow \mathbb{B}$
- $Repl_{CM_t} : N \rightarrow \mathbb{B}$

specifying where to store the result of a computation and where to replace an original computation of t by a reference to a stored value in G, respectively.

In the following we will consider a fixed candidate expression t allowing us to drop t as an index.

Chapter 7.1.2

Admissible Code Motion

Towards Admissible CM Transformations

Obviously, $\mathcal{CM}$ includes transformations, which do not preserve the semantics of the original program, and are thus not acceptable.

This leads us to the notion of admissible CM transformations:

- A CM transformation $CM \in \mathcal{CM}$ is called admissible, if CM is safe and correct.

Informally:

- Safe: There is no path, on which by inserting an initialization of h a new value is computed, i.e., a value that has not been computed in the original program along this path.
- Correct: Whenever the temporary h is referenced, it stores the "right" value, i.e., it stores the same value a recomputation of t at the use site would yield.

Towards formalising Safety and Correctness

We need to have three (local) predicates (defined on nodes):

- $Comp_t(n)$: the candidate expression t is computed at n.
- $Transp_t(n)$: n is transparent for t, i.e., n does not modify any operand of t.
- $Comp_{CM_t}(n) =_{df} Insert_{CM_t}(n) \lor Comp_t(n) \land \neg Repl_{CM_t}(n)$: The candidate expression t is computed at node n after CM_t has been applied.

Note: In the following we will resume dropping t as an index.

Extending Predicates from Nodes to Paths

Let p be a path (in terms of sequence of nodes) and let p_i denote the i-th node of p.

Then we define:

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

Safety and Correctness

Definition 7.1.2.1 (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:
   $Correct_{CM}(n) \iff_{df} \forall \langle n_1, \ldots, n_k \rangle \in \mathbf{P}[s, n]$
   $\exists\, i.\; Insert_{CM}(n_i) \wedge Transp^{\forall}(\langle n_i, \ldots, n_{k-1} \rangle)$

Up-Safety and Down-Safety

Considering the conditions (i) resp. (ii) of the definition of safety separately, leads us to the notions of

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

Intuitively, a computation of t at node n is

- up-safe, if t is computed on all paths p from $\mathbf{s}$ 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 $\mathbf{e}$ and the first computation of t on p is not preceded by a modification of (an operand of) t.

Up-Safety and Down-Safety

Definition 7.1.2.2 (Up-Safety and Down-Safety)

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

Admissible CM Transformations

Now we are ready to define the set of admissible CM transformations:

Definition 7.1.2.3 (Admissible CM-Transformation)

A CM transformation $CM \in \mathcal{CM}$ is admissible iff for every node $n \in N$ holds:

1. $Insert_{CM}(n) \Rightarrow Safe(n)$
2. $Repl_{CM}(n) \Rightarrow Correct_{CM}(n)$

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

Important Results on Safety and Correctness

Lemma 7.1.2.4 (Safety)

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

Lemma 7.1.2.5 (Correctness)

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

Chapter 7.1.3

Computationally Optimal Code Motion

Computationally Better: Higher Performance

Definition 7.1.3.1 (Computationally Better)

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

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

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

Computat. Optimal: Highest Performance

Definition 7.1.3.2 (Comp. Optimal Code Motion)

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

The set of computationally optimal CM transformations is denoted by $\mathcal{CM}_{CmpOpt}$.

Reminder: Selected Properties of Relations

Let M be a set and R be a relation on M, i.e., $R \subseteq M \times M$.

Then R is called

- reflexive iff $\forall\, m \in M.\ m\,R\,m$
- transitive iff $\forall\, m, n, p \in M.\ m\,R\,n \ \wedge\ n\,R\,p \ \Rightarrow\ m\,R\,p$
- anti-symmetric iff
  $\forall\, m, n \in M.\ m\,R\,n \ \wedge\ n\,R\,m \ \Rightarrow\ m = n$
- quasi order iff R is reflexive and transitive
- partial order iff R is reflexive, transitive and anti-symmetric

Chapter 7.2

The *BCM* Transformation

Conceptually

...code motion can be considered a two-stage process:

1. Hoisting expressions
   ...hoisting expressions to "earlier" safe computation points
2. Eliminating totally redundant expressions
   ...eliminating computations getting totally redundant by hoisting expressions

The Earliestness Principle

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

Placing computations as early as possible...

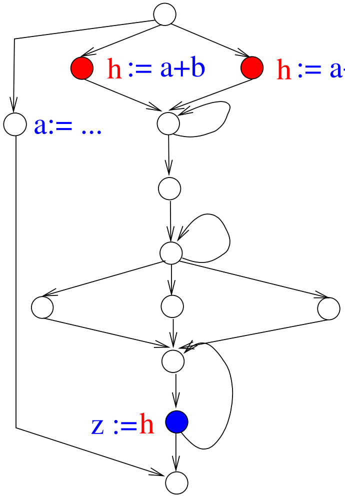
- Theorem (Computational Optimality)
  ...hoisting computations to their earliest safe computation points yields computationally optimal programs.

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

Illustrating the Earliestness Principle

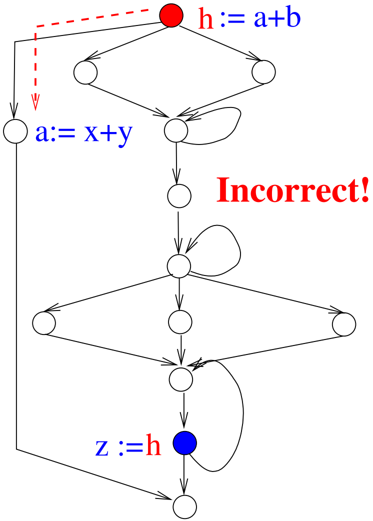
Placing computations as early as possible...

yields computationally optimal programs.



Note

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



Busy Code Motion

Intuitively:

Place computations as early as possible in a program while preserving safety and correctness!

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

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

Earliest Program Points

Definition 7.2.1 (Earliestness)

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

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

The *BCM* Transformation

The *BCM* Transformation is defined by:

- $\forall\, n \in N.\ Insert_{BCM}(n) =_{df} Earliest(n)$
- $\forall\, n \in N.\ Repl_{BCM}(n) =_{df} Comp(n)$

The *BCM* Transf.: Computationally Optimal

Theorem 7.2.2 (*BCM* Theorem)

The *BCM* transformation is computationally optimal, i.e.,

$$BCM \in \mathcal{CM}_{CmpOpt}$$

Proof. By means of the Earliestness Lemma 7.2.3 and the *BCM* Lemma 7.2.4.

Properties of Earliest Program Points

Lemma 7.2.3 (Earliestness Lemma)

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

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

Properties of the *BCM* Transformation

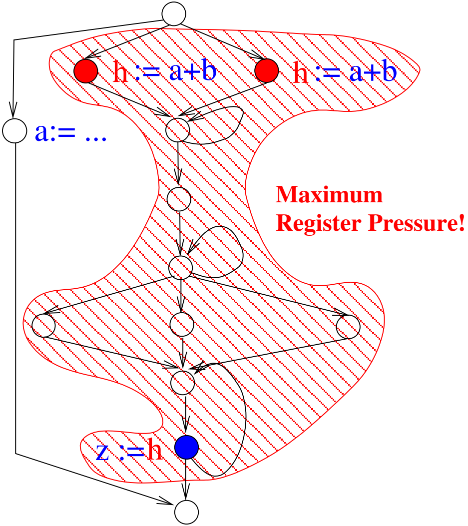
Lemma 7.2.4 (*BCM* Lemma)

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

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

The Result of the *BCM* Transformation

...computationally optimal but maximum register pressure.



Chapter 7.3

Up-Safety and Down-Safety: The DFA Specifications

Note

Up-safety and down-safety

- are just synonyms for availability and very busyness, respectively.

Hence

- the DFA specifications for availability and very busyness of Chapter 4 can be reused and need only be adapted from edge to node-labelled SI flow graphs.

Up-Safety: The DFA Specification

...for a CM candidate expression t, $t \in \mathbf{T}$.

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathbb{B}, \wedge, \vee, \leq, false, true) = \widehat{\mathbb{B}}$
- DFA functional
  $[\![\ ]\!]_{us}^{t} : N \rightarrow (\mathbb{B} \rightarrow \mathbb{B})$, where
  $\forall\, n \in N\ \forall\, b \in \mathbb{B}.\ [\![\, n\,]\!]_{us}^{t}(b) =_{df} (b \vee Comp_n^t) \wedge Transp_n^t$
- Initial information: $b_{\mathbf{s}} \in \mathbb{B}$
- Direction of information flow: forward

Up-Safety Specification for t

- Specification: $\mathcal{S}_G^{us,t} = (\widehat{\mathbb{B}}, [\![\ ]\!]_{us}^{t}, b_{\mathbf{s}}, fw)$

Down-Safety: The DFA Specification

...for a CM candidate expression t, $t \in \mathbf{T}$.

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top) =_{df} (\mathbb{B}, \wedge, \vee, \leq, \mathit{false}, \mathit{true}) = \widehat{\mathbb{B}}$
- DFA functional
  $[\![\ ]\!]_{ds}^{t} : N \to (\mathbb{B} \to \mathbb{B})$, where
  $\forall\, n \in N\ \forall\, b \in \mathbb{B}.\ [\![\, n\,]\!]_{ds}^{t}(b) =_{df} (b \wedge \mathit{Transp}_n^t) \vee \mathit{Comp}_n^t$
- Initial information: $b_{\mathbf{e}} \in \mathbb{B}$
- Direction of information flow: backward

Down-Safety Specification for t

- Specification: $\mathcal{S}_G^{ds,t} = (\widehat{\mathbb{B}}, [\![\ ]\!]_{ds}^{t}, b_{\mathbf{e}}, bw)$

A Hint to Appendix C

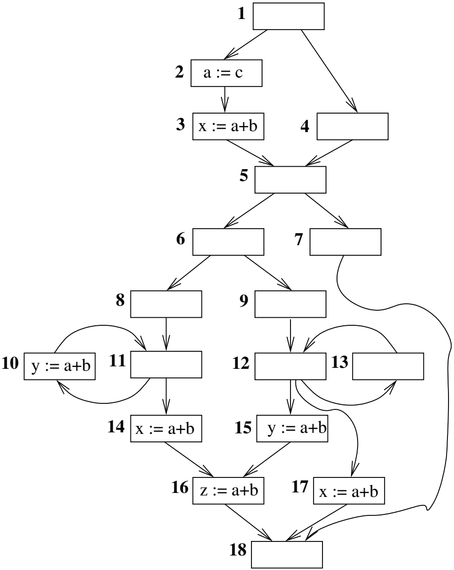
Appendix C presents

- the specialized versions of the *MaxFP* equation systems induced by $\mathcal{S}_G^{us,t}$ and $\mathcal{S}_G^{ds,t}$, respectively, for
  - single instruction flow graphs (cf. Appendix C.1.2)
  - basic block flow graphs (cf. Appendix C.2.2).
- an illustrating example of the *BCM* transformation on basic block flow graphs.

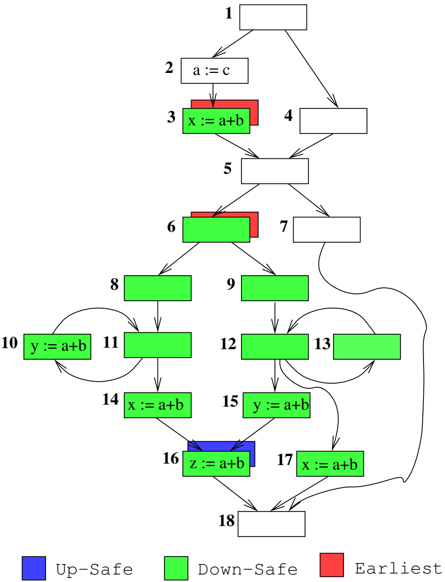
Chapter 7.4

Illustrating Example

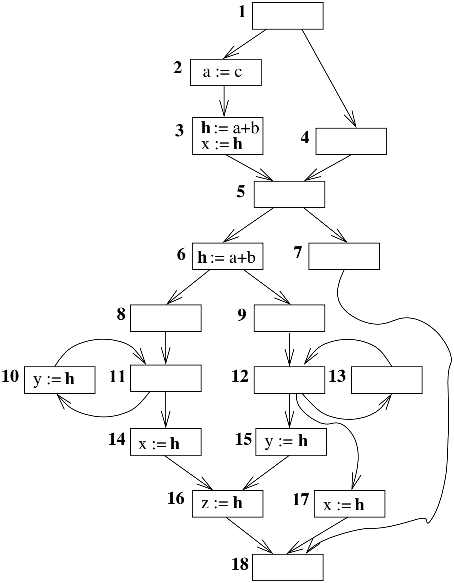
The Original Program



Up-Safe, Down-Safe & Earliest Program Points

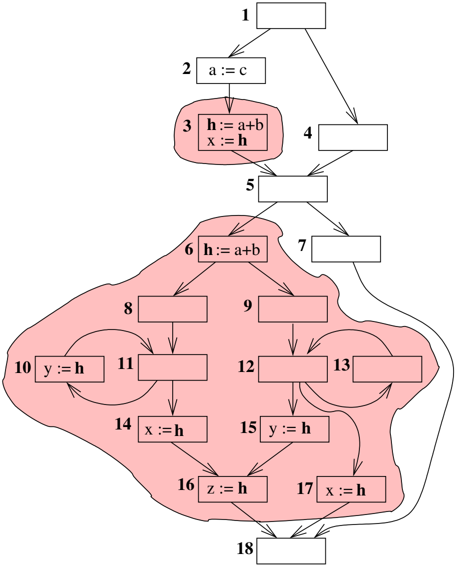


The Result of the *BCM* Transformation

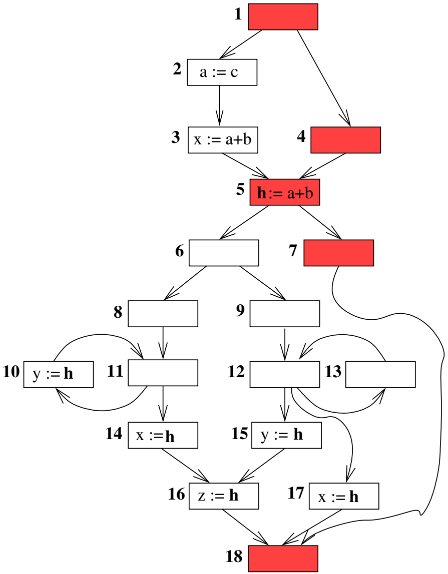


BCM Transf.: Achievements & Shortcomings

Computationally optimal but maximum register pressure.



Note: Initializing Even Earlier is Not Correct!



Chapter 7.5

References, Further Reading

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.

Chapter 8

Lazy Code Motion

The Latestness Principle

...induces an extreme placing strategy dual to the earliestness principle:

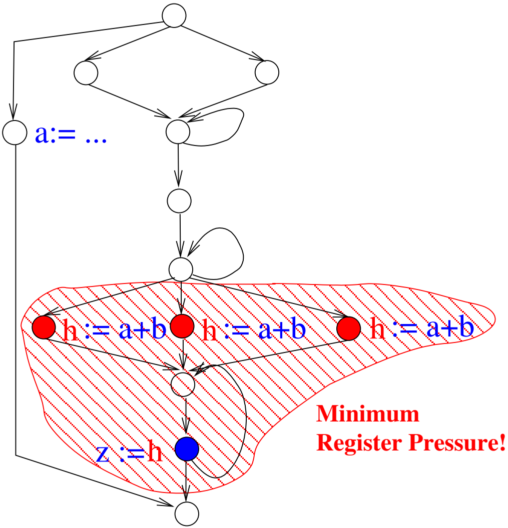
Placing computations as late as possible...

- 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

Illustrating the Latestness Principle

...computationally optimal w/ minimum register pressure!



Lazy Code Motion

Intuitively:

Place computations as late as possible in a program while preserving safety, correctness and computational optimality!

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

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

Chapter 8.1

Preliminaries, Problem Definition

Work Plan

In the following we will introduce and formally define:

- The notion of a lifetime range and a first-use lifetime range.
- The set of almost lifetime optimal CM transformations $\mathcal{CM}_{ALtOpt}$.
- The set of lifetime optimal CM transformations $\mathcal{CM}_{LtOpt}$.
- The LCM transformation as the uniquely determined sole computationally and lifetime optimal CM transformation.

Chapter 8.1.1

Lifetime Ranges

Central for Capturing Register Pressure

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

Definition 8.1.1.1 (Lifetime Ranges)

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

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

First-Use Lifetime Ranges do not Overlap

Lemma 8.1.1.2 (First-Use Lifetime-Range Lemma)

Let $CM \in \mathcal{CM}$, $p \in \mathbf{P}[s,e]$, and let i_1, i_2, j_1, j_2 indexes such that $p[i_1,j_1] \in \mathit{FU\text{-}LtRg}(CM)$ and $p[i_2,j_2] \in \mathit{FU\text{-}LtRg}(CM)$. Then we have:

- either $p[i_1,j_1]$ and $p[i_2,j_2]$ coincide, i.e., $i_1 = i_2$ and $j_1 = j_2$, or
- $p[i_1,j_1]$ and $p[i_2,j_2]$ are disjoint, i.e., $j_1 < i_2$ or $j_2 < i_1$.

Lifetime Better: Less Register Pressure

Definition 8.1.1.3 (Lifetime Better)

A CM-transformation $CM \in \mathcal{CM}$ is lifetime better than a CM-transformation $CM' \in \mathcal{CM}$ iff

$$\forall\, p \in LtRg(CM).\ \exists\, q \in LtRg(CM').\ p \sqsubseteq q$$

Note: The relation "lifetime better" is a partial order, i.e., a reflexive, transitive, and antisymmetric relation.

Chapter 8.1.2

Almost Lifetime Optimal Code Motion

Almost Lifetime Optimality: Almost Mininum Register Pressure

Definition 8.1.2.1 (Almost Lifetime Optimal CM)

A computationally optimal CM transformation $CM \in \mathcal{CM}_{CmpOpt}$ is almost lifetime optimal iff

$$\forall\, p \in LtRg(CM).\ \lambda_p \geq 2 \Rightarrow$$
$$\forall\, CM' \in \mathcal{CM}_{CmpOpt}\ \exists\, q \in LtRg(CM').\ p \sqsubseteq q$$

The set of all almost lifetime optimal CM transformations is denoted by $\mathcal{CM}_{ALtOpt}$.

Chapter 8.1.3

Lifetime Optimal Code Motion

Lifetime Optimal: Minimum Register Pressure

Definition 8.1.3.1 (Lifetime Optimal Code Motion)

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.

The set of all lifetime optimal CM transformations is denoted by $\mathcal{CM}_{LtOpt}$.

BCM Lifetime Ranges are Longest

Lemma 8.1.5 (*BCM* Lifetime Range Lemma)

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

$$p \sqsubseteq q$$

Intuitively

- There is no computationally optimal CM transformation which places computations earlier than the *BCM* transformation.
- The *BCM* transformation is the uniquely determined computationally optimal CM transformation w/ maximum register pressure.

Uniqueness of Lifetime Optimal Code Motion

Obviously, we have:

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

In fact, we have even:

Theorem 8.1.6 (Uniqueness of Lifetime Opt. CM)

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

Chapter 8.2

The *ALCM* Transformation

Delayability of Computations

Definition 8.2.1 (Delayability)

$\forall\, n \in N.\ \mathit{Delayed}(n) \iff_{df}$

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

Lemma 8.2.2 (Delayability Lemma)

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

Latest Program Points

Definition 8.2.3 (Latestness)

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

Lemma 8.2.4 (Latestness Lemma)

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

The $\mathcal{ALCM}$ Transformation

The *ALCM* Transformation is defined by:

- $\forall\, n \in N.\ Insert_{ALCM}(n) =_{df} Latest(n)$
- $\forall\, n \in N.\ Repl_{ALCM}(n) =_{df} Comp(n)$

The *ALCM* Transf.: Almost Lifetime Optimal

Theorem 8.2.5 (*ALCM* Theorem)

The *ALCM* transformation is almost lifetime optimal, i.e.,

$$ALCM \in \mathcal{CM}_{ALtOpt}$$

Chapter 8.3

The *LCM* Transformation

Isolated Computation Points of a CM Transf.

Definition 8.3.1 (Isolation$_{CM}$)

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

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. $\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, n \in N.\ Latest\,(n) \Rightarrow$
$(Isolated_{CM}(n) \iff Isolated_{BCM}(n))$

The *LCM* Transformation

The *LCM* Transformation is defined by:

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

The *LCM* Transf.: Comp. & Lifetime Optimal

Theorem 8.3.3 (*LCM* Theorem)

The *LCM* transformation is lifetime optimal, i.e.,

$$LCM \in \mathcal{CM}_{LtOpt}$$

Corollary 8.3.4 (*LCM* Corollary)

The *LCM* transformation is computationally optimal, i.e.,

$$LCM \in \mathcal{CM}_{CmpOpt}$$

Chapter 8.4

Delayability and Isolation: The DFA Specifications

Delayability: The DFA Specification

...for a CM candidate expression t, $t \in \mathbf{T}$.

DFA Specification

- DFA lattice
  $\widehat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathbb{B}, \wedge, \vee, \leq, \mathit{false}, \mathit{true}) = \widehat{\mathbb{B}}$
- DFA functional
  $[\![\ ]\!]_{dl}^{t} : N \to (\mathbb{B} \to \mathbb{B})$, where
  $\forall\, n \in N\ \forall\, b \in \mathbb{B}.\ [\![\, n\,]\!]_{dl}^{t}(b) =_{df}$
  $(b \vee \mathit{Earliest}^{t}(n)) \wedge \neg \mathit{Comp}_{n}^{t}$
- Initial information: $\mathit{Earliest}^{t}(\mathbf{s}) \in \mathbb{B}$
- Direction of information flow: forward

Delayability Specification for t

- Specification: $\mathcal{S}_{G}^{dl,t} = (\widehat{\mathbb{B}}, [\![\ ]\!]_{dl}^{t}, \mathit{Earliest}^{t}(\mathbf{s}), \mathit{fw})$

A Hint to Appendix C

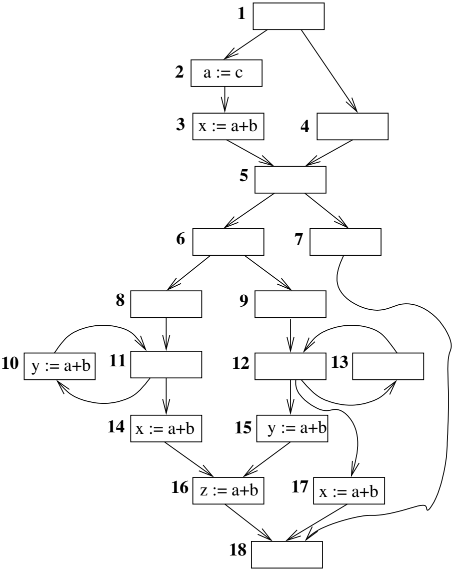
Appendix C presents

- the specialized versions of the *MaxFP* equation systems induced by $\mathcal{S}_G^{dl,t}$ and $\mathcal{S}_G^{iso,t}$, respectively, for
  - single instruction flow graphs (cf. Appendix C.1.3)
  - basic block flow graphs (cf. Appendix C.2.3).
- an illustrating example of the *ALCM* tranformation and the *LCM* transformation on basic block flow graphs.

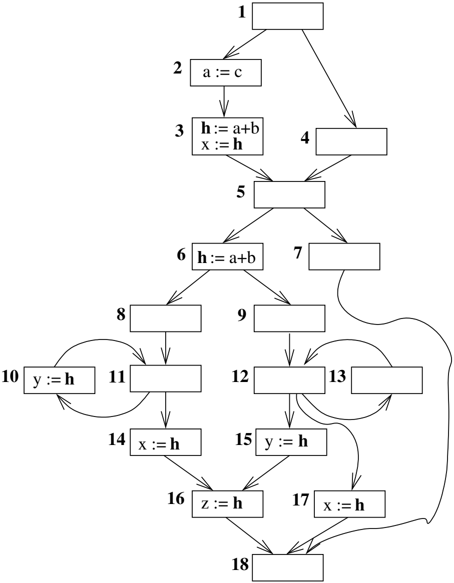
Chapter 8.5

Illustrating Example

The Original Program

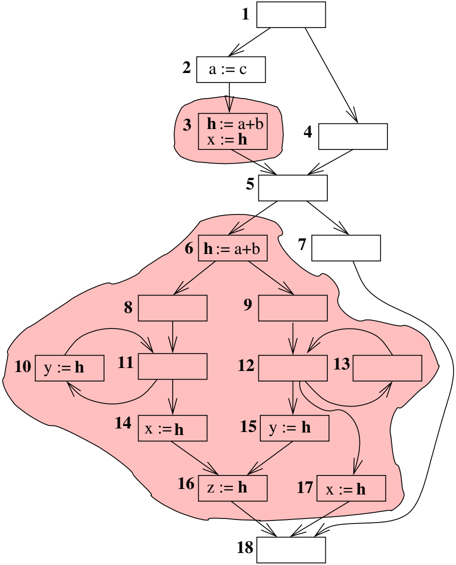


The Result of the *BCM* Transformation

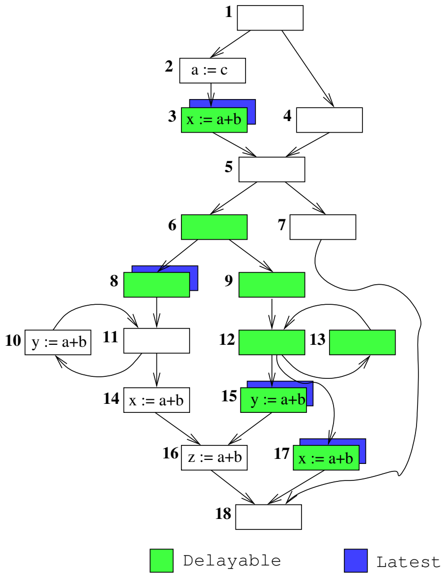


BCM Transf.: Achievements & Shortcomings

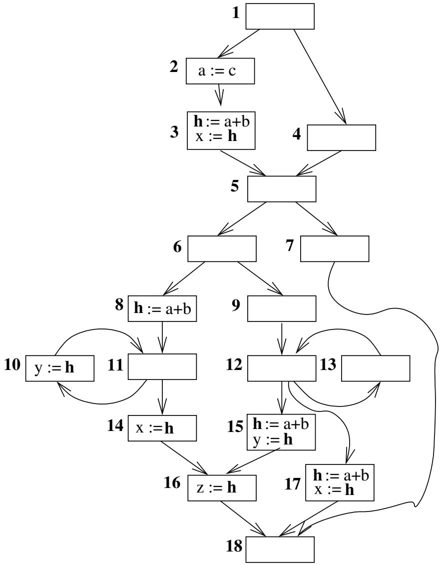
Computationally optimal but maximum register pressure.



Delayed and Latest Computation Points

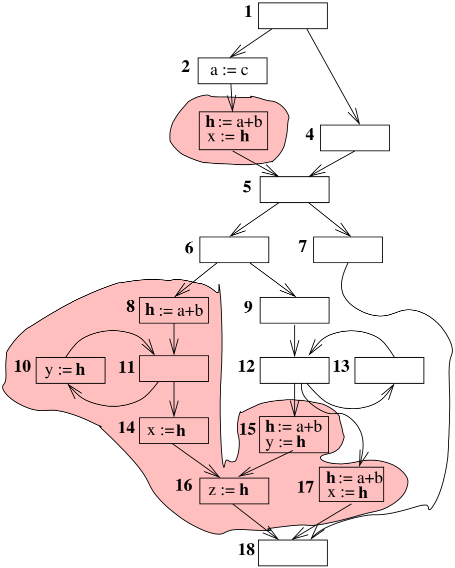


The Result of the *ALCM* Transformation

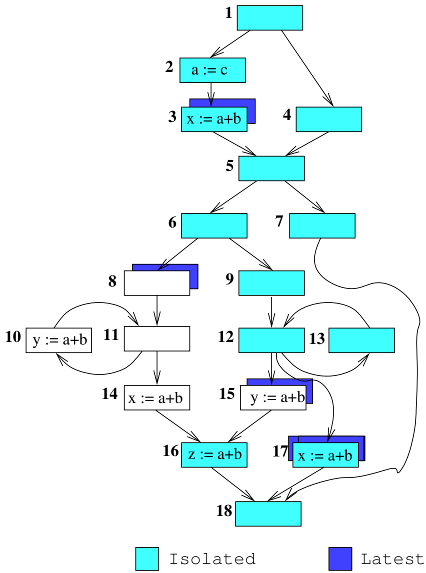


The *ALCM* Transformation: Achievements

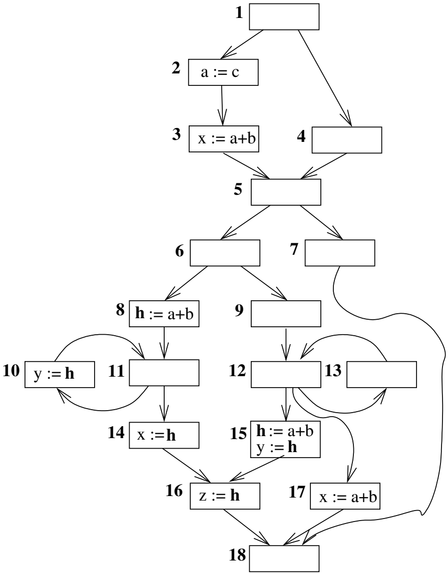
Comp. optimal with almost minimum register pressure.



Latest and Isolated Computation Points

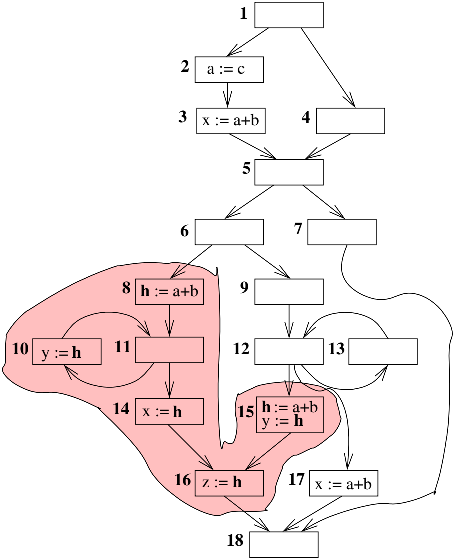


The Result of the *LCM* Transformation



The *LCM* Transformation: Achievements

Computationally optimal with minimum register pressure.



Chapter 8.6

References, Further Reading

Further Reading for Chapter 8 (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 9.5.3, The Lazy-Code-Motion Problem; Chapter 9.5.5, The Lazy-Code-Motion Algorithm)
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 10.3.2, Code Motion – Lazy Code Motion)
- Karl-Heinz Drechsler, Manfred P. Stadel. *A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION*. ACM SIGPLAN Notices 28(5):29-38, 1993.

Further Reading for Chapter 8 (2)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.

Further Reading for Chapter 8 (3)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retrospective: Lazy Code Motion*. In "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection", ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 13.3, Partial-Redundancy Elimination – Lazy Code Motion)
- Jean-Baptiste Tristan, Xavier Leroy. *Verified Validation of Lazy Code Motion*. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), 316-326, 2009.

Chapter 9

Sparse Code Motion

Chapter 9.1

Background and Motivation

Recall

Code Motion aims at

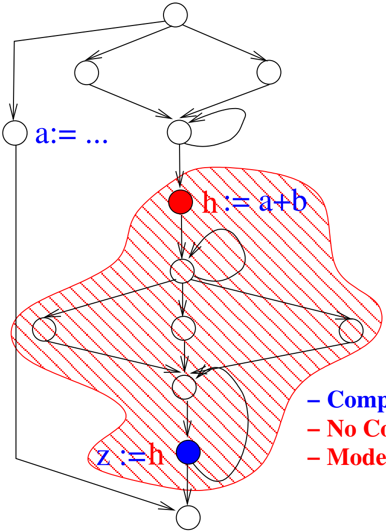
- eliminating unnecessary recomputations of values (e.g., *BCM*, *ALCM*, *LCM*)
- while simultaneously avoiding introducing unnecessary register pressure (e.g., *ALCM*, *LCM*)

Overall, code motion thus primarily aims at

- improving the runtime performance of a program.

However, there is more than Speed!

...code size, for example.



– **Computationally Optimal.**
– **No Code Replication.**
– **Moderate Register Pressure!**

Prioritization of Optimization Goals

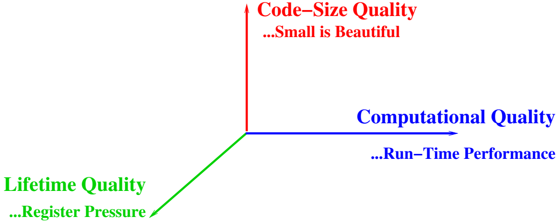
Recall that

- number of computations, register pressure, code size

can not be fully optimized at the same time (cf. Chapter 6).

In this chapter

- we present a CM algorithm taking user priorities into account!

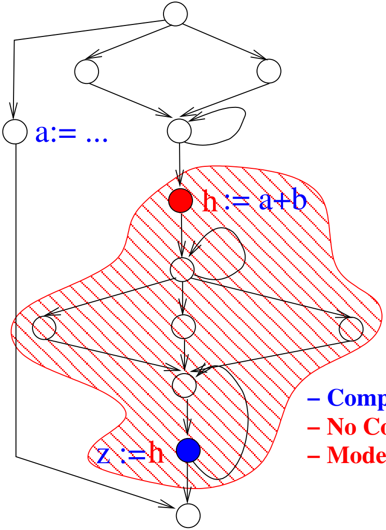


This algorithm, called Sparse Code Motion (*SpCM*)

- evolves as a modular extension of the *LCM* transf.

Sparse Code Motion

...can achieve the below result, if so desired:



– Computationally Optimal.
– No Code Replication.
– Moderate Register Pressure!

Chapter 9.1.1

The Embedded Systems Market

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.

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.

Think about

...domain-specific processors used in embedded systems:

- Telecommunication
  - Cellular phones, pagers,...
- Consumer electronics
  - MP3-players, cameras, game consoles, TVs,...
- Automative field
  - GPS navigation, airbags,...
- ...

Code for Embedded Systems (1)

...has high demands on

- Performance (often real-time demands)
- Code size (system-on-chip, on-chip RAM/ROM)
- Power consumption (batteries)
- . . .

For embedded systems

- Code size is often more critical than speed!

Code for Embedded Systems (2)

Typically, these demands are still addressed by

- Assembler programming
- Manual post-optimization

Shortcomings

- Error prone
- Delayed time-to-market

...problems getting more severe with increasing complexity.

Generally, there is

- a trend towards using high-level languages programming, particularly C, C++.

In View of this Trend

...how do classical compiler and optimizer technologies support the specific demands of code for embedded systems?

...unfortunately, only little.

As a Matter of Fact

Classical optimizations

- are tuned towards performance optimization
- are not code-size sensitive
- do not allow any control on their impact on the code size

This holds

...for code motion based optimizations, too, such as

- Partial redundancy elimination
- Partial dead-code elimination
  (cf. Lecture Course 185.276 Analysis and Verification)
- Partial redundant-assignment elimination
  (cf. Lecture Course 185.276 Analysis and Verification)
- Strength reduction
- ...

Recalling the Essence of CM in General

CM can conceptually be considered a two-stage process:

1. Expression Hoisting
   ...hoisting computations to "earlier" safe computation points
2. Totally Redundant Expression Elimination
   ...eliminating computations, which become totally redundant by expression hoisting

Recalling the Essence of *LCM*

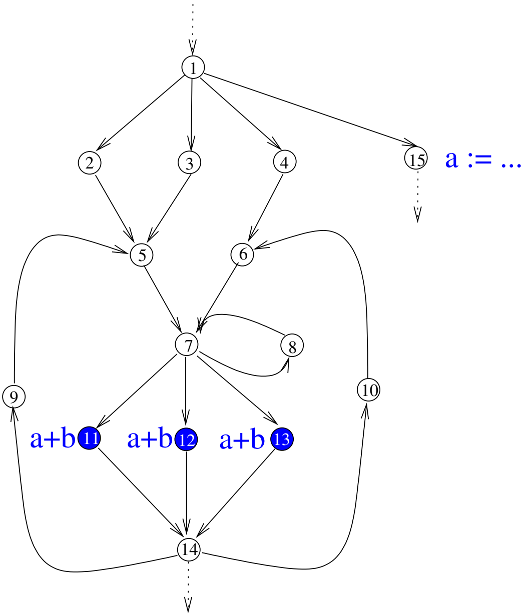
LCM can conceptually be considered the result of a two-stage process, too:

1. Hoisting Expressions
   ...to their "earliest" safe computation points
2. Sinking Expressions
   ...from their "earliest" safe computation points to their "latest" safe still computationally optimal computation points

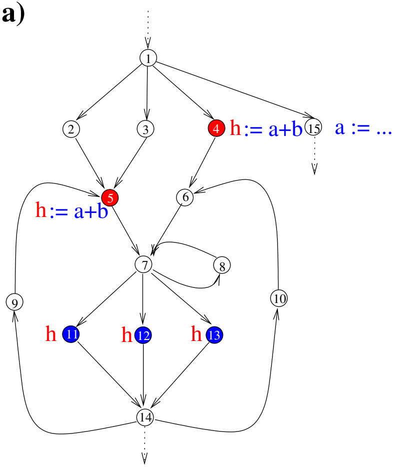
Chapter 9.2

Running Example

Running Example: The Original Program

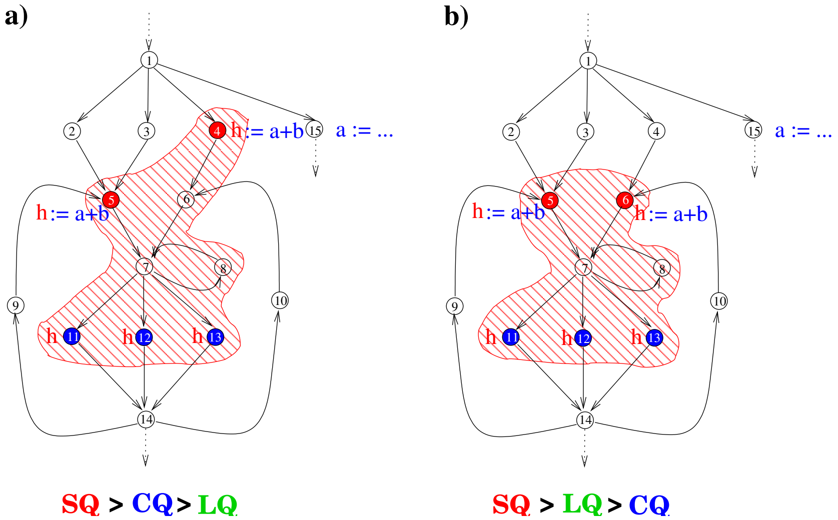


Running Example: Two Optimization Variants



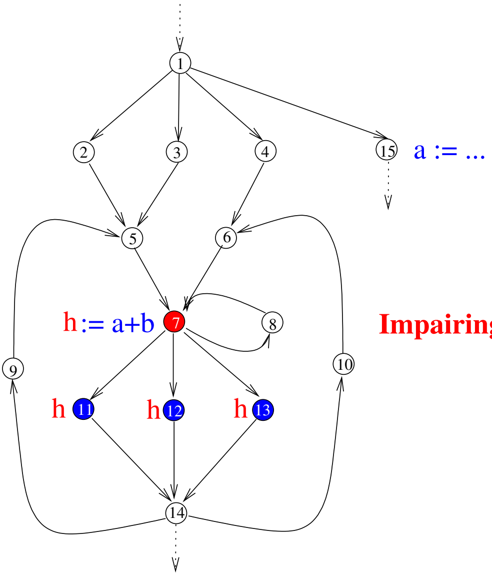
Two Code–size Optimal Programs

Running Example: Optimization Priorities



Running Example: Undesired Transformation

Recall: The below transformation is not desired!



Impairing!

Chapter 9.3

Code-size Sensitive Code Motion

Code-Size Sensitive Code Motion

$\rightsquigarrow$ 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?

$\rightsquigarrow$ The Solution: A new View to Code Motion

...consider CM as a trade-off problem: Exchange original computations for newly inserted ones!

$\rightsquigarrow$ The Clou: Use Graph Theory!

...reduce the trade-off problem to the computation of tight sets in bipartite graphs based on maximum matchings!

We postpone but keep in mind

...that we have to answer:

- Where are computations to be inserted and where are original computations to be replaced?

...and to prove:

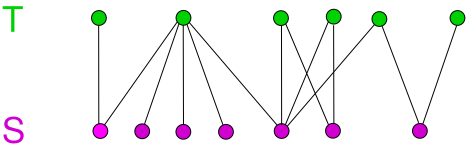
- Why is this correct (i.e., semantics preserving)?
- What is the impact on the code size?
- Why is this "optimal" wrt a given prioritization of goals?

For each of these questions we will provide a specific theorem that yields the corresponding answer!

Chapter 9.3.1

Graph-theoretical Preliminaries

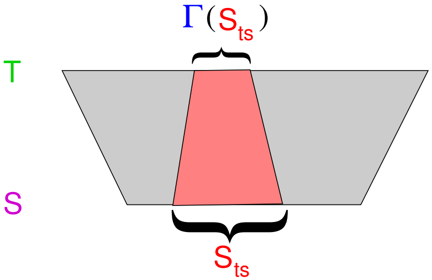
Bipartite Graphs



Tight Set

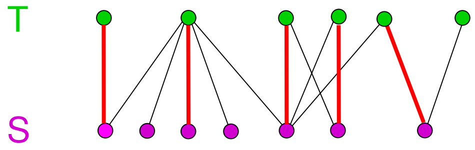
...of a bipartite graph $(S \cup T, E)$: Subset $S_{ts} \subseteq S$ w/

$$\forall\, S' \subseteq S.\ |S_{ts}| - |\Gamma(S_{ts})| \ \geq \ |S'| - |\Gamma(S')|$$



Two Variants: (1) Largest tight sets, (2) Smallest tight sets

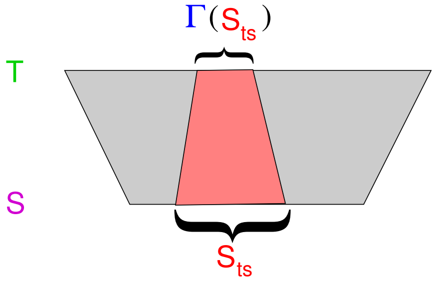
Bipartite Graphs



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

Obviously

...we can make use of off-the-shelve algorithms from graph theory in order to compute

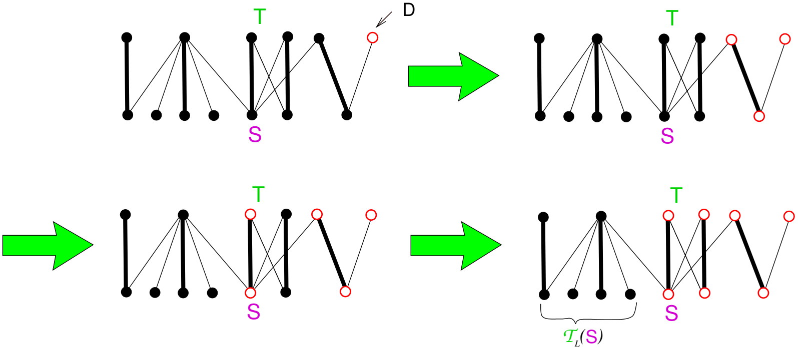
- Maximum matchings and
- Tight sets

This way the PRE problem boils down to

- constructing the bipartite graph that models the problem!

Computing Largest/Smallest Tight Sets

...based on maximum matchings:



Computing Largest Tight Sets

Algorithm 9.3.1.1 (Computing Largest Tight Sets)

Input: A bipartite graph $(S \dot{\cup} T, E)$, a maximum matching M.

Output: The largest tight set $\mathcal{T}_{LaTS}(S) \subseteq S$.

```
S_M := S; D := {t ∈ T | t is unmatched};
WHILE D ≠ ∅ DO
    choose some x ∈ D;  D := D \ {x};
    IF x ∈ S
       THEN S_M := S_M \ {x};
            D := D ∪ {y | {x,y} ∈ M}
       ELSE D := D ∪ (Γ(x) ∩ S_M)
    FI
OD;
T_LaTS(S) := S_M
```

Computing Smallest Tight Sets

Algorithm 9.3.1.2 (Computing Smallest Tight Sets)

Input: A bipartite graph $(S \dot{\cup} T, E)$, a maximum matching M.

Output: The smallest tight set $\mathcal{T}_{SmTS}(S) \subseteq S$.

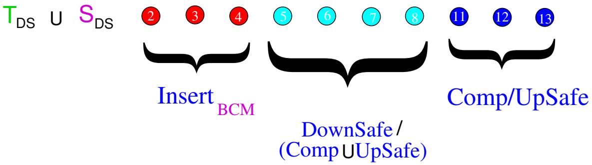
$S_M := \emptyset$; $A := \{s \in S \mid s \text{ is unmatched}\}$;
WHILE $A \neq \emptyset$ DO
    choose some $x \in A$; $A := A \setminus \{x\}$;
    IF $x \in S$
        THEN $S_M := S_M \cup \{x\}$;
            $A := A \cup (\Gamma(x) \setminus S_M)$
        ELSE $A := A \cup \{y \mid \{x, y\} \in M\}$
    FI
OD;
$\mathcal{T}_{SmTS}(S) := S_M$

Chapter 9.3.2

Modelling the Problem

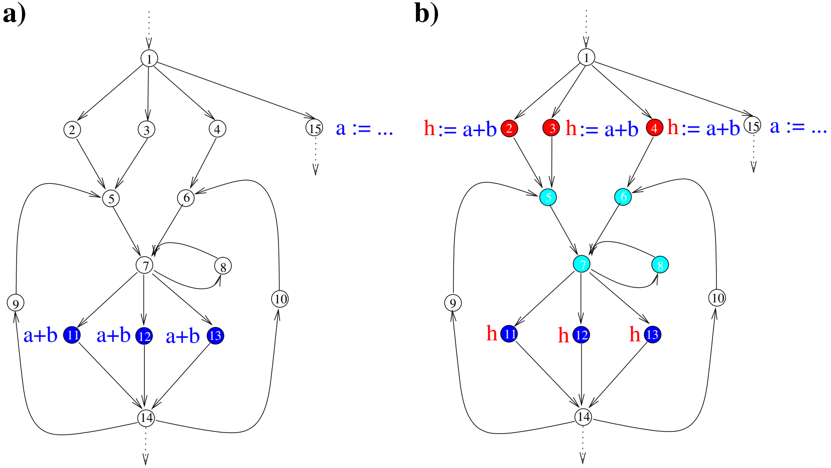
Modelling the Trade-off Problem

The Set of Nodes



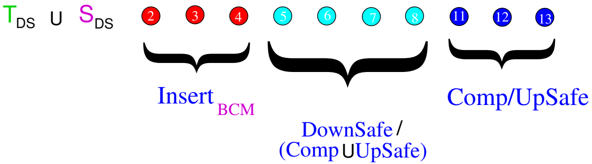
The Set of Edges...

The Set of Nodes

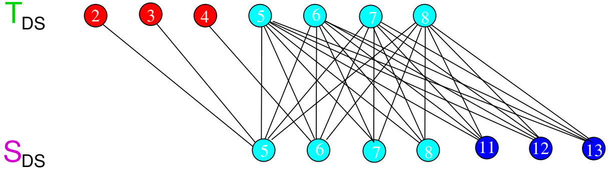


Modelling the Trade-off Problem

The Set of Nodes



The Bipartite Graph



The Set of Edges

...$\forall\, n \in S_{DS}\ \forall\, m \in T_{DS}$.

$$\{n, m\} \in E_{DS} \iff_{df} m \in \textbf{Closure}(pred(n))$$

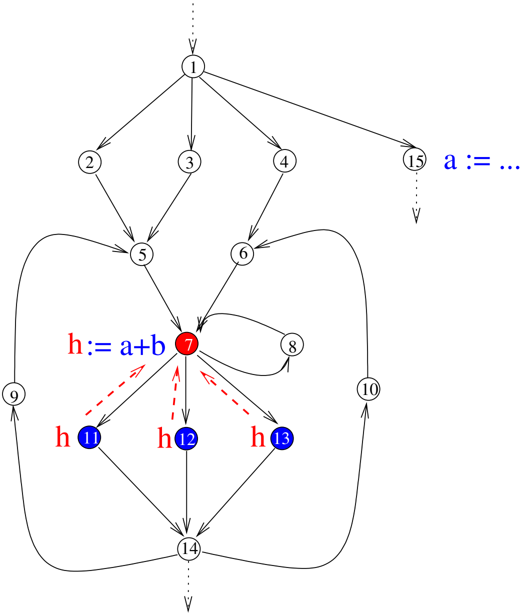
Down-Safety Closures

Definition 9.3.2.1 (Down-Safety Closure)

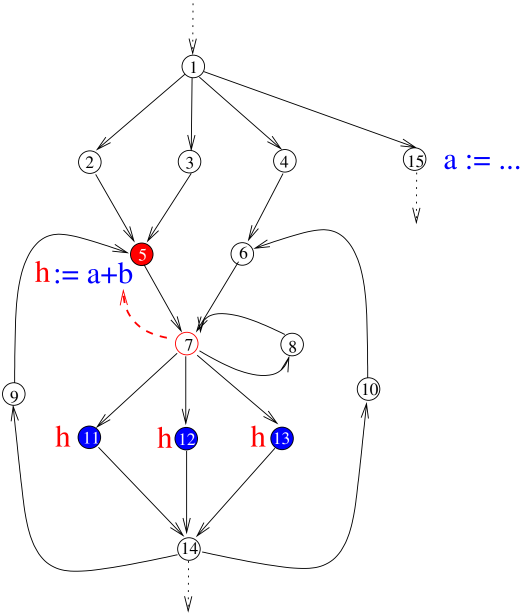
Let $n \in DownSafe/Upsafe$. Then the Down-Safety Closure $Closure(n)$ is the smallest set of nodes such that

1. $n \in Closure(n)$
2. $\forall\, m \in Closure(n) \setminus Comp.\ succ(m) \subseteq Closure(n)$
3. $\forall\, m \in Closure(n).\ pred(m) \cap Closure(n) \neq \emptyset \;\Rightarrow$ $pred(m) \setminus UpSafe \subseteq Closure(n)$

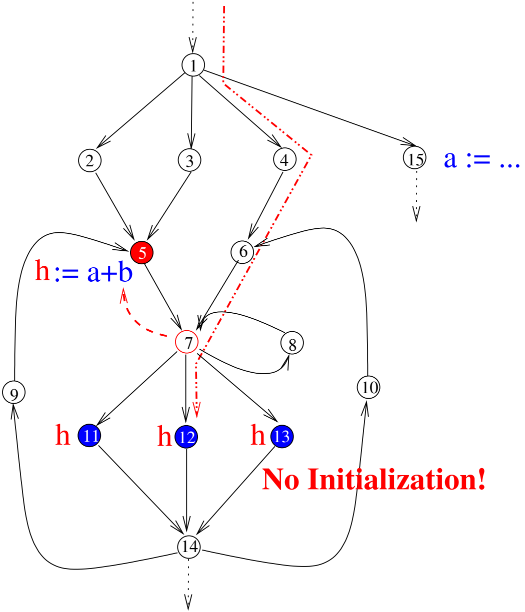
Down-Safety Closures: The Intuition (1)



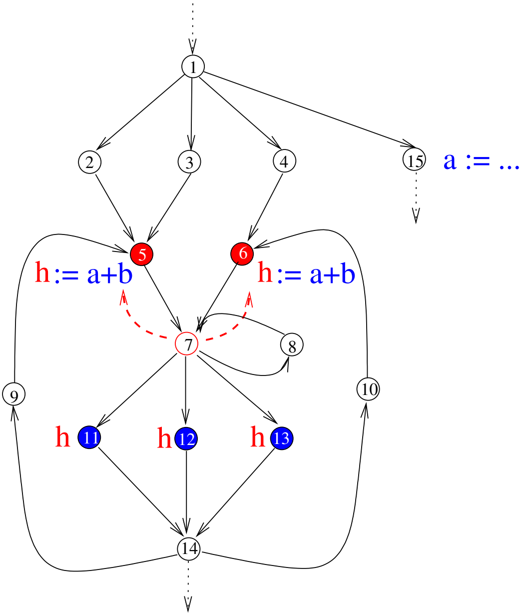
Down-Safety Closures: The Intuition (2)



Down-Safety Closures: The Intuition (3)



Down-Safety Closures: The Intuition (4)



This intuition

...is condensed in the notion of down-safety closures. Recall:

Definition 9.3.2.1 (Down-Safety Closure) – recalled

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. $\forall\, m \in Closure(n).\ pred(m) \cap Closure(n) \neq \emptyset \;\Rightarrow$
   $pred(m) \setminus UpSafe \subseteq Closure(n)$

Down-Safety Regions

...lead to a characterization of semantics-preserving PRE transformations via their insertion points.

Definition 9.3.2.2 (Down-Safety Region)

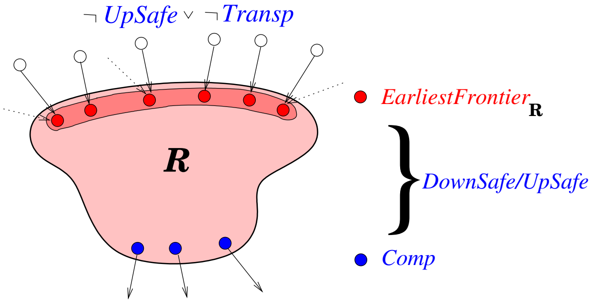
A set $\mathcal{R} \subseteq N$ of nodes is a down-safety region iff

1. $\mathit{Comp} \backslash \mathit{UpSafe} \subseteq \mathcal{R} \; \subseteq \; \mathit{DownSafe} \backslash \mathit{UpSafe}$
2. $\mathit{Closure}(\mathcal{R}) \; = \; \mathcal{R}$

Fundamental

Theorem 9.3.2.3 (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.

Chapter 9.3.3

Main Results: Correctness and Optimality

The Key Questions

...regarding correctness and optimality:

1. Where to insert computations, why is it correct?
2. What is the impact on the code size?
3. Why is the result optimal, i.e., code-size minimal?

...three theorems will answer one of these questions each.

Main Results / 1st Key Question

Question: Where to insert computations, why is it correct?

Answer: At the earliestness frontier of the DS-region induced by the tight set.

Theorem 9.3.3.1 (Tight Sets: Insertion Points)

Let $TS \subseteq S_{DS}$ be a tight set. Then we have:

$$\mathcal{R}_{TS} =_{df} \Gamma(TS) \cup (Comp \backslash UpSafe)$$

is a down-safety region w/ $Body_{\mathcal{R}_{TS}} = TS$

Correctness of the *SpCM* Transformation

- An immediate corollary of Theorem 9.3.3.1 and the Initialization Theorem 9.3.2.3

Main Results / 2nd Key Question

Question: What is the impact on the code size?

Answer: The difference between the number of inserted and replaced computations.

Theorem 9.3.3.2 (Down-Safety Reg.: Space Gain)

Let $\mathcal{R}$ be a down-safety region with

$$Body_{\mathcal{R}} =_{df} \mathcal{R} \setminus EarliestFrontier_{\mathcal{R}}$$

Then we have:

- Space Gain by Inserting at $EarliestFrontier_{\mathcal{R}}$:

  $$|Comp \setminus UpSafe| - |EarliestFrontier_{\mathcal{R}}| = |Body_{\mathcal{R}}| - |\Gamma(Body_{\mathcal{R}})| \; {}_{df}= defic(Body_{\mathcal{R}})$$

Main Results / 3rd Key Question

Question: Why is the result optimal, i.e., code-size minimal?

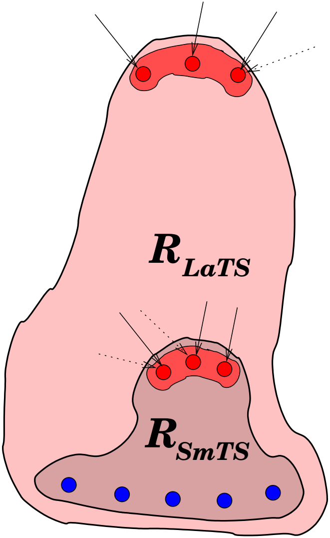
Answer: Due to a property inherent to tight sets (non-negative deficiency!).

Theorem 9.3.3.3 (Optimality: Transformation)

Let $TS \subseteq S_{DS}$ be a tight set.

- Insertion Points:
  $Insert_{SpCM} =_{df} EarliestFrontier_{\mathcal{R}_{TS}} = \mathcal{R}_{TS} \setminus TS$
- Space Gain:
  $defic(TS) =_{df} |TS| - |\Gamma(TS)| \geq 0$ max.

Largest vs. Smallest Tight Sets: The Impact



- $EarliestFrontier_{\mathbf{R}_{LaTS}}$

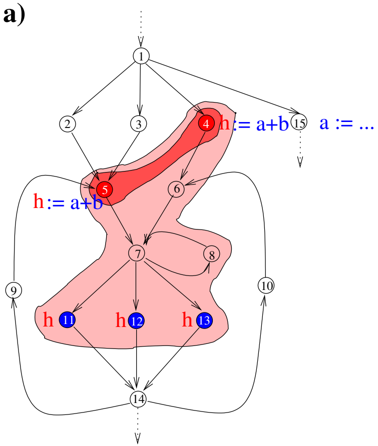
Largest tight sets favor
Computational Quality
→ **Earliestness Principle**

- $EarliestFrontier_{\mathbf{R}_{SmTS}}$

Smallest tight sets favor
Lifetime Quality
→ **Latestness Principle**

- *Comp*

The Impact illustrated on the Running Exam.



Largest Tight Set

(SQ > CQ)

Earliestness Principle

Smallest Tight Set

(SQ > LQ)

Latestness Principle

Chapter 9.4

The *SpCM* Transformation

The *SpCM* Transformation at a Glance

Preprocess

- **Optional:** Perform LCM (3 GEN/KILL–DFAs)
- Compute Predicates of BCM for G resp. LCM(G) (2 GEN/KILL–DFAs)

↓

Main Process

Reduction Phase

- Construct Bipartite Graph
- Compute Maximum Matching

↓

Optimization Phase

- Compute Largest/Smallest Tight Set
- Determine Insertion Points

Chapter 9.5

The Cookbook: Recipes for Code Motion

The Cookbook: CM Recipes for Prioritization

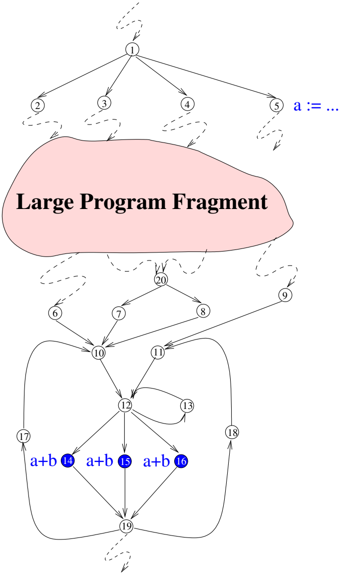
Choice of Priority	Apply	To	Using	Yields	Auxiliary Information Required
$\mathcal{LQ}$	Not meaningful: The identity, i.e., G itself is optimal!				
$\mathcal{SQ}$	Subsumed by $\mathcal{SQ} > \mathcal{CQ}$ and $\mathcal{SQ} > \mathcal{LQ}$!				
$\mathcal{CQ}$	**BCM**	G			UpSafe(G), DownSafe(G)
$\mathcal{CQ} > \mathcal{LQ}$	**LCM**	G		**LCM**(G)	UpSafe(G), DownSafe(G), Delay(G)
$\mathcal{SQ} > \mathcal{CQ}$	**SpCM**	G	**Largest tight set**	$\mathbf{SpCM}_{LTS}$(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**($\mathbf{SpCM}_{LTS}$(G))	**Smallest tight set**		UpSafe(G), DownSafe(G), Delay($\mathbf{SpCM}_{LTS}$(G)), UpSafe(**DL**($\mathbf{SpCM}_{LTS}$(G))), DownSafe(**DL**($\mathbf{SpCM}_{LTS}$(G)))

Chapter 9.6

Illustrating Example

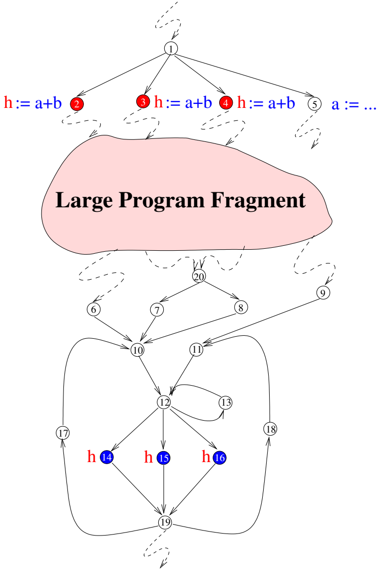
Sparse Code Motion: Flexible and Powerful

The original program:



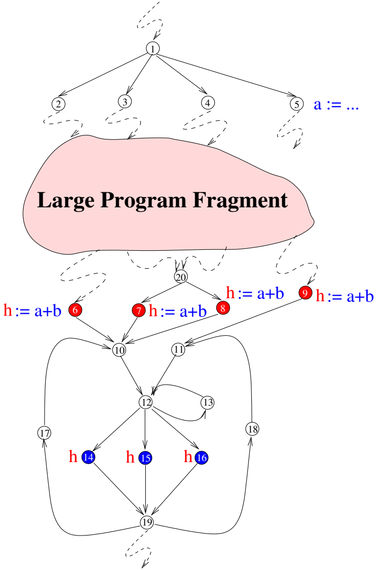
SpCM: Computationally Optimal, 2 DFAs

BCM: A computationally optimal program ($\mathcal{CQ}$)



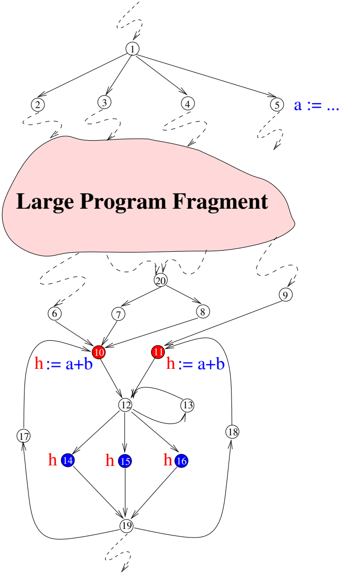
SpCM: Comp. and Lifetime Optimal, 4 DFAs

LCM: A computationally & lifetime opt. program ($\mathcal{CQ} > \mathcal{LQ}$)



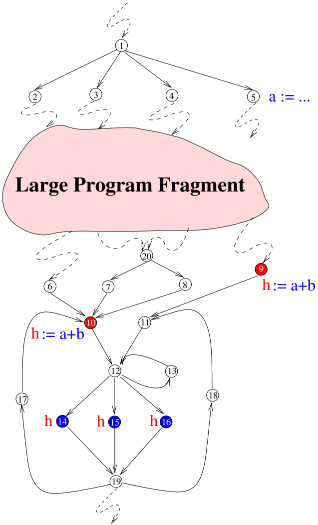
SpCM: Lifetime-Best Code-Size Optimal

SpCM: A code-size & lifetime opt. program ($\mathcal{SQ} > \mathcal{LQ}$)



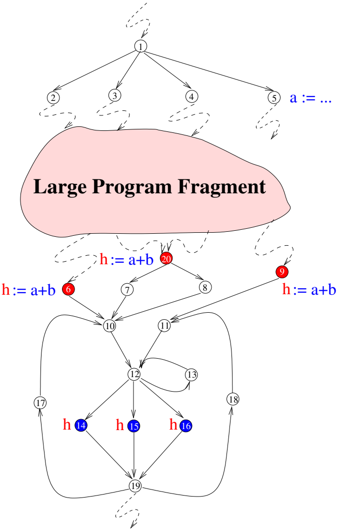
SpCM: Lifet.-Best Comp. Code-Size Optimal

SpCM: A computationally & lifetime best code-size optimal program ($\mathcal{SQ} > \mathcal{CQ} > \mathcal{LQ}$)



SpCM: Lifet.&Code-Size Best Comp. Optimal

SpCM: A lifetime& code-size best computationally optimal program ($\mathcal{CQ} > \mathcal{SQ} > \mathcal{LQ}$)



Chapter 9.7

References, Further Reading

Further Reading for Chapter 9

- Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Sparse Code Motion*. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), 170-183, 2000.
- Bernhard Scholz, R. Nigel Horspool, Jens Knoop. *Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination*. Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), ACM SIGPLAN Notices 39(7):221-230, 2004.

Chapter 10

Code Motion: Summary, Looking Ahead

Chapter 10.1

Summary: Roots and Relevance of Code Motion

Chapter 10.1.1

On the Roots and History of Code Motion

On the Origins & History of CM ($\equiv$ PRE) (1)

- 1958: A first glimpse of PRE
  - $\leadsto$ Ershov's work on "On Programming of Arithmetic Operations."
- $<$ 1979: Structurally Restricted PRE Techniques
  - $\leadsto$ Totally redundant expression elimination (TRE), loop invariant code motion (LICM)
- 1979: The origin of modern PRE
  - $\leadsto$ Morel and Renvoise's groundbreaking work on PRE
- $<$ ca. 1992: Heuristic improvements of the PRE algorithm of Morel and Renvoise
  - $\leadsto$ Dhamdhere [1988, 1991]; Drechsler, Stadel [1988]; Sorkin [1989]; Dhamdhere, Rosen, Zadeck [1992], Briggs, Cooper [1994],...

On the Origins & History of CM ($\equiv$ PRE) (2)

- 1992: *BCM* and *LCM* [Knoop Rüthing, Steffen (PLDI'92)]
  - $\rightsquigarrow$ *BCM* first to achieve computational optimality based on the earliestness principle
  - $\rightsquigarrow$ *LCM* first to achieve computational optimality with minimum register pressure based on the latestness principle
  - $\rightsquigarrow$ *BCM*, *LCM* first to be purely unidirectional
  - $\rightsquigarrow$ first to be rigorously proven correct and optimal
- 2000: *SpCM*: The origin of code-size sensitive PRE [Knoop, Rüthing, Steffen (POPL 2000)]
  - $\rightsquigarrow$ first to be code-size sensitive
  - $\rightsquigarrow$ first to allow users prioritization of optimization goals
  - $\rightsquigarrow$ rigorously be proven correct and optimal
  - $\rightsquigarrow$ first to bridge the gap between compilation for general purpose processors and embedded systems

On the Origins & History of CM ($\equiv$ PRE) (3)

- Since ca. 1997: A new strand of research on PRE
  - $\rightsquigarrow$ Speculative PRE: Gupta, Horspool, Soffa, Xue, Scholz, Knoop,...
- 2005: A fresh look at PRE (as maximum flow problem)
  - $\rightsquigarrow$ Unifying PRE and Speculative PRE [Xue, Knoop (CC 2006)]

These days, lazy code motion is the

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

Chapter 10.1.2

On the Relevance of Code Motion

Code Motion is Relevant and Challenging (1)

Why is it worthwhile and rewarding to investigate CM?

Because code motion is

- general: A family of optimizations rather than a single optimization.
- well understood: Algorithms, which are proven correct and optimal.
- truly classical: Looks back to a long history originated by
  - Etienne Morel, Claude Renvoise. Global Optimization by Suppression of Partial Redundancies. Communications of the ACM 22(2):96-103, 1979.
  - Ken Kennedy. Safety of Code Motion. International Journal of Computer Mathematics 3(2-3):117-130, 1972.
  - Andrei P. Ershov. On Programming of Arithmetic Operations. Communications of the ACM 1(8):3-6, 1958.

Code Motion is Relevant and Challenging (2)

In particular, code motion is

- relevant: Widely used in practice because of its power.

Last but not least, code motion is

- challenging: Conceptually simple but exhibits a variety of thought provoking phenomenons and pitfalls.

Some of these challenges we are going to illustrate next.

Code Motion Reconsidered

Traditionally:

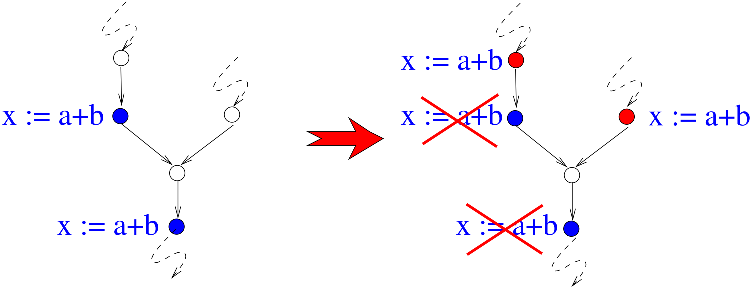
- Code (C) means expressions.
- Motion (M) means hoisting.
- CM means partially redundant expression elimination.

But:

- CM is more than hoisting of expressions and PR(E)E!

Assignments

...are code, too, of course.

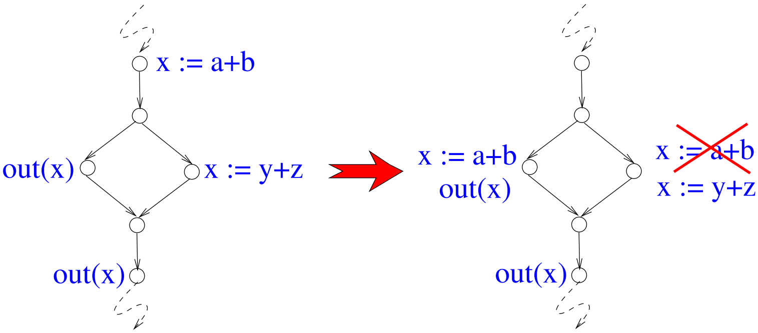


In this example, CM means

- partially redundant assignment elimination (PRAE).

Assignments

...can be hoisted like expressions but conversely, also be sunk!



In this example, CM means

- partially dead assignment (or code) elimination (PDCE).

Design Space of CM Algorithms (1)

In general

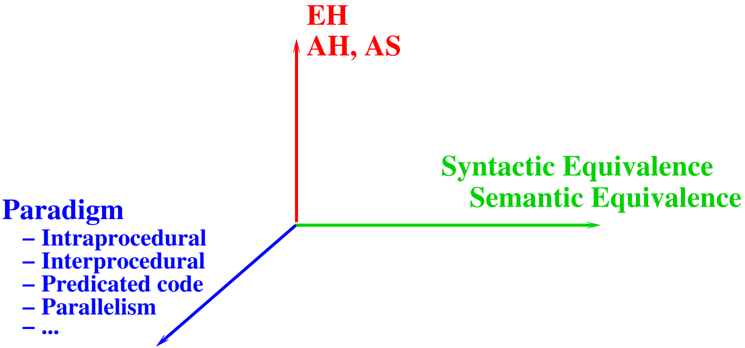
- Code means expressions/assignments.
- Motion means hoisting/sinking.

Code / Motion	Hoisting	Sinking
Expressions	EH	n.a.
Assignments	AH	AS

....which spans a first set of dimensions for designing code motion algorithms.

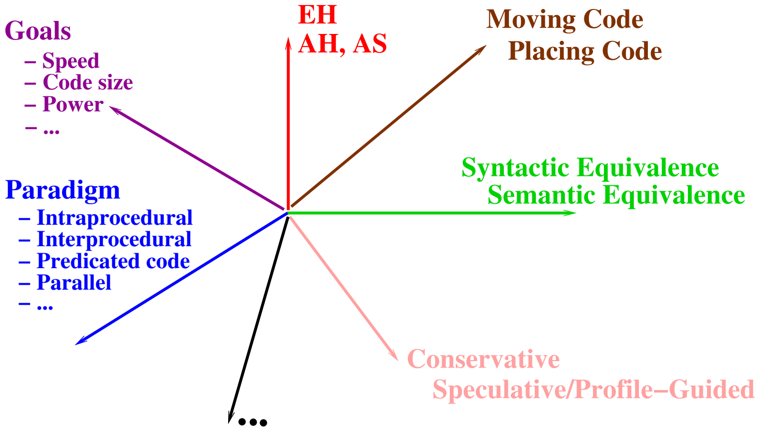
Design Space of CM Algorithms (2)

...but there are more dimensions for the design of code motion algorithms:



Design Space of CM Algorithms (3)

...and even more:



Chapter 10.2

Looking Ahead: Value Numbering

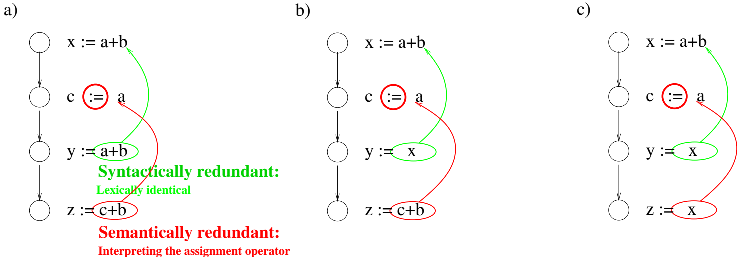
Chapter 10.2.1

(Local) Value Numbering

(Local) Value Numbering

...eliminating semantically redundant computations in basic blocks.

Illustrating Example:

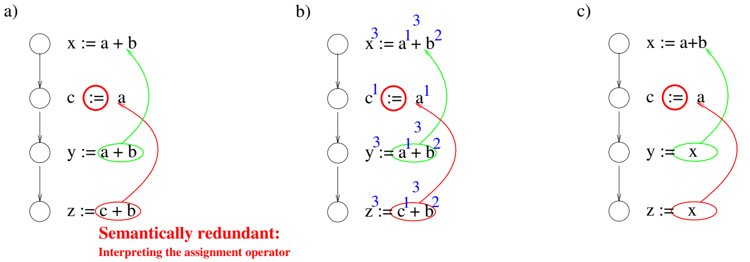


The Original Program **After Syntactic PRE/CM** **After Semantic PRE/CM**

Local Value Numbering

Intuitively

- value numbering works by assigning terms a so-called value number representing symbolically their values.
- same value number implies same value.



The Original Program | After Value Numbering | After Semantic PRE/CM

...has been described early by

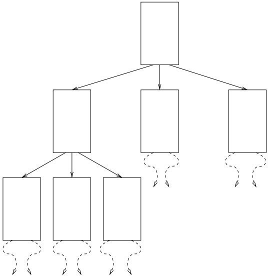
- John Cocke and Jacob T. Schwartz in 1970.

Local Value Numbering

...can straightforward be extended to

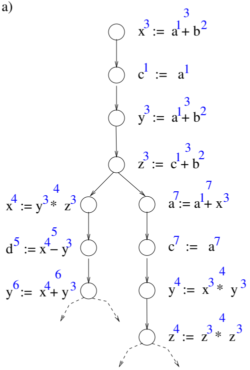
- extended basic blocks (i.e., trees of basic blocks).

Illustrating Example:



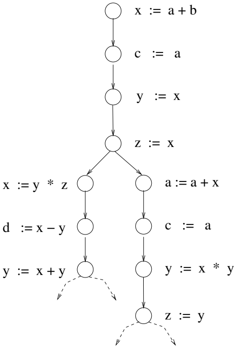
Extended Basic Block – A Tree of Basic Blocks

a)



After Value Numbering

b)



After Semantic PRE/CM

References for Chapter 10.2.1

John Cocke, Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 2nd Revised Version, 771 pages, 1970. (Chapter 6, Optimization Methods for Algebraic Languages)

Chapter 10.2.2

Global Value Numbering: Semantic Code Motion

Motivation

Syntactic PRE (or Syntactic Code Motion), e.g., *BCM, LCM*

- \+ Global: Works for whole programs.
- – Equivalence: Limited to lexical identity.

(Local) Value Numbering

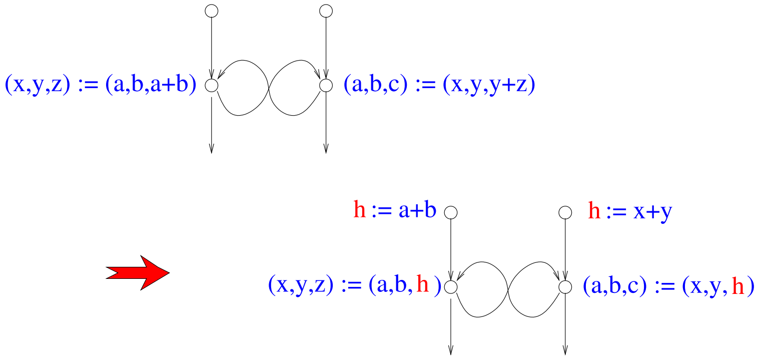
- \+ Equivalence: Captures semantic equivalence of terms.
- – Local: Limited to basic blocks.

Global Value Numbering (or Semantic Code Motion)

- \+ combines the best features of syntactic PRE and value numbering
- \+ while avoiding their weaknesses.

Global Value Numbering: Global Semantic Code Motion (SCM)

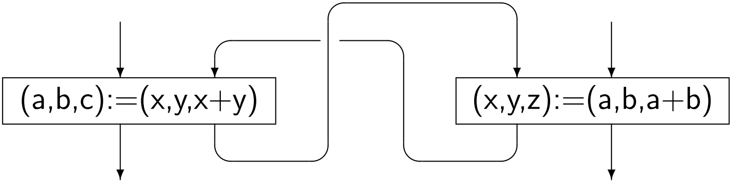
...extending the idea of value numbering to whole programs.



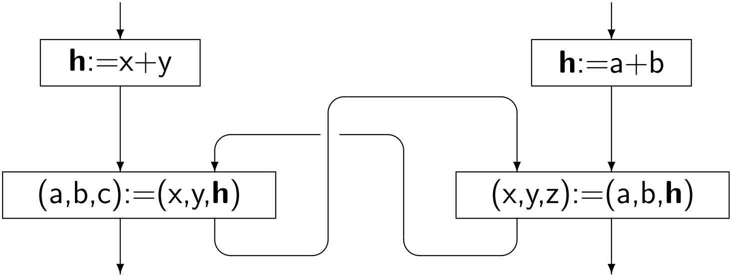
Bernhard Steffen (TAPSOFT'87)

Semantic CM: Illustrating the Essence (1)

The running example:



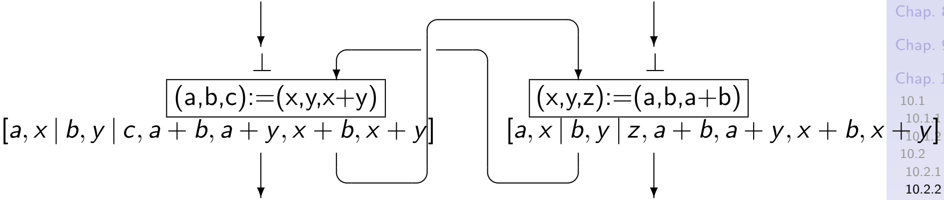
The optimized program:



Semantic CM: Illustrating the Essence (2)

Stage 1: The Analysis Phase

Step 1.1: Determining semantically equivalent terms (wrt the Herbrand interpretation).



Semantic CM: Illustrating the Essence (3)

Step 1.2: Enlarging the set of term equivalences syntactically represented.

$[a \mid b \mid c \mid x \mid y \mid z \mid a+b \mid x+y]$

$\boxed{\text{(a,b,c):=(x,y,x+y)}}$

$[a, x \mid b, y \mid c, a+b, a+y, x+b, x+y]$

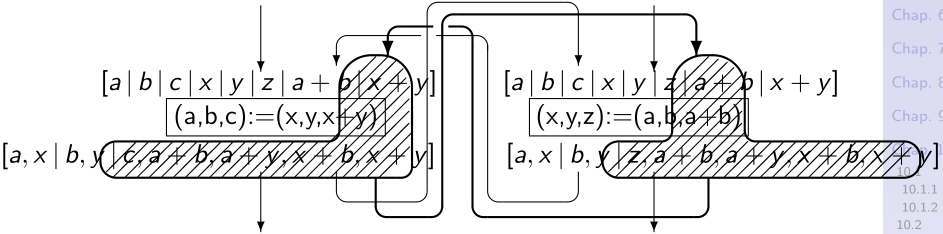
$[a \mid b \mid c \mid x \mid y \mid z \mid a+b \mid x+y]$

$\boxed{\text{(x,y,z):=(a,b,a+b)}}$

$[a, x \mid b, y \mid z, a+b, a+y, x+b, x+y]$

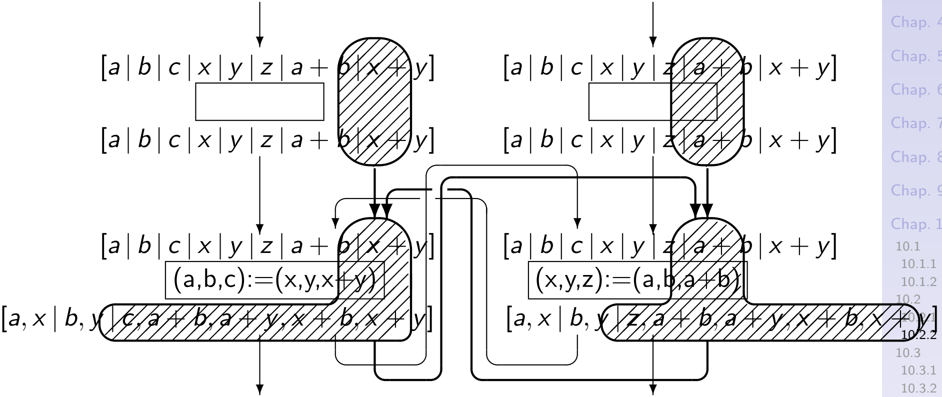
Semantic CM: Illustrating the Essence (4)

Step 1.3: Constructing the value flow graph.



Semantic CM: Illustrating the Essence (5)

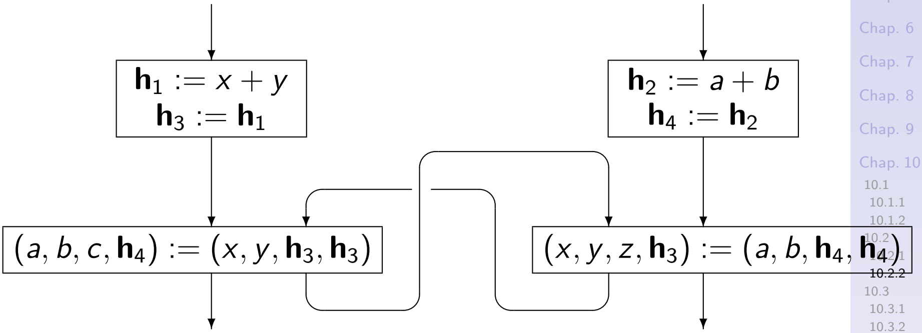
Step 1.3: The value flow graph, displaying a larger fragment.



Semantic CM: Illustrating the Essence (6)

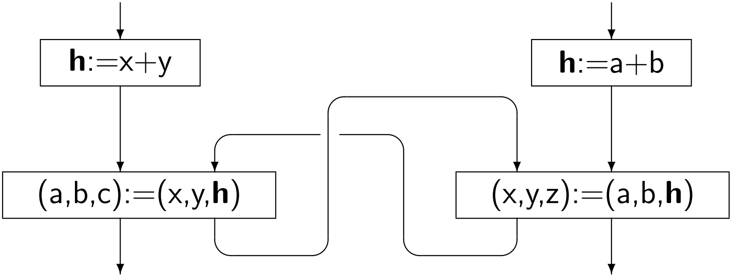
Stage 2: The Transformation Phase

Step 2.1: The Semantic Code Motion Optimization.



Semantic CM: Illustrating the Essence (7)

Step 2.1: Variable subsumption yields the final optimization.



...for details see: Steffen, Knoop, Rüthing (ESOP'90).

Semantic CM: Computing Insertion Points (1)

...by analysing the value flow graph.

Algorithm 10.2.2.1 (Computing Insertion Points)

The Frame Conditions (Local Properties):

$$\mathbf{ANTLOC}(\nu) \iff \nu{\downarrow_1} \cap \mathit{Terms}(\,\mathcal{N}(\nu)\,) \neq \emptyset$$

$$\mathbf{AVIN}(\nu) = \mathbf{PPIN}(\nu) = \mathit{false} \ \text{ if } \ \nu \in \mathit{VFN}_{\mathbf{s}}$$

$$\mathbf{PPOUT}(\nu) = \mathit{false} \ \text{ if } \ \nu \in \mathit{VFN}_{\mathbf{e}}$$

where

- $\nu{\downarrow_1}$: the projection of ν to its first component.
- $\mathcal{N}(\nu)$: the node of the flow graph ν is associated with.
- $\mathit{VFN}_{\mathbf{s}}$, $\mathit{VFN}_{\mathbf{e}}$: the set of start and end nodes of the VFG.
- $\mathit{Terms}(n)$: the set of terms of the assignment at node n.

Semantic CM: Computing Insertion Points (2)

The Fixed Point Equations (Global Properties):

$$\mathbf{AVIN}(\nu) \iff \prod_{\kappa' \in pred(\kappa)} \mathbf{AVOUT}(\nu')$$

$$\mathbf{AVOUT}(\nu) \iff \mathbf{AVIN}(\nu) \vee \mathbf{PPOUT}(\nu)$$

$$\mathbf{PPIN}(\nu) \iff \mathbf{AVIN}(\nu) \wedge (\,\mathbf{ANTLOC}(\nu) \vee \mathbf{PPOUT}(\nu)\,)$$

$$\mathbf{PPOUT}(\nu) \iff \prod_{m \in succ(\mathcal{N}(\kappa))} \sum_{\substack{\kappa' \in succ(\kappa) \\ \mathcal{N}(\kappa') = m}} \mathbf{PPIN}(\kappa')$$

The Insertion Points:

$$\mathbf{INSERT}(\kappa) =_{df} \mathbf{PPOUT}(\kappa) \wedge \neg \mathbf{PPIN}(\kappa)$$

Semantic CM: Main Results

Theorem 10.2.2.2 (Optimality of Analysis)

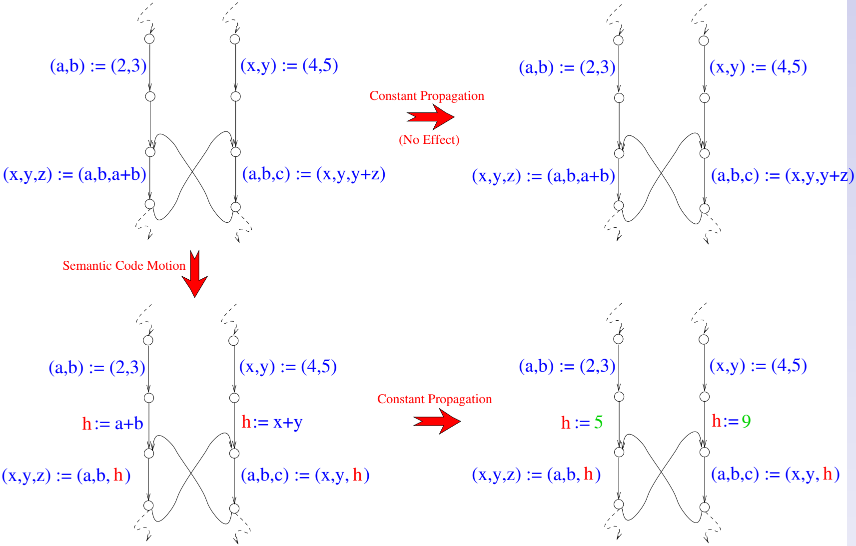
Given an arbitrary flow graph, the analysis stage terminates with a flow graph annotation which exactly characterizes all equivalences of program terms wrt the Herbrand interpretation.

Theorem 10.2.2.3 (Optimality of Transformation)

Every flow graph transformed by the two stage algorithm (in the full variant) is Herbrand optimal.

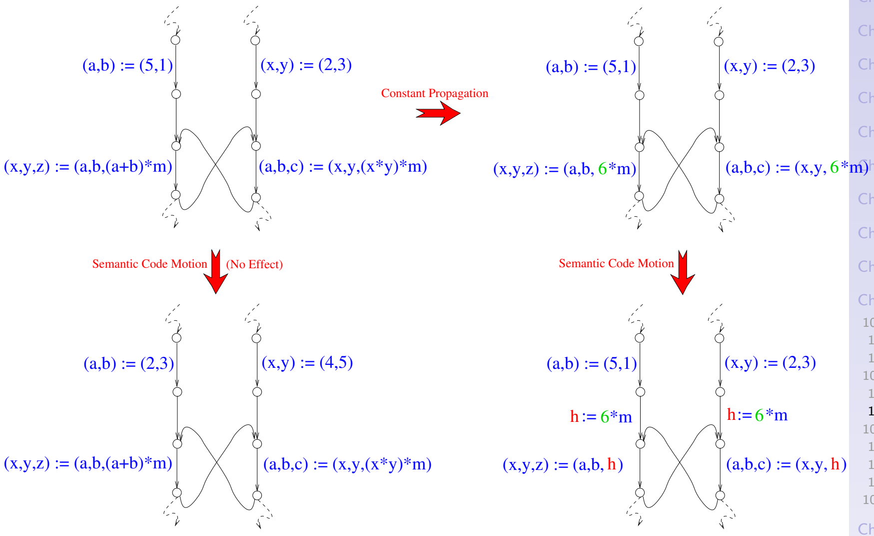
Semantic Code Motion, Constant Propagation

Recall:



Semantic Code Motion, Constant Propagation

Recall:



References for Chapter 10.2.2

Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Proceedings of the 2nd Joint Conference on Theory and Practice of Software Development (TAPSOFT'87), Springer-V., LNCS 249, 52-68, 1987.

Bernhard Steffen, Jens Knoop, Oliver Rüthing. *The Value Flow Graph: A Program Representation for Optimal Program Transformations*. In Proceedings of the 3rd European Symposium on Programming (ESOP'90), Springer-V., LNCS 432, 389-405, 1990.

Chapter 10.3

Looking Ahead: Challenges and Pitfalls

The Impact of Setting Changes on Safety and Optimality

Safety and optimality statements are quite sensitive towards setting changes!

Three examples shall provide evidence for this:

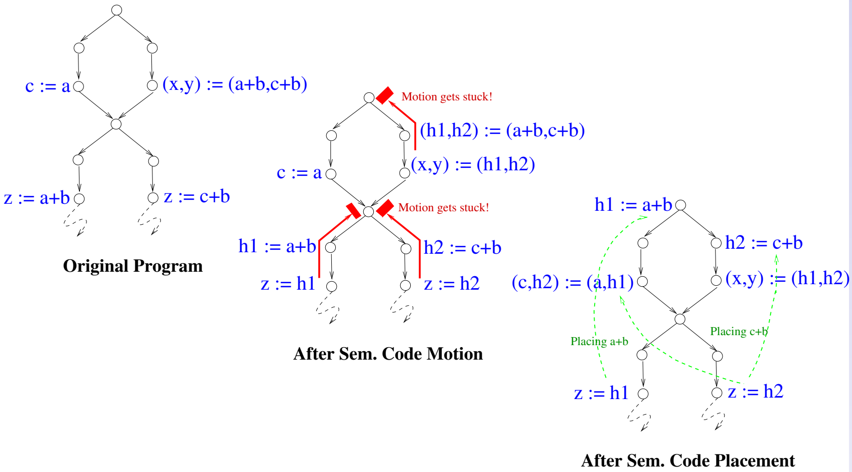
- Code motion vs. code placement
- Interdependencies of elementary transformations
- Paradigm dependencies

Chapter 10.3.1

The Impact of Moving or Placing Code

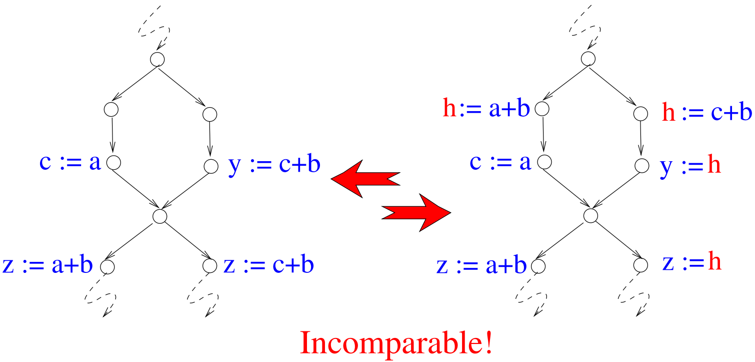
Code Motion (CM) vs. Code Placement (CP)

...CM and CP are no synonyms!



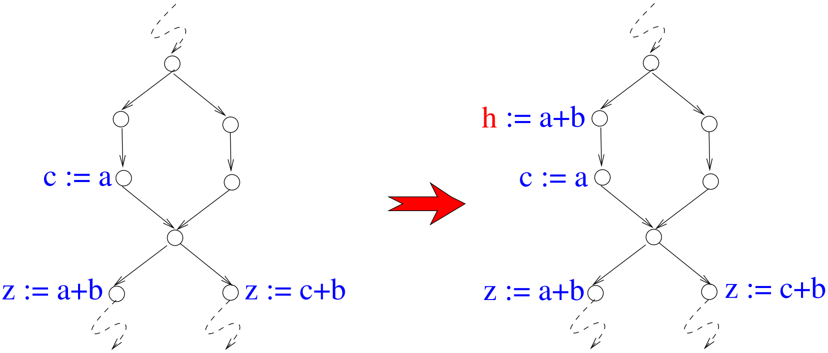
Even worse

Optimality is lost!



Even worser

The performance can be impaired, when applied naively!



References for Chapter 10.3.1

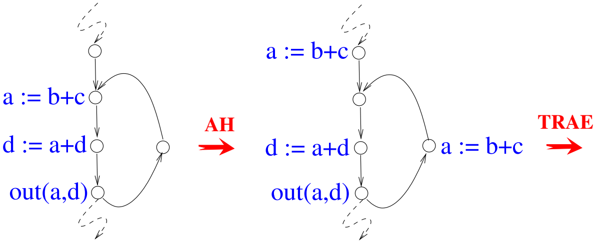
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Code Motion and Code Placement: Just Synonyms?* In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 154-169, 1998.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Expansion-based Removal of Semantic Partial Redundancies*. In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.

Chapter 10.3.2

The Impact of Interacting Transformations

Assignment Hoisting (AH) plus Totally Redundant Assignment Elimination (TRAE)

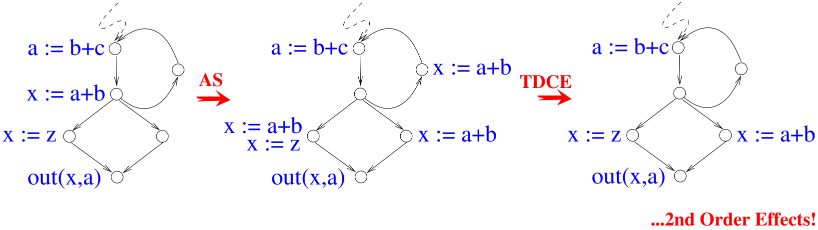
...leads to Partially Redundant Assignment Elimination (PRAE):



...2nd Order Effects!

Assignment Sinking (AS) plus Total Dead-Code Elimination (TDCE)

...leads to Partial Dead-Code Elimination (PDCE):



Conceptually

...PREE, PRAE, and PDCE can be understood as follows:

- PREE $=$ EH ; TREE
- PRAE $= (\text{AH} + \text{TRAE})^*$
- PDCE $= (\text{AS} + \text{TDCE})^*$

Optimality Results for PREE

Theorem 10.3.2.1 (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.

Optimality Results for (Pure) PRAE/PDCE

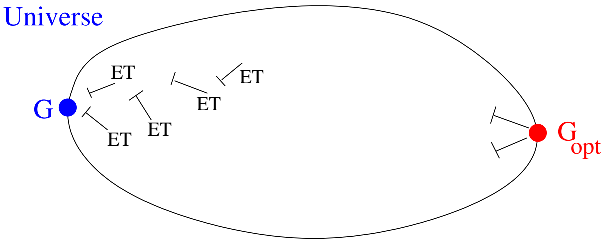
Deriving relation $\vdash$...

- PRAE... $G \vdash_{AH,TRAE} G'$ (ET={AH,TRAE})
- PDCE... $G \vdash_{AS,TDCE} G'$ (ET={AS,TDCE})

We can prove:

Theorem 10.3.2.2 (Optimality)

For PRAE and PDCE the deriving relation $\vdash_{ET}$ is confluent and terminiating.



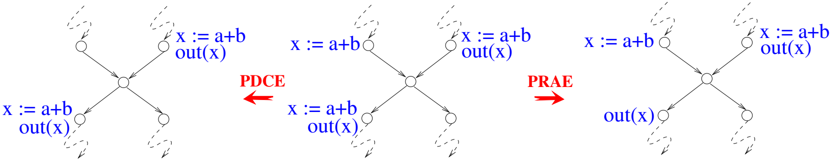
Now

...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!

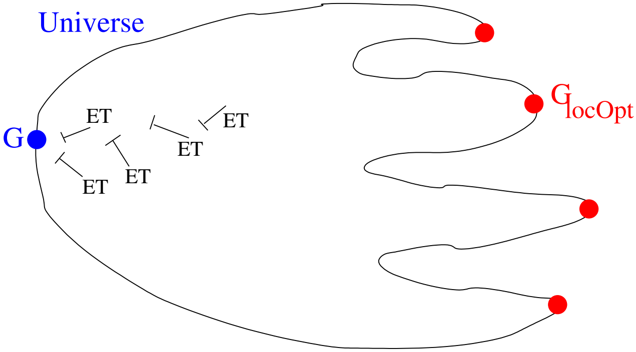
Indeed, it is but:



The resulting two programs are incomparable.

Confluence

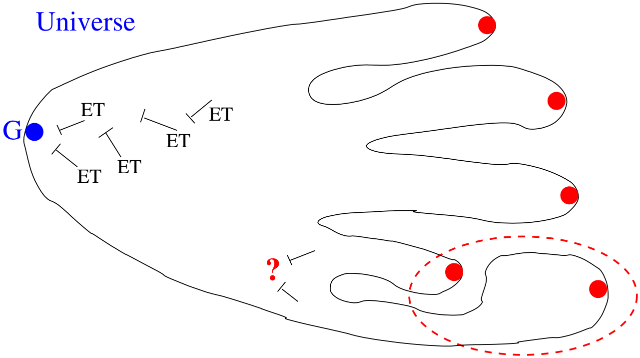
...and hence (global) optimality is lost!



Fortunately, we retain local optimality!

However

...there are settings, where we end up w/ universes like the following:



Here, even local optimality is lost!

References for Chapter 10.3.2

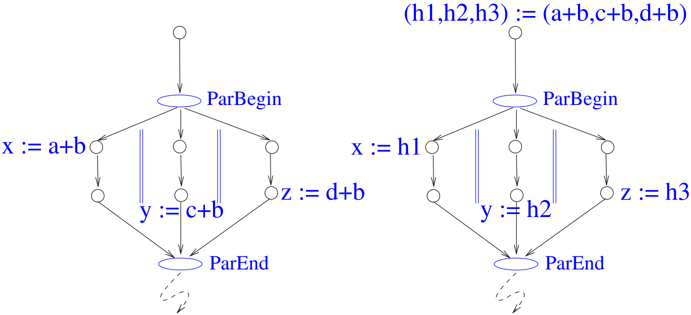
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Partial Dead Code Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):147-158, 1994.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *The Power of Assignment Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), ACM SIGPLAN Notices 30(6):233-245, 1995.
- Jens Knoop, Eduard Mehofer. *Optimal Distribution Assignment Placement*. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Springer-V., LNCS 1300, 364 - 373, 1997.

Chapter 10.3.3

The Impact of Paradigm Shifts

Adding Parallelism

...analysis and optimization of parallel programs.



Original Program **After Earliestness Transformation**

...naively transferring the strategy of "placing computations as early as possible" leads here to an essentially sequential program!

Adding Procedures

...interprocedural analysis and optimization.

Similar phenomena are encountered when naively transferring successful transformation strategies

- from the intraprocedural
- to the interprocedural

setting, e.g., the optimal PRE placement strategies of

- Busy Code Motion
- Lazy Code Motion

References for Chapter 10.3.3

- Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998. (Chapter 10, Interprocedural Code Motion: The Transformations; Chapter 10.1, Essential Differences to the Intraprocedural Setting)
- Jens Knoop, Bernhard Steffen. *Code Motion for Explicitly Parallel Programs*. In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), ACM SIGPLAN Notices 34(8):13-24, 1999.

Chapter 10.4

References, Further Reading

Further Reading for Chapter 10 (1)

Syntactic PRE

Pioneering, Groundbreaking

- Ken Kennedy. *Safety of Code Motion*. International Journal of Computer Mathematics 3(2-3):117-130, 1972.
- Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.

Lazy Code Motion and More

- Ras Bodik, Rajiv Gupta. *Register Pressure Sensitive Redundancy Elimination*. In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 107-121, 1999.

Further Reading for Chapter 10 (2)

- Preston Briggs, Keith D. Cooper. *Effective Partial Redundancy Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):159-170, 1994.
- Keith D. Cooper, Jason Eckhardt, Ken Kennedy. *Redundancy Elimination Revisited*. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008), 12-21, 2008.

Further Reading for Chapter 10 (3)

- Fred C. Chow, Sun Chan, Robert Kennedy, Shing-Ming Liu, Raymond Lo, Peng Tu. *A New Algorithm for Partial Redundancy Elimination based upon SSA Form*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM SIGPLAN Notices 32(5):273-286, 1997.
- Robert Kennedy, Sun Chan, Shing-Ming Liu, Raymond Lo, Peng Tu, Fred C. Chow. *Partial Redundancy Elimination in SSA Form*. ACM Transactions of Programming Languages and Systems 32(3):627-676, 1999.

Further Reading for Chapter 10 (4)

- Dhananjay M. Dhamdhere. *E-path_pre: Partial Redundancy Elimination Made Easy*. ACM SIGPLAN Notices 37(8):53-65, 2002.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.

Further Reading for Chapter 10 (5)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retrospective: Lazy Code Motion*. In "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection", ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.

Code-Size Sensitive Code Motion

- Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Sparse Code Motion*. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), 170-183, 2000.

Further Reading for Chapter 10 (6)

- Bernhard Scholz, R. Nigel Horspool, Jens Knoop. *Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination*. Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), ACM SIGPLAN Notices 39(7):221-230, 2004.

Complete Removal of Redundancies

- Ras Bodik, Rajiv Gupta, Mary Lou Soffa. *Complete Removal of Redundant Expressions*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), ACM SIGPLAN Notices 33(5):1-14, 1998.

Further Reading for Chapter 10 (7)

- Bernhard Steffen. *Property-Oriented Expansion*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 22-41, 1996.

Code Motion for Parallel Programs

- Jens Knoop, Bernhard Steffen. *Code Motion for Explicitly Parallel Programs*. In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), ACM SIGPLAN Notices 34(8):13-24, 1999.

Further Reading for Chapter 10 (8)

Speculative Code Motion

- Qiong Cai, Jingling Xue. *Optimal and Efficient Speculation-based Partial Redundancy Elimination*. In Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2003), 91-104, 2003.
- Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Redundancy Elimination Using Speculation*. In Proceedings of the 1998 International Conference on Computer Languages (ICCL'98), 230-239, 1998.
- R. Nigel Horspool, H. C. Ho. *Partial Redundancy Elimination Driven by a Cost-benefit Analysis*. In Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering (CSSE'97), 111-118, 1997.

Further Reading for Chapter 10 (9)

- R. Nigel Horspool, David J. Pereira, Bernhard Scholz. *Fast Profile-based Partial Redundancy Elimination*. In Proceedings of the 7th Joint Modular Languages Conference (JMLC 2006), 362-376, 2006.
- Jingling Xue, Qiong Cai. *A Lifetime Optimal Algorithm for Speculative PRE*. ACM Transactions on Architecture and Code Optimization 3(2):115-155, 2006.
- Hucheng Zhou, Wenguang Chen, Fred C. Chow. *An SSA-based Algorithm for Optimal Speculative Code Motion under an Execution Profile*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), ACM SIGPLAN Notices 46(6):98-108, 2011.

Further Reading for Chapter 10 (10)

Unifying Speculative and Non-Speculative Classical PRE

- Jingling Xue, Jens Knoop. *A Fresh Look at PRE as a Maximum Flow Problem*. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006), Springer-V., LNCS 3923, 139 - 154, 2006.

Composite Syn. Code Motion & Strength Reduction

- Dhananjay M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. International Journal of Computer Mathematics 27:1-14,31-32, 1989.
- Dhananjay M. Dhamdhere, J. R. Isaac. *A Composite Algorithm for Strength Reduction and Code Movement Optimization*. International Journal of Computer and Information Sciences 9(3):243-273, 1980.

Further Reading for Chapter 10 (11)

- Robert Kennedy, Fred C. Chow, Peter Dahl, Shing-Ming Liu, Raymond Lo, Mark Streich. *Strength Reduction via SSAPRE*. In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 144-158, 1998.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. Journal of Programming Languages 1(1):71-91, 1993.
- S. M. Joshi, Dhananjay M. Dhamdhere. *A Composite Hoisting- strength Reduction Transformation for Global Program Optimization – Part I and Part II*. International Journal of Computer Mathematics 11:21-41,111-126, 1982.

Further Reading for Chapter 10 (12)

Eliminating Partially Dead/Redundant Code

- Ras Bodik, Rajiv Gupta. *Partial Dead Code Elimination using Slicing Transformations*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM SIGPLAN Notices 32:159-170, 1997.
- Alfons Geser, Jens Knoop, Gerald Lüttgen, Oliver Rüthing, Bernhard Steffen. *Non-Monotone Fixpoint Iterations to Resolve Second Order Effects*. In Proceedings of the 6th International Conference on Compiler Construction (CC'96), Springer-V., LNCS 1060, 106-120, 1996.

Further Reading for Chapter 10 (13)

- Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Dead Code Elimination using Predication*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 102-115, 1997.
- Jens Knoop, Eduard Mehofer. *Optimal Distribution Assignment Placement*. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Springer-V., LNCS 1300, 364 - 373, 1997.
- Jens Knoop, Eduard Mehofer. *Interprocedural Distribution Assignment Placement: More than just Enhancing Intraprocedural Placing Techniques*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 26-37, 1997.

Further Reading for Chapter 10 (14)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Partial Dead Code Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):147-158, 1994.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *The Power of Assignment Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), ACM SIGPLAN Notices 30(6):233-245, 1995.
- Munehiro Takimoto, Kenichi Harada. *Partial Dead Code Elimination Using Extended Value Graph*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Springer-V., LNCS 1694, 179-193, 1999.

Further Reading for Chapter 10 (15)

Semantic PRE

Local Value Numbering ((Extended) Basic Blocks)

John Cocke, Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 2nd Revised Version, 771 pages, 1970. (Chapter 6, Optimization Methods for Algebraic Languages)

Global Value Numbering

Bowen Alpern, Mark N. Wegman, F. Kenneth Zadeck. *Detecting Equality of Variables in Programs*. In Conference Record of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 1-11, 1988.

Further Reading for Chapter 10 (16)

- Preston Briggs, Keith D. Cooper, L. Taylor Simpson. *Value Numbering*. Software: Practice and Experience 27(6):701-724, 1997.
- Cliff Click. *Global Code Motion, Global Value Numbering*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):246-257, 1995.
- Cliff Click, Keith D. Cooper. *Combing Analyses, Combining Optimizations*. ACM Transactions on Programming Languages and Systems 17(2):181-196, 1995.

Further Reading for Chapter 10 (17)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Expansion-based Removal of Semantic Partial Redundancies*. In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.
- Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. *Global Value Numbers and Redundant Computations*. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 12-27, 1988.
- Oliver Rüthing. *Interacting Code Motion Transformations: Their Impact and Their Complexity*. Springer-V., LNCS 1539, 1998.

Further Reading for Chapter 10 (18)

- Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Detecting Equalities of Variables: Combining Efficiency with Precision*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Springer-V., LNCS 1694, 232-247, 1999.
- Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Proceedings of the 2nd Joint Conference on Theory and Practice of Software Development (TAPSOFT'87), Springer-V., LNCS 249, 52-68, 1987.
- Bernhard Steffen, Jens Knoop, Oliver Rüthing. *The Value Flow Graph: A Program Representation for Optimal Program Transformations*. In Proceedings of the 3rd European Symposium on Programming (ESOP'90), Springer-V., LNCS 432, 389-405, 1990.

Further Reading for Chapter 10 (19)

Munehiro Takimoto, Kenichi Harada. *Effective Partial Redundancy Elimination based on Extended Value Graph*. Information Processing Society of Japan 38(11):2237-2250, 1990.

Composite Sem. Code Motion & Strength Reduction

Bernhard Steffen, Jens Knoop, Oliver Rüthing. *Efficient Code Motion and an Adaption to Strength Reduction*. In Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), Springer-V., LNCS 494, 394-415, 1991.

Further Reading for Chapter 10 (20)

Code Motion vs. Code Placement

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Code Motion and Code Placement: Just Synonyms?* In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 154-169, 1998.

BB Graphs vs. SI Graphs

- Jens Knoop, Dirk Koschützki, Bernhard Steffen. *Basic-block Graphs: Living Dinosaurs?* In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 65-79, 1998.

Part III

Interprocedural Data Flow Analysis

Outline

We consider:

- The Functional Approach: Basic Setting (cf. Chap. 11)
  - Mutually recursive procedures (no parameters, no local variables)
- The Functional Approach: Full Setting (cf. Chap. 12)
  - Adding value parameters, local variables: DFA stacks
  - Adding reference parameters, procedural parameters
- The Context Information Approach (cf. Chap. 13)
  - Call Strings
  - Assumption Sets
  - The Cloning-Based Approach

Chapter 11

The Functional Approach: Basic Setting

Chapter 11.1

Preliminaries, the Setting

The Basic Setting of Interprocedural DFA

Program Setting

- Programs Π with mutually recursive procedures without parameters and local variables.

Program Representations

- Flow graph systems
- Interprocedural flow graphs

...two program representations which are complimentary to each other.

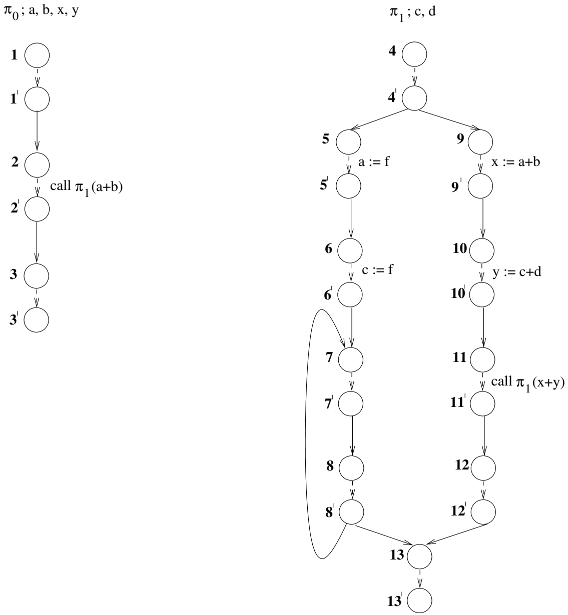
Flow Graph Systems

Intuitively, a flow graph system is a system of flow graphs, where every flow graph is a flow graph in the intraprocedural sense representing a procedure of a program Π.

Definition 11.1.1 (Flow Graph System)

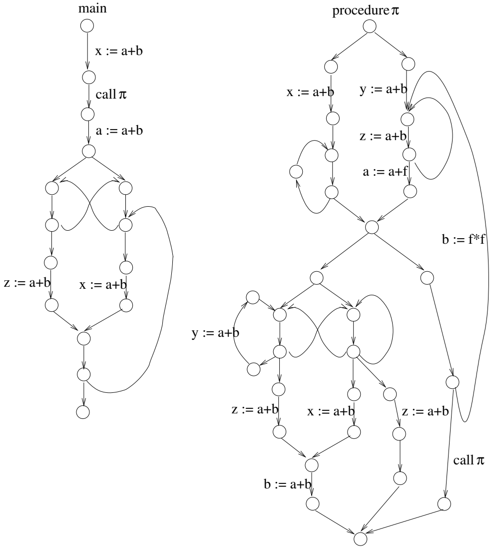
Let $\Pi = \langle \pi_0, \pi_1, \ldots, \pi_k \rangle$ be a program with main procedure (or main program) π_0 and procedures $\pi_1, \ldots, \pi_k$. A flow graph system $S_\Pi = \langle G_0, G_1, \ldots, G_k \rangle$ for Π is a system of edge-labelled or node-labelled (intraprocedural) flow graphs in the sense of Chapter 3, where flow graph G_i represents procedure π_i, $0 \leq i \leq k$.

An Edge-Labelled Flow Graph System



Flow Graph System after Cleaning Up

...unnecessary/unused nodes and edges can be removed:



Notations for Flow Graph Systems

Let $S_\Pi = \langle G_0, G_1, \ldots, G_k \rangle$ be a flow graph system.

Then:

- G_0 represents the main procedure of Π. Instead of $\mathbf{s}_0$ and $\mathbf{e}_0$, we often simply write $\mathbf{s}$ and $\mathbf{e}$.
- The sets of nodes and edges N_i and E_i, $0 \leq i \leq k$, of all flow graphs of S_Π are assumed to be pairwise disjoint.
- $N =_{df} \bigcup_{i=0}^{k} N_i$ and $E =_{df} \bigcup_{i=0}^{k} E_i$ denote the set of all nodes and edges of a flow graph system, respectively.
- $E_{call} \subseteq E$ denotes the set of edges representing a procedure call, the set of call edges.
- If Π is obvious from the context and of no further relevance, we often write S instead of S_Π.

Interprocedural Flow Graphs

Intuitively, an interprocedural flow graph melts the flow graphs of a flow graph system to a a single graph.

Definition 11.1.2 (Interprocedural Flow Graph)

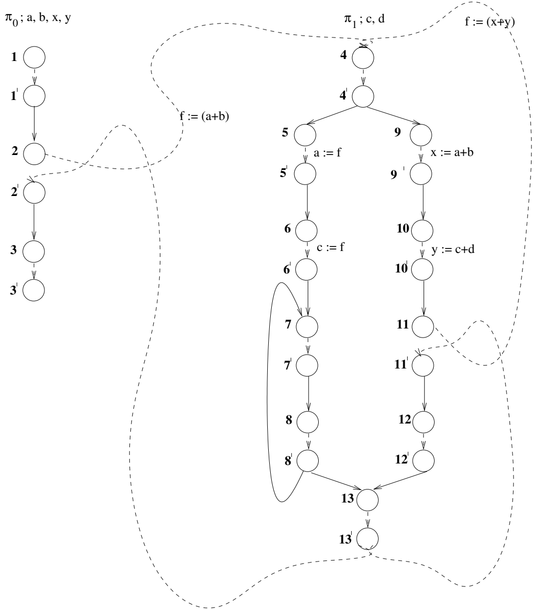
An interprocedural flow graph $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ is induced by a flow graph system S, where G^* evolves from S by replacing every call edge e of a flow graph G_i of S by two new edges, the call edge e_c and the return edge e_r.

The call edge e_c connects the source node of e with the start node of the flow graph representing the called procedure.

The return edge e_r connects the end node of the flow graph representing the called procedure with the final node of e.

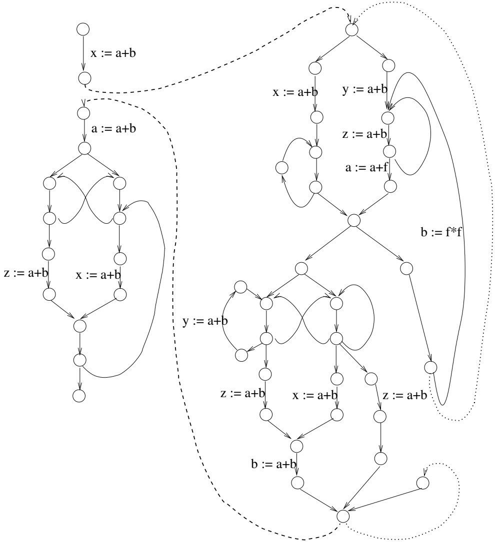
In particular, $\mathbf{s}^*$ and $\mathbf{e}^*$ are given by $\mathbf{s}_0$ and $\mathbf{e}_0$, respectively.

An Edge-Labelled Interprocedural Flow Graph



Interprocedural Flow Graph after Cleaning Up

...unnecessary/unused nodes and edges can be removed:



Notations for Interprocedural Flow Graphs

Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be an interprocedural flow graph.

Then:

- E_c^* and E_r^* denote the set of all call edges and return edges of G^*, respectively.
- $E_{call}^* =_{df} E_c^* \cup E_r^*$ denotes the union of the sets of call and return edges of G^*.
- Instead of $\mathbf{s}^*$ and $\mathbf{e}^*$, we often simply write $\mathbf{s}$ and $\mathbf{e}$.

Chapter 11.2

IDFA Specifications, IDFA Problems

Interprocedural DFA Specification

Let S be an edge-labelled flow graph system, and let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be the interprocedural flow graph induced by S.

Definition 11.2.1 (IDFA Specification)

An (interprocedural) DFA specification for G^* is a quadruple $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_\mathbf{s}, d)$ with

- $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ a DFA lattice
- $[\![\ ]\!]^* : E^* \to (\mathcal{C} \to \mathcal{C})$ a DFA functional
- $c_\mathbf{s} \in \mathcal{C}$ an initial information/assertion
- $d \in \{fw, bw\}$ a direction of information flow

Note: As intraprocedurally, the validity of $c_\mathbf{s} \in \mathcal{C}$ at $\mathbf{s} \equiv \mathbf{s}^*$ needs to be ensured by the calling context of G^*.

Notations for IDFA Specifications

Let $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, d)$ be a DFA specification for G^*.

Then

- The elements of $\mathcal{C}$ represent the data flow information of interest.
- The functions $[\![\ e\ ]\!]^*$, $e \in E^*$, abstract the concrete semantics of instructions to the level of the analysis.
- In the parameterless setting considered in this chapter, the local abstract semantics of call edges and return edges of E^* are given the identity function on $\mathcal{C}$.

As intraprocedurally

- $\widehat{C}$ is called a DFA lattice.
- $[\![\ ]\!]^*$ is called an (interprocedural) DFA functional.
- $[\![\ e\ ]\!]^*$, $e \in E^*$, is called a (local) DFA function.

Interprocedural DFA Problem

Definition 11.2.2 (IDFA Problem)

An interprocedural DFA specification $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, d)$ defines an interprocedural DFA problem for G^*.

Practically Relevant IDFA Problems

...similarly to the intraprocedural case, interprocedural DFA problems are practically relevant, if they are

- monotonic
- distributive (additive)

and satisfy the

- descending (ascending) chain condition (for the function lattice of the DFA lattice!).

Properties of IDFA Problems

Definition 11.2.3 (Properties of IDFA Problems)

Let $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, d)$ be an IDFA specification for G^*.

The IDFA problem induced by $\mathcal{S}_{G^*}$

- is monotonic/distributive/additive iff the DFA functional $[\![\ ]\!]^*$ of $\mathcal{S}_{G^*}$ is monotonic/distributive/additive.
- satisfies the descending (ascending) chain condition iff the function lattice $[\mathcal{C} \to \mathcal{C}]$ of the DFA lattice $\widehat{C}$ of $\mathcal{S}_{G^*}$ satisfies the descending (ascending) chain condition.

Note: If $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending (ascending) chain condition, then also $\widehat{C}$ satisfies the descending (ascending) chain condition.

Chapter 11.3

Naive Interprocedural DFA

Naive Interprocedural DFA

Note:

Considering an interprocedural flow graph G^* an intraprocedural flow graph, the (intraprocedural) notions of

- a path
- the *MOP* approach
- the *MaxFP* approach
- the Theorems for Safety, Coincidence, and Termination

carry over from the intraprocedural setting and an intraprocedural flow graph G to the interprocedural setting and an interprocedural flow graph G^*.

The *MOP* Approach and *MOP* Solution for G^*

Let $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, fw)$ be a DFA specification for G^*.

Definition 11.3.1 (The *MOP* Solution for G^*)

The *MOP* solution of $\mathcal{S}_{G^*}$ for G^* is defined by:

$$MOP_{\mathcal{S}_{G^*}} : N^* \to \mathcal{C}$$

$$\forall\, n \in N^*.\ MOP_{\mathcal{S}_{G^*}}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!]^*(c_{\mathbf{s}}) \mid p \in \mathbf{P}_{G^*}[\mathbf{s}, n] \,\}$$

The *MaxFP* Approach for G^*

Let $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, fw)$ be a DFA specification for G^*.

Equation System 11.3.2 (The *MaxFP* EQS for G^*)

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

Let

$$inf^{\star}_{c_{\mathbf{s}}}(n),\;\; n \in N^*$$

denote the greatest solution of Equation System 11.3.2.

The *MaxFP* Solution for G^*

Definition 11.3.3 (The *MaxFP* Solution for G^*)

The *MaxFP* solution of $\mathcal{S}_{G^*}$ for G^* is defined by:

$$MaxFP_{\mathcal{S}_{G^*}} : N^* \rightarrow \mathcal{C}$$

$$\forall\, n \in N^*.\ MaxFP_{\mathcal{S}_{G^*}}(n) =_{df} inf^{\star}_{\mathbf{c_s}}(n)$$

Safety and Coincidence

Corollary 11.3.4 (Safety)

The *MaxFP* solution of $\mathcal{S}_{G^*}$ for G^* is a safe (i.e., lower) approximation of the *MOP* solution of $\mathcal{S}_{G^*}$ for G^*, i.e.,

$$\forall\, n \in N^*.\ \mathit{MaxFP}_{\mathcal{S}_{G^*}}(n) \sqsubseteq \mathit{MOP}_{\mathcal{S}_{G^*}}(n)$$

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

Corollary 11.3.5 (Coincidence)

The *MaxFP* solution of $\mathcal{S}_{G^*}$ for G^* and the *MOP* solution of $\mathcal{S}_{G^*}$ for G^* coincide, i.e.,

$$\forall\, n \in N^*.\ \mathit{MaxFP}_{\mathcal{S}_{G^*}}(n) = \mathit{MOP}_{\mathcal{S}_{G^*}}(n)$$

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

Termination

Corollary 11.3.6 (Termination)

Applied to G^* and $\mathcal{S}_{G^*}$, the Generic Fixed Point Algorithm 3.4.3 terminates with the *MaxFP* solution of $\mathcal{S}_{G^*}$ for G^*, if

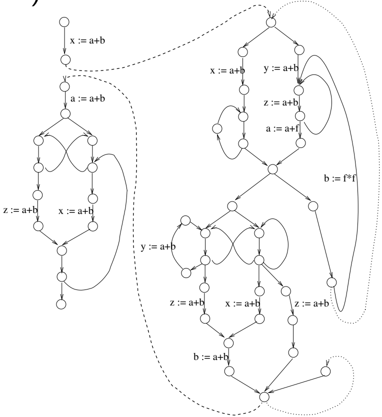
1. $[\![\ ]\!]^*$ is monotonic
2. $\widehat{\mathcal{C}}$ satisfies the descending chain condition.

...all three corollaries follow immediately from their intra-procedural counterparts of Chapter 3.

Everything done? Unfortunately, not!

...the *MOP* approach for G^* considers much too many paths as it does not respect the call/return behaviour of interprocedural program paths.

For illustration, consider the interprocedurally infeasible path (highlighted in red):



Observations on Naive Interprocedural DFA

- The notion of a (finite) path of intraprocedural flow graphs extends naturally to interprocedural flow graphs.
- In contrast to intraprocedural flow graphs, however, where every path connecting two nodes represents (up to non-determinism) a feasible execution of the program, this does not hold for interprocedural flow graphs.
- This causes the solutions of the naive extensions of the intraprocedural *MOP* approach and *MaxFP* approach to an interprocedural flow graph to be overly conservative.

In truely interprocedural DFA considered next this is taken care of and avoided by focusing on interprocedurally valid paths.

Chapter 11.4

The Interprocedural Meet over All Paths Approach

Interprocedurally Valid Paths

Intuitively, interprocedurally valid paths respect the call/return behaviour of procedure calls

Definition 11.4.1 (Interprocedurally Valid Path)

Identifying call and return edges of G^* with opening and closing brackets "(" and ")", respectively, the set of interprocedurally valid paths is given by the set of prefix-closed expressions of the language of balanced bracket expressions.

Notation: In the following we denote the set of interprocedurally valid paths (for short: interprocedural paths) from a node m to a node n by $\mathbf{IP}[m, n]$.

Remarks on Interprocedurally Valid Paths

- Considering the sequences of edge labelings (we suppose that each edge is uniquely marked by some label) of a path a word of a formal language, the set of intraprocedurally valid paths is given by a regular language, the one of interprocedurally valid paths by a context-free language.
- The notion of interprocedurally valid paths can and has been defined in various ways:
  - The definition of interprocedurally valid paths as in Definition 11.4.1 has been proposed by Reps, Horwitz, and Sagiv (POPL'95).
  - Sharir and Pnueli gave an algorithmic definition of interprocedurally valid paths in 1981.
  - Based on the preceding remark, interprocedurally valid paths can also be defined in terms of a context-free language/grammar.

The *IMOP* Approach and the *IMOP* Solution

Let $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, fw)$ be a DFA specification for G^*.

Definition 11.4.2 (The *IMOP* Solution)

The *IMOP* solution of $\mathcal{S}_{G^*}$ is defined by:

$$IMOP_{\mathcal{S}_{G^*}} : N^* \rightarrow \mathcal{C}$$

$$\forall\, n \in N^*.\ IMOP_{\mathcal{S}_{G^*}}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!]^*(c_{\mathbf{s}}) \mid p \in \mathbf{IP}[\mathbf{s}, n] \,\}$$

where $\mathbf{IP}[\mathbf{s}, n]$ denotes the set of interprocedurally valid paths from $\mathbf{s}$ to n.

Conservative and Optimal IDFA Algorithms

Definition 11.4.3 (Conservative IDFA Algorithm)

An IDFA algorithm A is *IMOP* conservative for $\mathcal{S}_{G^*}$, if A terminates with a lower approximation of the *IMOP* solution of $\mathcal{S}_{G^*}$.

Definition 11.4.5 (Optimal IDFA Algorithm)

An IDFA algorithm A is *IMOP* optimal for $\mathcal{S}_{G^*}$, if A terminates with the *IMOP* solution of $\mathcal{S}_{G^*}$.

Chapter 11.5

The Interprocedural Maximal Fixed Point Approach

The Key to Interprocedural DFA: Intuitively

Let $S = (G_0, G_1, \ldots, G_k)$ be an intraprocedural flow graph.

The function

$$[\![\ ]\!] : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

$$\forall\, n \in N.\ \forall\, c_{\mathsf{s}} \in \mathcal{C}.\ [\![\, n\,]\!](c_{\mathsf{s}}) =_{df} MaxFP_{\mathcal{S}_G^{c_{\mathsf{s}}}}(n)$$

with $\mathcal{S}_G^{c_{\mathsf{s}}} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!], c_{\mathsf{s}}, fw)$ is the key to computable interprocedural DFA.

We have:

Lemma 11.5.1

- $\forall\, n \in N. \forall\, c_{\mathsf{s}} \in \mathcal{C}.\ [\![\, n\,]\!](c_{\mathsf{s}}) \sqsubseteq MOP_{\mathcal{S}_G^{c_{\mathsf{s}}}}(n)$, if $[\![\ ]\!]$ is monotonic.
- $\forall\, n \in N. \forall\, c_{\mathsf{s}} \in \mathcal{C}.\ [\![\, n\,]\!](c_{\mathsf{s}}) = MOP_{\mathcal{S}_G^{c_{\mathsf{s}}}}(n)$, if $[\![\ ]\!]$ is distributive.

The Key to Interprocedural DFA: Intuitively

Obviously

The function $[\![\ ]\!]$ can stepwise be computed by iteratively applying the Generic Fixed Point Algorithm 3.4.3 to the elements $c_s \in \mathcal{C}$.

Next, we will present a less naive, systematic approach for computing $[\![\ ]\!]$.

The Key to Interprocedural DFA: Formally

The Functional *MaxFP* Approach

- lifts the *MaxFP* approach from elements of $\mathcal{C}$ to functions on $\mathcal{C}$. Intuitively, it is the pointwise extension of the *MaxFP* approach to all DFA lattice elements computing the *MaxFP* solution for all of them simultaneously.

Equation System 11.5.2 (Functional *MaxFP* EQS)

$$[\![\, n \,]\!] = \begin{cases} Id_{\mathcal{C}} & \text{if } \; n = \mathbf{s} \\ \bigsqcap \{ [\![\, (n,m) \,]\!] \circ [\![\, m \,]\!] \mid m \in pred(n) \} & \text{otherwise} \end{cases}$$

Let

$$[\![\;\;]\!]^{\star} : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

denote the greatest solution of Equation System 11.5.2.

Main Result: Equivalence

The *MaxFP* and the functional *MaxFP* approach are

- equivalent.

Theorem 11.5.3 (Equivalence)

$$\forall\, n \in N.\ \forall\, c_{\mathsf{s}} \in \mathcal{C}.\ [\![\, n \,]\!]^{\star}(c_{\mathsf{s}}) = MaxFP_{\mathcal{S}_G^{c_{\mathsf{s}}}}(n)$$

This means: The function $[\![\ ]\!]^{\star}$ is the function $[\![\ ]\!]$ we identified as the key to interprocedural DFA.

Theorem 11.5.4 (*MOP* Equivalence)

$$\forall\, n \in N.\ \forall\, c_{\mathsf{s}} \in \mathcal{C}.\ [\![\, n \,]\!]^{\star}(c_{\mathsf{s}}) = MOP_{\mathcal{S}_G^{c_{\mathsf{s}}}}(n)$$

if $[\![\ ]\!]$ is distributive.

Note

The functional variant of the *MaxFP* approach is the key not only to computable

- interprocedural DFA (i.e., of programs w/ procedures)

but also to e.g., computable

- object-oriented (i.e., of programs w/ classes, objects, and methods)
- parallel (i.e., of programs w/ parallelism)

data flow analysis.

The *IMaxFP* Approach

...is a two-stage approach:

- Stage 1: Preprocess – Computing the Semantics of Procedures
- Stage 2: Main Process – Computing the *IMaxFP* solution

Notations

The definition of the two stage *IMaxFP* approach requires the following mappings on a flow graph system S:

- $\mathit{flowGraph} : N \cup E \to S$ 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, respectively.

The *IMaxFP* Approach (1)

Stage 1: Preprocess – Computing the Semantics of Procedures

Equation System 11.5.5 (2nd Order *IMaxFP* EQS)

$$[\![n]\!] = \begin{cases} Id_{\mathcal{C}} & \text{if } n \in \{ \mathbf{s}_0, \ldots, \mathbf{s}_k \} \\ \bigsqcap \{ [\![(m,n)]\!] \circ [\![m]\!] \mid m \in pred_{flowGraph(n)}(n) \} & \text{otherwise} \end{cases}$$

and

$$[\![e]\!] = \begin{cases} [\![e]\!]^* & \text{if } e \in E \backslash E_{call} \\ [\![end(caller(e))]\!] & \text{otherwise} \end{cases}$$

Let

$$[\![n]\!]^\star, n \in N, \quad [\![e]\!]^\star, e \in E$$

denote the greatest solutions of Equation System 11.5.5.

Stage 2: Main Process – The "Actual" Interprocedural DFA

Equation System 11.5.6 (1st Order *IMaxFP* EQS)

$$inf(n) =$$

$$\begin{cases} c_{\mathbf{s}} \quad \text{if } n = \mathbf{s} \; (\equiv \mathbf{s}_0) \\ \bigsqcap \{ \, inf(src(e)) \mid e \in caller(flowGraph(n)) \, \} \; \text{ if } n \in \{\mathbf{s}_1, \ldots, \mathbf{s}_k\} \\ \bigsqcap \{ \, [\![\, (m,n) \,]\!]^{\star}(inf(m)) \mid m \in pred_{flowGraph(n)}(n) \, \} \; \text{ otherwise} \end{cases}$$

Let

$$inf^{\star}_{c_{\mathbf{s}}}(n), n \in N$$

denote the greatest solution of Equation System 11.5.6.

The *IMaxFP* Solution

Let $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, fw)$ be a DFA specification for G^*.

Definition 11.5.7 (The *IMaxFP* Solution)

The *IMaxFP* solution of $\mathcal{S}_{G^*}$ is defined by:

$$IMaxFP_{\mathcal{S}_{G^*}} : N^* \to \mathcal{C}$$

$$\forall\, n \in N^*.\ IMaxFP_{\mathcal{S}_{G^*}}(n) =_{df} inf^{\star}_{c_{\mathbf{s}}}(n)$$

Note that $N = N^*$ allowing us to identify corresponding nodes of S and G^*.

Chapter 11.6

The Generic Fixed Point Algorithms

Chapter 11.6.1

Basic Algorithms: Plain Vanilla

Algorithm 11.6.1.1 – 2nd Order Preprocess

Input: A DFA specification $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, d)$ for G^* resp. S. If $d = bw$, the reversed versions of all graphs are used.

Output: On termination (cf. Theorem 11.6.3.1), the variables *gtr* (*global transformation*) and *ltr* (*local transformation*) store the values of the functions $[\![\![n]\!]\!]^\star : \mathcal{C} \to \mathcal{C}$, $n \in N$, and $[\![e]\!]^\star : \mathcal{C} \to \mathcal{C}$, $e \in E$, being the greatest solutions of the 2nd order *IMaxFP* Equation System 11.5.5.

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 function $[\![\ ]\!]$ of call edges. The temporary *meet* stores the result of the most recent meet operation.

Algorithm 11.6.1.1 – 2nd Order Preprocess

(Prologue: Initializing the annotation arrays *gtr* and *ltr* and the variable *workset*)

FORALL $n \in N$ DO
    IF $n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ 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;
$workset := \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$;

Algorithm 11.6.1.1 – 2nd Order Preprocess

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO
    CHOOSE $m \in workset$;
      $workset := workset \backslash \{m\}$;
      (Update the successor-environment of node m)
      IF $m \in \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$
        THEN
          FORALL $e \in caller(flowGraph(m))$ DO
            $ltr[e] := gtr[m]$;
            $meet := ltr[e] \circ gtr[src(e)] \sqcap gtr[dst(e)]$;
            IF $gtr[dst(e)] \sqsupset meet$
              THEN
                $gtr[dst(e)] := meet$;
                $workset := workset \cup \{dst(e)\}$
            FI
          OD

Algorithm 11.6.1.1 – 2nd Order Preprocess

```
        ELSE ( i.e., m ∉ {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.
```

Algorithm 11.6.1.2 – 1st Order Main Process

Input: A DFA specification $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_{\mathbf{s}}, d)$ for G^* resp. S. If $d = bw$, the reversed versions of all graphs are used, and the data flow functional $[\![\ ]\!]^\star : E \to (\mathcal{C} \to \mathcal{C})$ computed by Algorithm 11.6.1.1 for $\mathcal{S}_{G^*}$.

Output: On termination (cf. Theorem 11.6.3.1), variable $inf[n]$, $n \in N$, stores the *IMaxFP*-solution of $\mathcal{S}_{G^*}$ at node n.

Additionally, we have (cf. Interprocedural Safety Theorem 11.7.3 and Interprocedural Coincidence Theorem 11.7.4): If

- $[\![\ ]\!]^*$ is distributive: *inf* stores
- $[\![\ ]\!]^*$ is monotonic: *inf* stores a lower approximation of

the *IMOP* solution of $\mathcal{S}_{G^*}$ at node n.

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S. The temporary *meet* stores the result of the most recent meet operation.

Algorithm 11.6.1.2 – 1st Order Main Process

(Prologue: Initialization of the annotation array *inf* and the variable *workset*)
FORALL $n \in N \backslash \{\mathbf{s}_0\}$ DO $inf[n] := \top$ OD;
$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;
$workset := \{ \mathbf{s}_0 \}$;

Algorithm 11.6.1.2 – 1st Order Main Process

(Main process: Iterative fixed point computation)
WHILE $workset \neq \emptyset$ DO
    CHOOSE $m \in workset$;
        $workset := workset \backslash \{ m \}$;
        (Update the successor-environment of node m)
        FORALL $n \in succ_{flowGraph(m)}(m)$ DO
            $meet := [\![(m, n)]\!]^{\star}(inf[m]) \sqcap inf[n]$;
            IF $inf[n] \sqsupset meet$
                THEN
                    $inf[n] := meet$;
                    $workset := workset \cup \{n\}$
            FI;

Algorithm 11.6.1.2 – 1st Order 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.
```

Chapter 11.6.2

Enhanced Algorithms: Improving Performance

Variant 1: Exploiting $[\![\ ]\!]^\star$ More Effectively

...improving performance of the 1st order *IMaxFP*-Algorithm:

- Algorithm 11.6.1.1 and Algorithm 11.6.2.1 constitute a second pair of algorithms computing the *IMaxFP* solution.
- Algorithm 11.6.2.1 uses the semantics functions computed by Algorithm 11.6.1.1 more effectively than Algorithm 11.6.1.2.
- Unlike Algorithm 11.6.1.2, Algorithm 11.6.2.1 does not iterate over all nodes of S but only over procedure start nodes. After stabilization of the solution for the start nodes, a single run over all other nodes in the epilogue suffices to compute the *IMaxFP* solution at every node.

Algorithm 11.6.2.1 – 1st Order Main Process

Input: A DFA specification $\mathcal{S}_{G^*} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]^*, c_s, d)$ for G^* resp. S. If $d = bw$, the reversed versions of all graphs are used, and the data flow functional $[\![\ ]\!]^\star : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ computed by Algorithm 11.6.1.1 for $\mathcal{S}_{G^*}$.

Output: On termination (cf. Theorem 13.6.3.1), variable $inf[n]$, $n \in N$, stores the *IMaxFP*-solution of $\mathcal{S}_{G^*}$ at node n.

Additionally, we have (cf. Interprocedural Safety Theorem 11.7.3 and Interprocedural Coincidence Theorem 11.7.4): If

- $[\![\ ]\!]^*$ is distributive: *inf* stores
- $[\![\ ]\!]^*$ is monotonic: *inf* stores a lower approximation of

the *IMOP* solution of $\mathcal{S}_{G^*}$ at node n.

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S. The temporary *meet* stores the result of the most recent meet operation.

Algorithm 11.6.2.1 – 1st Order Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)
FORALL $\mathbf{s} \in \{\mathbf{s}_i \mid i \in \{1, \ldots, k\}\}$ DO $inf[\mathbf{s}] := \top$ OD;
$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;
$workset := \{\mathbf{s}_i \mid i \in \{1, 2, \ldots, k\}\}$;

Algorithm 11.6.2.1 – 1st Order Main Process

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO
    CHOOSE $\mathbf{s} \in workset$;
        $workset := workset \backslash \{\mathbf{s}\}$;
        $meet := inf[\mathbf{s}] \sqcap$
            $\bigsqcap\{[\![\, src(e)\,]\!]^{\star}(inf[start(flowGraph(e))]) \mid e \in caller(flowGraph(\mathbf{s}))\}$;
        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;

Algorithm 11.6.2.1 – 1st Order Main Process

(Epilogue)

FORALL $n \in N \backslash \{\mathbf{s}_i \mid i \in \{0, \ldots, k\}\}$ DO

$inf[n] := [\![\, n \,]\!]^{\star}(inf[start(flowGraph(n))])$ OD.

Variant 2: Interleaving 2nd & 1st Order Alg.

...improving performance of the algorithm composition by applying the 2nd order algorithm demand-drivenly controlled by the 1st order algorithm:

- Unlike the two algorithm pairs introduced so far, this algorithm variant interleaves the 1st order main process and the 2nd order preprocess.
- In effect, the semantics of procedures $[\![\ ]\!]^{\star} : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ is computed demand-drivenly partially only instead of exhaustively totally, i.e., it is computed only for arguments encountered in the course of the 1st order main process instead of unguidedly for all arguments.

Variant 2: Interleaving 2nd & 1st Order Alg.

Algorithm 11.6.2.2 – Sketch of the Interleaved Algorithms

- The computation starts with the 1st order main process algorithm.
- If a procedure call is encountered during the iterative process, the 2nd order preprocess algorithm is started for this procedure and the current data flow fact.
- After completion of the computation of the effect of the procedure for this data flow fact, the 1st order main process algorithm is continued with the computed result.

Note:

- The semantics of procedures is computed demand-drivenly exclusively for required arguments.
- Overall, this leads to some performance gain in practice, which, however, is difficult to quantify.

Chapter 11.6.3

Termination

Termination

Theorem 11.6.3.1 (Termination)

- The sequential compositions of Algorithm 11.6.1.1 (2nd order) and Algorithm 11.6.1.2 (1st order) resp. Algorithm 11.6.2.1 (1st order)
- Algorithm 11.6.2.2 interleaving Algorithm 11.6.1.2 resp. Algorithm 11.6.2.1 and Algorithm 11.6.1.1

terminate with the *IMaxFP* solution, if the data flow analysis functional $[\![\ ]\!]^*$ is monotonic and the function lattice $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition.

Note: Validity of the descending chain condition on the function lattice $[\mathcal{C} \to \mathcal{C}]$ implies validity of the descending chain condition on the underlying lattice $\widehat{\mathcal{C}}$.

Chapter 11.7

Safety and Coincidence

Complete Interprocedural Paths

Definition 11.7.1 (Complete Interprocedural 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, completeness of a path p, i.e., $p \in \mathbf{CIP}[\mathbf{s}_i, n]$, ensures that the occurrences of $\mathbf{s}_i$ and n belong to the same incarnation of the procedure.
- The subpaths of a complete interprocedural path that belong to a procedure call, are either disjoint or properly nested.

Main Results: 2nd Order Analysis

Safety and coincidence results of the 2nd order analysis:

Theorem 11.7.2 (2nd Order Analysis)

For all $e \in E_{call}$ we have:

1. Safety:
   $[\![\, e \,]\!]^\star \sqsubseteq \bigsqcap\{[\![\, p \,]\!]^* \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, if the data flow analysis functional $[\![\ ]\!]^*$ is monotonic.
2. Coincidence:
   $[\![\, e \,]\!]^\star = \bigsqcap\{[\![\, p \,]\!]^* \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, if the data flow analysis functional $[\![\ ]\!]^*$ is distributive.

where the mappings *src* and *dst* yield the start node and the final node of an edge, respectively.

Main Results: 1st Order Analysis

Safety and coincidence results of the 1st order analysis:

Theorem 11.7.3 (Interprocedural Safety)

The *IMaxFP* solution of $\mathcal{S}_{G^*}$ is a safe (i.e., lower) approximation of the *IMOP* solution of $\mathcal{S}_{G^*}$, i.e.,

$$\forall\, n \in N.\ \mathit{IMaxFP}_{\mathcal{S}_{G^*}}(n) \sqsubseteq \mathit{IMOP}_{\mathcal{S}_{G^*}}(n)$$

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

Theorem 11.7.4 (Interprocedural Coincidence)

The *IMaxFP* solution of $\mathcal{S}_{G^*}$ coincides with the *IMOP* solution of $\mathcal{S}_{G^*}$, i.e.,

$$\forall\, n \in N.\ \mathit{IMaxFP}_{\mathcal{S}_{G^*}}(n) = \mathit{IMOP}_{\mathcal{S}_{G^*}}(n)$$

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

Conservativity, Optimality of IDFA Algorithms

Corollary 11.7.5 (*IMOP* Conservativity)

The IDFA algorithms of Chapter 11.6 are *IMOP* conservative for $\mathcal{S}_{G^*}$ (i.e., terminate with a lower approximation of the *IMOP* solution of $\mathcal{S}_{G^*}$), if $[\![\ ]\!]^*$ is monotonic and $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition.

Corollary 11.7.6 (*IMOP* Optimality)

The IDFA algorithms of Chapter 11.6 are *IMOP* optimal for $\mathcal{S}_{G^*}$ (i.e., terminate with the *IMOP* solution of $\mathcal{S}_{G^*}$), if $[\![\ ]\!]^*$ is distributive and $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition.

Chapter 11.8

Soundness and Completeness

Soundness and Completeness (1)

Analysis Scenario:

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

Definition 11.8.1 (Soundness)

$\mathcal{S}^{\phi}_{G*}$ is sound for ϕ, if, whenever the *IMOP* solution of $\mathcal{S}^{\phi}_{G*}$ indicates that ϕ is valid, then ϕ is valid.

Definition 11.8.2 (Completeness)

$\mathcal{S}^{\phi}_{G*}$ is complete for ϕ, if, whenever ϕ is valid, then the *IMOP* solution of $\mathcal{S}^{\phi}_{G*}$ indicates that ϕ is valid.

Soundness and Completeness (2)

Intuitively

- Soundness means: $IMOP_{\mathcal{S}^{\phi}_{G*}}$ implies ϕ.
- Completeness means: ϕ implies $IMOP_{\mathcal{S}^{\phi}_{G*}}$.

Soundness and Completeness (3)

If $\mathcal{S}_{G*}^{\phi}$ is sound and complete for ϕ, this intuitively means:

We compute

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

In other words

- We compute the program property of interest accurately!

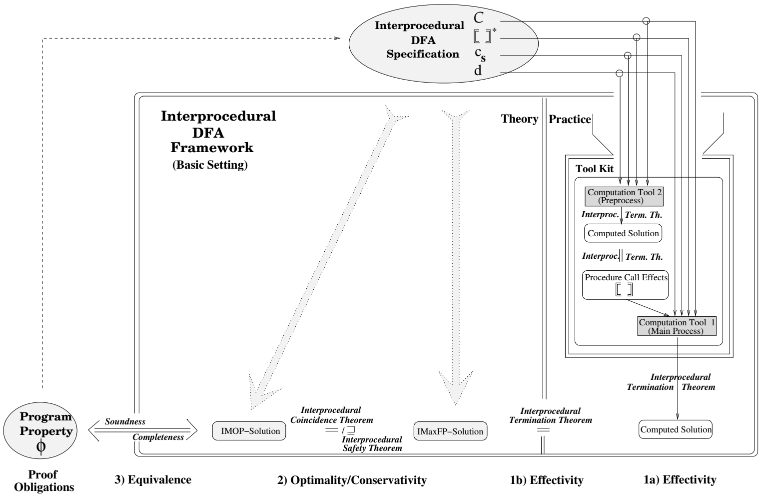
Chapter 11.9

A Uniform Framework and Toolkit View

Interprocedural DFA: A Holistic Uniform View

...considering interprocedural DFA from a holistic angle:

- A Uniform Framework and Toolkit View

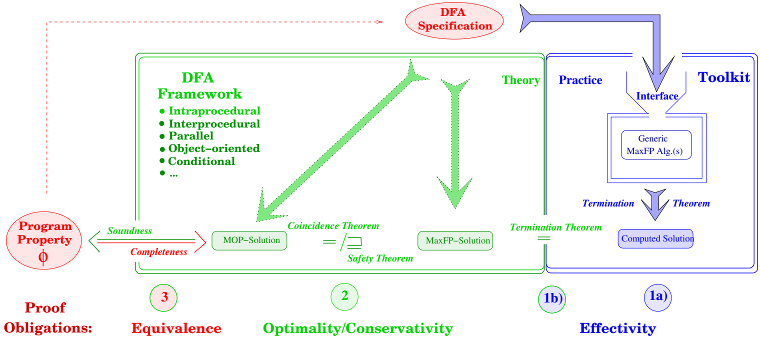


Note: The Preceding Schematic View of IDFA

...provides evidence for the claim of Chapter 3.8 that

- The Uniform Framework and Toolkit View of DFA

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



Chapter 11.10

Applications

Applications

For the parameterless base setting of interprocedural DFA

- the specifications of intraprocedural DFA problems can be reused unmodified.

In order to be effective

- the descending chain condition must hold both for the DFA lattice and its corresponding function lattice.

This requirement is satisfied in particular for all

- bitvector problems (availability of expressions, liveness of variables, reaching definitions, etc.) but not for simple constants. Therefore, weaker and simpler classes of constants are considered interprocedurally, e.g., linear constants.

Chapter 11.11

References, Further Reading

Further Reading for Chapter 11 (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 12, Interprocedural Analysis)

- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 11, Interprocedural Analysis and Optimization)

Further Reading for Chapter 11 (2)

- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 19, Interprocedural Analysis and Optimization)
- Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.3, The Functional Approach to Interprocedural Analysis, 196-209.

Chapter 12

The Functional Approach: Full Setting

Outline

In this chapter, we extend the parameterless basic setting of interprocedural DFA considered in Chapter 11 by successively adding

- value parameters and local variables (cf. Chapter 12.1)
- procedural parameters (cf. Chapter 12.2)
- reference parameters (cf. Chapter 12.3)
- static procedure nesting (cf. Chapter 12.4)

Subsequently, we sketch

- applications (cf. Chapter 12.5)
  - bitvector analyses: interprocedural availability
  - constant propagation: interprocedural copy constants

Chapter 12.1

Adding Value Parameters and Local Variables

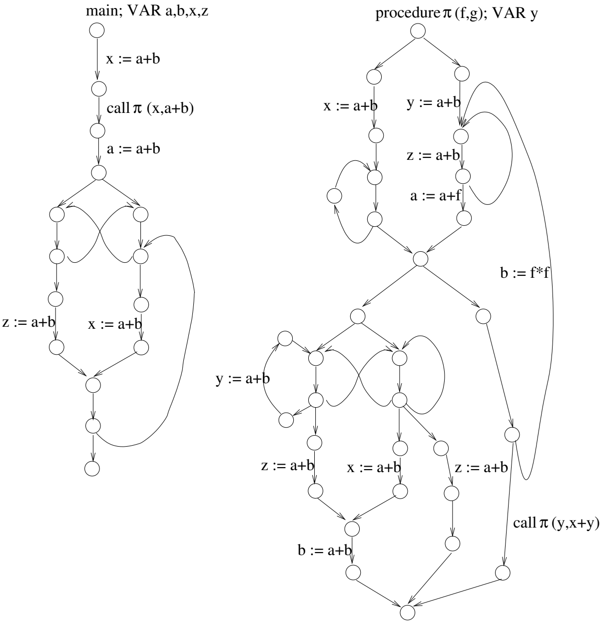
Flow Graph Systems, Interprocedural Flow Graphs

Introducing value parameters and local variables requires to extend the notions of flow graph systems (FGS) and interprocedural flow graphs (IFG) to

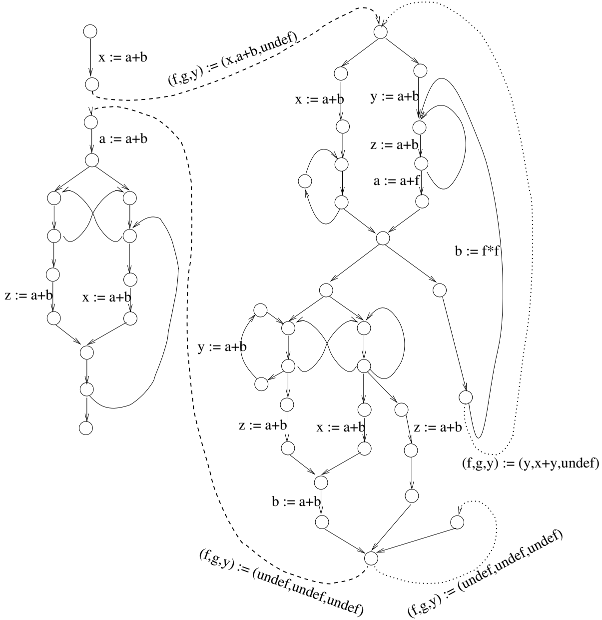
- flow graph systems with value parameters and local variables
- interprocedural flow graphs with value parameters and local variables

This extension is straightforward as illustrated next.

FGS w/ Value Parameters and Local Variables



IFG w/ Value Parameters and Local Variables



New Phenomena

...by recursive procedures, value parameters, local variables:

- Existence of a potentially unlimited number of copies (incarnations) of local variables and value parameters due to (mutually) recursive procedure calls at run time.
- After termination of a recursive procedure call the local variables and value parameters of the preceding not yet finished procedure call become valid again.

The run-time system

- handles these phenomena by means of a run-time stack which stores the activation records of the various procedure incarnations.

In data flow analysis

- we have to take these phenomena into account and to model them properly introducing DFA stacks.

Data Flow Analysis Stacks

Intuitively

- DFA stacks are a compile-time equivalent of run-time stacks.
- Entries in DFA stacks are elements (or data flow facts) of an underlying DFA lattics $\widehat{\mathcal{C}}$.
- DFA stacks contain at least one entry abstracting the activation record of the main program; DFA stacks are thus non-empty.

We denote

- the set of all (non-empty) DFA stacks by STACK.

Generating and Manipulating DFA Stacks

DFA stacks can be generated and manipulated by:

1. newstack : $\mathcal{C} \rightarrow STACK$
   newstack(c) generates a new DFA stack with entry c.
2. push : $STACK \times \mathcal{C} \rightarrow STACK$
   push stores a new entry on top of a DFA stack.
3. pop : $STACK \rightarrow STACK$
   pop removes the top-most entry of a DFA stack.
4. top : $STACK \rightarrow \mathcal{C}$
   top yields the contents of the top-most entry of a DFA stack w/out modifying the stack.

Remarks on DFA Stacks (1)

- DFA stack entries are abstractions of the activation records of procedure calls.
- The top-most entry of a DFA stack represents the currently valid activation record.
  - Therefore, DFA stacks are never empty and the commonly considered stack function emptystack : $\to STACK$ yielding an empty stack is replaced by newstack : $\mathcal{C} \to STACK$ yielding a stack with one entry.
- DFA stack entries other than the top-most entry abstract the activation records of started but not yet finished procedure calls.

Remarks on DFA Stacks (2)

- DFA stacks are only conceptually relevant, i.e., for the specifying *IMOP* approach but not for the algorithmic *IMaxFP* approach.
- In fact, the algorithmic *IMaxFP* approach requires only a temporary storing the abstraction of a single activation record instead of a DFA stack.
  - This ensures that
    - the *IMaxFP* solution gets effectively computable (in practically relevant scenarios).
    - push and pop allowing and limited to manipulating the top-most entries of a DFA stack are sufficient for interprocedural DFA though a run-time stack is manipulated much more flexible by the run-time system for performance reasons.

Chapter 12.1.1

IDFA$_{Stk}$ Specifications, IDFA$_{Stk}$ Problems

DFA Functions and Return Functions

Let S be an edge-labelled flow graph system, and let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be the interprocedural flow graph induced by S.

Moreover, let

- $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \perp, \top)$ be a DFA lattice
- $[\![\ ]\!]^* : E^* \to (\mathcal{C} \to \mathcal{C})$ be a DFA functional for G^*
- $\mathcal{R} : E_{call} \to (\mathcal{C} \times \mathcal{C} \to \mathcal{C})$ be a return functional for S

Then $[\![\ ]\!]^*$ and $\mathcal{R}$ induce a DFA_{Stk} functional on DFA stacks

- $[\![\ ]\!]^*_{Stk} : E^* \to (STACK \to STACK)$

for G^* defined next.

Induced DFA$_{Stk}$ Functional on DFA Stacks

Definition 12.1.1.1 (Induced DFA$_{Stk}$ Functional)

The DFA functional $[\![\ ]\!]_{Stk}^{*} : E^{*} \to (STACK \to STACK)$ on DFA stacks induced by $[\![\ ]\!]^{*}$ and $\mathcal{R}$ is defined by

$$\forall\, e \in E^{*}\ \forall\, stk \in STACK.\ [\![\, e\,]\!]_{Stk}^{*}(stk) =_{df}$$

$$\begin{cases} \text{push}(\,\text{pop}(stk),\ [\![\, e\,]\!]^{*}(\text{top}(stk))\,) & \text{if } e \in E^{*} \backslash E_{call}^{*} \\ \text{push}(\, stk,\ [\![\, e\,]\!]^{*}(\text{top}(stk))\,) & \text{if } e \in E_{c}^{*} \\ \text{push}(\,\text{pop}(\text{pop}(stk)),\ \mathcal{R}_{e_S}(\,\text{top}(\text{pop}(stk)), [\![\, e\,]\!]^{*}(\text{top}(stk))\,)\,) & \\ & \text{if } e \in E_{r}^{*} \end{cases}$$

where e_S denotes the call edge of S inducing the return edge $e \in E_{r}^{*}$ of G^{*}.

Interprocedural DFA$_{Stk}$ Specification

Definition 12.1.1.2 (IDFA$_{Stk}$ Specification)

An (interprocedural) DFA$_{Stk}$ specification for G^* is a quintuple $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\![\ ]\!]^*, \mathcal{R}, c_{\mathbf{s}}, d)$ with

- $\widehat{\mathcal{C}} = (\mathcal{C}, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$ a DFA lattice
- $[\![\ ]\!]^* : E^* \to (\mathcal{C} \to \mathcal{C})$ a DFA functional
- $\mathcal{R} : E_{call} \to (\mathcal{C} \times \mathcal{C} \to \mathcal{C})$ a return functional
- $c_{\mathbf{s}} \in \mathcal{C}$ an initial information/assertion
- $d \in \{fw, bw\}$ a direction of information flow

Note: Definition 12.1.1.2 and Definition 11.2.1 differ only by the return functional *ret*. Moreover, DFA stacks need not be dealt with on the level of an IDFA$_{Stk}$ specification.

Interprocedural DFA$_{Stk}$ Problem

Definition 12.1.1.3 (IDFA$_{Stk}$ Problem)

An IDFA$_{Stk}$ specification $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\![\ ]\!]^*, \mathcal{R}, c_{\mathbf{s}}, d)$ defines an interprocedural DFA$_{Stk}$ problem for G^*.

The Structure of DFA_{Stk} Functions

Every DFA_{Stk} function occurring in interprocedural DFA is an element of one of the subsets

- $\mathcal{F}_{ord}$, $\mathcal{F}_{psh}$, $\mathcal{F}_{pop}$

of the set of all functions $\mathcal{F} =_{df} [\, STACK \rightarrow STACK \,]$ on DFA stacks defined by:

$$\mathcal{F}_{ord} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK.\ \mathrm{pop}(f(stk)) = pop(stk) \,\}$$

$$\mathcal{F}_{psh} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK.\ \mathrm{pop}(f(stk)) = stk \,\}$$

$$\mathcal{F}_{pop} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK_{\geq 2}.\ \mathrm{pop}(f(stk)) = \mathrm{pop}(\mathrm{pop}(stk)) \,\}$$

Characterizing DFA$_{Stk}$ Functions

Lemma 12.1.1.4

$\forall\, f_{pp} \in \mathcal{F}_{pop} \;\; \forall\, f_o, f'_o \in \mathcal{F}_{ord} \;\; \forall\, f_{ph} \in \mathcal{F}_{psh}.$

1. $f_o \circ f'_o \in \mathcal{F}_{ord}$
2. $f_{pp} \circ f_o \circ f_{ph} \in \mathcal{F}_{ord}$

Lemma 12.1.1.5

1. $\forall\, e \in E^* \backslash E^*_{call}.\; [\![\, e \,]\!]^*_{Stk} \in \mathcal{F}_{ord}$
2. $\forall\, e \in E^*_c.\; [\![\, e \,]\!]^*_{Stk} \in \mathcal{F}_{psh}$
3. $\forall\, e \in E^*_r.\; [\![\, e \,]\!]^*_{Stk} \in \mathcal{F}_{pop}$

The Significant Function of DFA_{Stk} Functions

At most the top or the two top-most entries of DFA stacks are modified by DFA_{Stk} functions (cf. Lemma 12.1.1.5). This gives rise to the following definition:

Definition 12.1.1.6 (Significant Function)

- Let $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh}$: Then $f_{sig} : \mathcal{C} \to \mathcal{C}$ is defined by:
  $f_{sig}(c) =_{df} \text{top}(f(\text{newstack}(c)))$
- Let $f \in \mathcal{F}_{pop}$: Then $f_{sig} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined by:
  $f_{sig}(c_1, c_2) =_{df} \text{top}(f(\text{push}(\text{newstack}(c_1), c_2)))$
  (Recall that $\mathcal{C} \times \mathcal{C}$ is a lattice, if $\mathcal{C}$ is a lattice.)

The functions f_{sig} are the significant functions of the DFA_{Stk} functions f.

Characterizing DFA$_{Stk}$ Functions (Cont'd)

...via significant functions:

Lemma 12.1.1.7

1. $\forall\, e \in E^* \backslash E_r^*.\ [\![\, e \,]\!]^*_{Stk\,sig} = [\![\, e \,]\!]^*$
2. $\forall\, e \in E_r^*\ \forall\, c_1, c_2 \in \mathcal{C} \times \mathcal{C}.\ [\![\, e \,]\!]^*_{Stk\,sig} = \mathcal{R}_{e_S}(c_1, [\![\, e \,]\!]^*(c_2))$
   where e_S denotes the call edge of S inducing the return edge $e \in E_r^*$ of G^*.

S-Monotonicity, S-Distributivity

Definition 12.1.1.8 (S-Monontonicity., S-Distrib.)

A DFA_{Stk} function $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh} \cup \mathcal{F}_{pop}$ is

1. s-monotonic iff f_{sig} is monotonic
2. s-distributive iff f_{sig} is distributive

Characterizing DFA$_{Stk}$ Functions (Cont'd)

Lemma 12.1.1.9

Let $e \in E^*$. The function $[\![\, e \,]\!]^*_{Stk}$ is s-monotonic (s-distributive), if

- $e \in E^* \backslash E^*_r$: $[\![\, e \,]\!]^*$ is monotonic (distributive)
- $e \in E^*_r$: $[\![\, e \,]\!]^*$ and $\mathcal{R}_{e_S}$ are monotonic (distributive) where e_S denotes the call edge of S inducing the return edge $e \in E^*_r$ of G^*.

Conventions

In the following, we

- identify lattice elements with their representation as a DFA stack with just a single entry.
- extend the meet and join operation $\sqcap$ and $\sqcup$ on DFA lattices in the following fashion to (the top most entries of) sets of DFA stacks $STK \subseteq STACK$:

$$\bigsqcap STK =_{df} \text{newstack}(\bigsqcap \{ top(stk) \mid stk \in STK \})$$

$$\bigsqcup STK =_{df} \text{newstack}(\bigsqcup \{ top(stk) \mid stk \in STK \})$$

This allows us

- to consider s-monotonicity and s-distributivity generalizations of the usual monotonicity and distributivity properties.

Chapter 12.1.2

The $IMOP_{Stk}$ Approach

The $IMOP_{Stk}$ Approach & $IMOP_{Stk}$ Solution

Let $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\![\ ]\!]^*, \mathcal{R}, c_{\mathbf{s}}, d)$ be a DFA_{Stk} specification for G^*.

Definition 12.1.2.1 (The $IMOP_{Stk}$ Solution)

The $IMOP_{Stk}$ solution of $\mathcal{S}_{G^*}$ is defined by:

$$IMOP_{Stk}^{\mathcal{S}_{G^*}} : N^* \to STACK_1$$

$$\forall\, n \in N^*.\ IMOP_{Stk}^{\mathcal{S}_{G^*}}(n) =_{df} \prod \{\, [\![\, p \,]\!]^*_{Stk}(\text{newstack}(c_{\mathbf{s}})) \mid p \in \mathbf{IP}[\mathbf{s}, n] \,\}$$

where $STACK_1$ denotes the set of DFA stacks with exactly one entry.

Conservative and Optimal IDFA$_{Stk}$ Algorithms

Definition 12.1.2.2 (Conservative IDFA Algorithm)

An IDFA algorithm A is $IMOP_{Stk}$ conservative for $\mathcal{S}_{G^*}$, if A terminates with a lower approximation of the $IMOP_{Stk}$ solution of $\mathcal{S}_{G^*}$.

Definition 12.1.2.3 (Optimal IDFA Algorithm)

An IDFA algorithm A is $IMOP_{Stk}$ optimal for $\mathcal{S}_{G^*}$, if A terminates with the $IMOP_{Stk}$ solution of $\mathcal{S}_{G^*}$.

Chapter 12.1.3

The IMaxFP_{Stk} Approach

The IMaxFP_{Stk} Approach

...is a two-stage approach:

- Stage 1: Preprocess – Computing the Semantics of Procedures
- Stage 2: Main Process – Computing the IMaxFP_{Stk} solution

Preliminaries

Let

- Id_{STACK} denote the identity on $STACK$, 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 $\forall\, stk \in STACK.\ \ topl(f''(stk)) = \text{top}(f(stk)) \sqcap \text{top}(f'(stk))$.
- "$\sqcap$" induces an inclusion relation "$\sqsubseteq$" on $\mathcal{F}_{ord}$ by:

$$f \sqsubseteq f' \text{ iff } f \sqcap f' = f.$$

The $IMaxFP_{Stk}$ Approach: 2nd Order

Stage 1: Preprocess – Computing the Semantics of Procedures

Equation System 12.1.3.1 (2nd Order $IMaxFP_{Stk}$)

$$[\![\, n \,]\!]_{Stk} = \begin{cases} Id_{STACK} & \text{if } n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\} \\ \bigsqcap \{ [\![\, (m,n) \,]\!]_{Stk} \circ [\![\, m \,]\!]_{Stk} \mid m \in pred_{flowGraph(n)}(n) \} & \text{otherwise} \end{cases}$$

and

$$[\![\, e \,]\!]_{Stk} = \begin{cases} [\![\, e \,]\!]^{*}_{Stk} & \text{if } e \in E \backslash E_{call} \\ [\![\, e_r \,]\!]^{*}_{Stk} \circ [\![\, end(callee(e)) \,]\!]_{Stk} \circ [\![\, e_c \,]\!]^{*}_{Stk} & \text{otherwise} \end{cases}$$

Let

$$[\![\, n \,]\!]^{\star}_{Stk}, n \in N, \quad [\![\, e \,]\!]^{\star}_{Stk}, e \in E$$

denote the greatest solutions of Equation System 12.1.3.1.

The IMaxFP_{Stk} Approach: 1st Order

Stage 2: Main Process – The "Actual" Interprocedural DFA

Equation System 12.1.3.2 (1st Order IMaxFP_{Stk})

$$\mathit{inf}(n) = \begin{cases} \text{newstack}(c_{\mathbf{s}}) & \text{if } n = \mathbf{s}_0 \\ \bigsqcap \{\, [\![\, e_c \,]\!]^{*}_{Stk}(\mathit{inf}(\mathit{src}(e))) \mid e \in \mathit{caller}(\mathit{flowGraph}(n)) \,\} & \\ & \text{if } n \in \mathit{start}(S) \backslash \{\mathbf{s}_0\} \\ \bigsqcap \{[\![\, (m,n) \,]\!]^{\star}_{Stk}(\mathit{inf}(m)) \mid m \in \mathit{pred}_{\mathit{flowGraph}(n)}(n)\} & \\ & \text{otherwise} \end{cases}$$

Let

$$\mathit{inf}^{\star}_{c_{\mathbf{s}}}(n),\, n \in N$$

denote the greatest solution of Equation System 12.1.3.2.

The $IMaxFP_{Stk}$ Solution

Let $\mathcal{S}_{G^*} = (\widehat{\mathcal{C}}, [\![\ ]\!]^*, \mathcal{R}, c_{\mathsf{s}}, d)$ be a DFA$_{Stk}$ specification for G^*.

Definition 12.1.3.3 (The $IMaxFP_{Stk}$ Solution)

The $IMaxFP_{Stk}$ solution of $\mathcal{S}_{G^*}$ is defined by:

$$IMaxFP_{Stk}^{\mathcal{S}_{G^*}} : N^* \to STACK_1$$

$$\forall\, n \in N^*.\ IMaxFP_{Stk}^{\mathcal{S}_{G^*}}(n) =_{df} inf^{\star}_{c_{\mathsf{s}}}(n)$$

where $STACK_1$ denotes the set of DFA stacks with exactly one entry.

Note that $N = N^*$ allowing us to identify corresponding nodes of S and G^*.

Chapter 12.1.4

Safety and Coincidence

Towards the Main Results

...on safety and coincidence.

Lemma 12.1.4.1

For all $n \in N$ the semantic functions $[\![\, e \,]\!]^*$, $e \in E^*$, are

1. s-monotonic: $[\![\, n \,]\!]^\star_{Stk} \sqsubseteq imop_n$
2. s-distributive: $[\![\, n \,]\!]^\star_{Stk} = imop_n$

where $imop_n : N \to (STACK \to STACK)$ denotes a functional that is defined by:

$$\forall\, n \in N.\ imop_n =_{df} \begin{cases} Id_{STACK} & \text{if } n \in start(S) \\ \bigsqcap \{ [\![\, p \,]\!]^*_{Stk} \mid p \in \mathbf{CIP}[start(flowGraph(n)), n] \} & \text{otherwise} \end{cases}$$

Main Results: 2nd Order Analysis

Safety and coincidence results of the 2nd order analysis:

Theorem 12.1.4.2 (2nd Order Analysis)

For all $e \in E_{call}$ we have:

1. $[\![\, e \,]\!]^{\star}_{Stk} \sqsubseteq \bigsqcap \{ [\![\, p \,]\!]^{*}_{Stk} \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, if the data flow analysis functional $[\![\ ]\!]^{*}_{Stk}$ is s-monotonic.
2. $[\![\, e \,]\!]^{\star}_{Stk} = \bigsqcap \{ [\![\, p \,]\!]^{*}_{Stk} \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$ if the data flow analysis functional $[\![\ ]\!]^{*}_{Stk}$ is s-distributive.

where the mappings *src* and *dst* yield the start and the final node of an edge, respectively.

Main Results: 1st Order Analysis

Safety and coincidence results of the 1st order analysis:

Theorem 12.1.4.3 (Interprocedural Safety)

The $IMaxFP_{Stk}$ solution of $\mathcal{S}_{G^*}$ is a safe (i.e., lower) approximation of the $IMOP_{Stk}$ solution of $\mathcal{S}_{G^*}$, i.e.,

$$\forall\, n \in N.\ IMaxFP_{Stk}^{\mathcal{S}_{G^*}}(n) \sqsubseteq IMOP_{Stk}^{\mathcal{S}_{G^*}}(n)$$

if the DFA functional $[\![\ ]\!]_{Stk}^*$ is s-monotonic.

Theorem 12.1.4.4 (Interprocedural Coincidence)

The $IMaxFP_{Stk}$ solution of $\mathcal{S}_{G^*}$ coincides with the $IMOP_{Stk}$ solution of $\mathcal{S}_{G^*}$, i.e.,

$$\forall\, n \in N.\ IMaxFP_{Stk}^{\mathcal{S}_{G^*}}(n) = IMOP_{Stk}^{\mathcal{S}_{G^*}}(n)$$

if the DFA functional $[\![\ ]\!]_{Stk}^*$ is s-distributive.

Chapter 12.1.5

The Generic Fixed Point Algorithms

Algorithms

The generic fixed point algorithms of Chapter 11.6

- can straightforwardly be extended to stack-based ones.
- This way, we receive
  - the standard variant of preprocess and main process
  - the more efficient variant of preprocess and functional main process.
  - a demand-driven "by-need" variant interleaving the 1st and 2nd order analyses.

In the following, we present

- another variant, which is stackless.
- The clou of this variant is that stacks have at most 2 entries during analysis time.
- Therefore, a single temporary storing the temporarily existing stack entry during procedure calls suffices for the implementation.

Algorithm 12.1.5.1 – Stackless 2nd Order Preprocess

Input: (1) A flow-graph system S, and (2) an abstract semantics consisting of a data-flow lattice $\mathcal{C}$, and a data-flow functional $[\![\ ]\!]^* : E^* \to (\mathcal{C} \to \mathcal{C})$.

Output: Under the assumption of termination (cf. Theorem 12.1.5.4), 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 Equation System 12.1.3.1.

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S. Note that due to the mutual interdependence of the definitions of $[\![\![\ ]\!]\!]$ 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.

Algorithm 12.1.5.1 – Stackless 2nd Order Preprocess

(Prologue: Initialization of the annotation arrays *gtr* and *ltr* and the variable *workset*)

FORALL $n \in N$ DO
  IF $n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ 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; $\langle\star\rangle$
$workset := \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$;

Algorithm 12.1.5.1 – Stackless 2nd Order Preprocess

(Main process: Iterative fixed point computation)
WHILE $workset \neq \emptyset$ DO
    CHOOSE $m \in workset$;
        $workset := workset \backslash \{ m \}$;
        (Update the successor-environment of node m)
        IF $m \in \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$
            THEN
                FORALL $e \in caller(flowGraph(m))$ DO
                    $ltr[e] := \mathcal{R}_e \circ (Id_{\mathcal{C}}, [\![\, e_r \,]\!]^* \circ gtr[m] \circ [\![\, e_c \,]\!]^*);$ $\langle\star\rangle$
                    $meet := ltr[e] \circ gtr[src(e)] \sqcap gtr[dst(e)];$
                    IF $gtr[dst(e)] \sqsupset meet$
                        THEN
                            $gtr[dst(e)] := meet;$
                            $workset := workset \cup \{dst(e)\}$
                    FI
                OD

Algorithm 12.1.5.1 – Stackless 2nd Order 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.
```

Algorithm 12.1.5.2 – Stackless 1st Order 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 16.6.1, and (3) a context information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 12.1.5.4), the $\mathit{IMaxFP}_{\mathit{StkLss}}$-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 $c_{\mathbf{s}}$.
(2) $[\![\ ]\!]$ is *monotonic*: variable *inf* stores for every node a valid component information with respect to 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.

Algorithm 12.1.5.2 – Stackless 1st Order 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;

Algorithm 12.1.5.2 – Stackless 1st Order 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.

Algorithm 12.1.5.3 – Stackless "Functional" 1st Order 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 12.1.5.1), and (4) a context information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 12.1.5.4), the $\mathit{IMaxFP}_{\mathit{StkLss}}$-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 $c_{\mathbf{s}}$.
(2) $[\![\ ]\!]$ is monotonic: variable *inf* stores for every node a valid component information with respect to 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, and the temporary *meet* stores the most recent approximation.

Algorithm 12.1.5.3 – Stackless "Functional" 1st Order Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)
FORALL $\mathbf{s} \in \{\mathbf{s}_i \mid i \in \{1, \ldots, k\}\}$ DO $inf[\mathbf{s}] := \top$ OD;
$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;
$workset := \{\mathbf{s}_i \mid i \in \{1, 2, \ldots, k\}\}$;

Algorithm 12.1.5.3 – Stackless "Functional" 1st Order Main Process

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO

CHOOSE $\mathbf{s} \in workset$;

$workset := workset \backslash \{\mathbf{s}\}$;

$meet := inf[\mathbf{s}] \sqcap$

$\bigsqcap\{ [\![\, e_c \,]\!]^* \circ [\![\![\, src(e) \,]\!]\!](inf[start(flowGraph(e))]) \mid$

$e \in caller(flowGraph(\mathbf{s}))\,\};$ $\quad\langle\star\rangle$

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;

Algorithm 12.1.5.3 – Stackless "Functional" 1st Order 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.

Termination

Theorem 12.1.5.4 (Termination)

The sequential composition of Algorithm 12.1.5.1 (2nd order) and Algorithmus 12.1.5.2 (1st order) resp. Algorithm 12.1.5.3 (1st order) terminates with the IMaxFP_{Stk} solution, if the DFA functional $[\![\ ]\!]^*$ and the return functional $\mathcal{R}$ are monotonic and the function lattice $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Note: Validity of the descending chain condition on the function lattice $[\mathcal{C} \rightarrow \mathcal{C}]$ implies validity of the descending chain condition on the underlying lattice $\widehat{\mathcal{C}}$.

Conservativity, Optimality of IDFA Algorithms

Corollary 12.1.5.5 (IMOP_{Stk} Conservativity)

The IDFA algorithms of Chapter 12.1.5 are IMOP_{Stk} conservative for $\mathcal{S}_{G^*}$ (i.e., terminate with a lower approximation of the IMOP_{Stk} solution of $\mathcal{S}_{G^*}$), if $[\![\ ]\!]^*$ and $\mathcal{R}$ are monotonic and $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition.

Corollary 12.1.5.6 (IMOP_{Stk} Optimality)

The IDFA algorithms of Chapter 12.1.5 are IMOP_{Stk} optimal for $\mathcal{S}_{G^*}$ (i.e., terminate with the IMOP_{Stk} solution of $\mathcal{S}_{G^*}$), if $[\![\ ]\!]^*$ and $\mathcal{R}$ are distributive and $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition.

Chapter 12.1.6

Soundness and Completeness

Soundness and Completeness (1)

Analysis Scenario:

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

Definition 12.1.6.1 (Soundness)

$\mathcal{S}^{\phi}_{G*}$ is sound for ϕ, if, whenever the $IMOP_{Stk}$ solution of $\mathcal{S}^{\phi}_{G*}$ indicates that ϕ is valid, then ϕ is valid.

Definition 12.1.6.2 (Completeness)

$\mathcal{S}^{\phi}_{G*}$ is complete for ϕ, if, whenever ϕ is valid, then the $IMOP_{Stk}$ solution of $\mathcal{S}^{\phi}_{G*}$ indicates that ϕ is valid.

Soundness and Completeness (2)

Intuitively

- Soundness means: $IMOP_{Stk}^{\mathcal{S}_{G*}^{\phi}}$ implies ϕ.
- Completeness means: ϕ implies $IMOP_{Stk}^{\mathcal{S}_{G*}^{\phi}}$.

Soundness and Completeness (3)

If $\mathcal{S}^{\phi}_{G*}$ is sound and complete for ϕ, this intuitively means:

We compute

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

In other words

- We compute the program property of interest accurately!

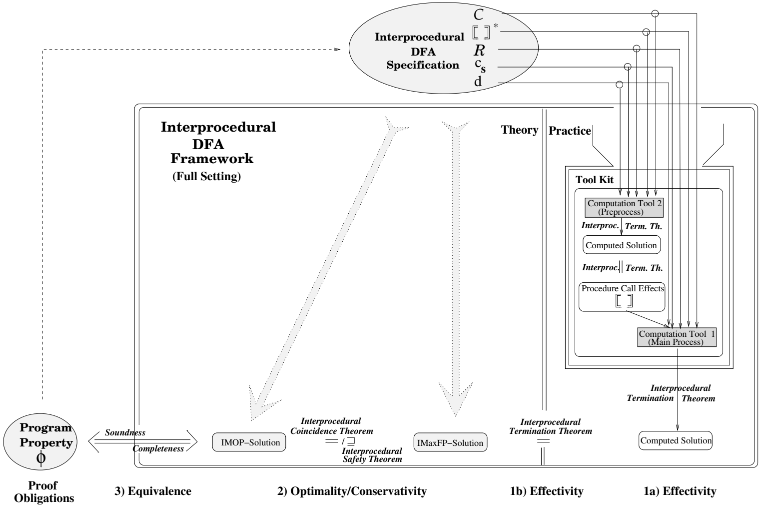
Chapter 12.1.7

A Uniform Framework and Toolkit View

IDFA with Value Parameters, Local Variables

...considered from a holistic angle yields a

- Uniform Framework and Toolkit View



Chapter 12.2

Adding Procedural Parameters

Procedural Parameters

Let $\Pi = \langle \pi_0, \ldots, \pi_k \rangle$ be a program.

So far, we considered

- Procedure declarations: *proc* $\pi(p_1, \ldots, p_r)$
- Ordinary procedure call: *call* $\pi(a_1, \ldots, a_r)$, $\pi \in \{\pi_1, \ldots, \pi_k\}$

Now, we introduce procedural parameters allowing

- Procedure declarations: *proc* $\pi(p_1, \ldots, p_r, \psi_1, \ldots, \psi_q)$
- Formal procedure call: *call* $\psi(a_1, \ldots, a_r, \bar{\pi}_1, \ldots, \bar{\pi}_q)$, $\bar{\pi}_1, \ldots, \bar{\pi}_q \in \{\pi_1, \ldots, \pi_k\}$

Handling Procedural Parameters in IDFA (1)

Key idea

- The semantics of a formal procedure call ψ is considered the meet of the semantics of the ordinary procedures it might call at runtime:

$$\bigsqcap \{ [\![\, e_r \,]\!]^*_{Stk} \circ [\![\, end(\pi) \,]\!]^\star_{Stk} \circ [\![\, e_c \,]\!]^*_{Stk} \mid \psi \text{ might call } \pi \}$$

Technically, this means

- replacing formal procedure calls by the set of ordinary procedure calls that they might stand for.

Handling Procedural Parameters in IDFA (2)

Algorithmically

- this set of procedures can be computed by a suitable preprocess.
- depending on the expressive power of the programming language considered, the specific program under investigation, and the power of the analysis algorithm, the computed set of procedures can be exact or a safe approximation.

Overall

- exploiting precomputed calling information for formal procedure call reduces the analysis of programs with formal procedure calls to the analysis of programs without formal procedure calls.

Replacing the Basic 2nd Order Analysis...

...of the *IMaxFP$_{Stk}$* approach characterized by

Equation System 12.1.3.1 (2nd Order *IMaxFP$_{Stk}$*)

$$[\![\, n \,]\!]_{Stk} = \begin{cases} Id_{STACK} & \text{if } n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\} \\ \bigsqcap\{[\![\,(m,n)\,]\!]_{Stk} \circ [\![\, m \,]\!]_{Stk} \mid m \in pred_{flowGraph(n)}(n)\} \\ & \text{otherwise} \end{cases}$$

and

$$[\![\, e \,]\!]_{Stk} = \begin{cases} [\![\, e \,]\!]^*_{Stk} & \text{if } e \in E \backslash E_{call} \\ [\![\, e_r \,]\!]^*_{Stk} \circ [\![\, end(callee(e)) \,]\!]_{Stk} \circ [\![\, e_c \,]\!]^*_{Stk} & \text{otherwise} \end{cases}$$

...by the Enhanced 2nd Order Analysis

Equation System 12.2.1 (Enh'd 2nd Order *IMaxFP*)

$$[\![\, n \,]\!]_{Stk} = \begin{cases} Id_{STACK} & \text{if } n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\} \\ \bigsqcap\{[\![\, (m,n) \,]\!]_{Stk} \circ [\![\, m \,]\!]_{Stk} \mid m \in pred_{flowGraph(n)}(n)\} & \\ & \text{otherwise} \end{cases}$$

and $[\![\, e \,]\!]_{Stk} =$

$$\begin{cases} [\![\, e \,]\!]^*_{Stk} & \text{if } e \in E \backslash E_{call} \\ [\![\, e_r \,]\!]^*_{Stk} \circ [\![\, end(callee(e)) \,]\!]_{Stk} \circ [\![\, e_c \,]\!]^*_{Stk} & \\ & \text{if } e \in E_{call}, \text{ ordinary procedure call} \\ \bigsqcap\{[\![\, e_r \,]\!]^*_{Stk} \circ [\![\, end(\pi) \,]\!]_{Stk} \circ [\![\, e_c \,]\!]^*_{Stk} \mid \psi \text{ might call } \pi\} & \\ & \text{if } e \in E_{call}, \text{ formal procedure call} \end{cases}$$

where the set of procedures that may be called by a formal call is computed in a separate and independent preprocess.

Suitable Preprocesses

...are sketched in Chapter 15 for an object-oriented setting:

- Class Hierarchy Analysis (ch. Chapter 15.2.1)
- Rapid Type Analysis (ch. Chapter 15.2.2)

Chapter 12.3

Adding Reference Parameters

Reference Parameters

Handling the effect of reference parameters

- The effect of reference parameters can be encoded in the DFA functionals of the application problems.

Algorithmically

- This requires may and must alias information of of variables and parameters, which can be computed by suitable preprocesses (cf. Chapter 14).
- The computed alias information is then fed into the generic algorithms of the toolkit of Chapter 12.1.7 via the definitions of the DFA functions of the application problems (cf. Chapter 12.5).

Chapter 12.4

Adding Static Procedure Nesting

Static Procedure Nesting

Static nesting of procedures

- introduces statically nested definitions (or declarations) of local variables: Variables are no longer either global (declared in the main program) or local (declared in a procedure) but "relatively global."

Algorithmically

- The effects relatively global variables can be encoded in the DFA functionals of the application problems.

Alternatively

- De-nesting of procedures by a suitable preprocess.
- This way, the analysis of programs with static procedure nesting is reduced to analysing programs without static procedure nesting.

Chapter 12.5

Applications

Chapter 12.5.1

Interprocedural Availability

Preliminaries

In the following we assume:

- No static procedure nesting, no procedural parameters.
- $MstAliases_G(v)$ und $MayAliases_G(v)$ denote the sets of must-aliases and may-aliases different from v.

These notions can straightforward be extended to terms t:

- A term t' is a must-alias (may-alias) of t, if t' results from t by replacing of variables by variables that are must-aliases (may-aliases) of each other.

This allows us to feed alias information in a parameterized fashion into the definitions of DFA functionals and return functionals and to take their effects during the analysis into account.

Useful Notations

We define:

- $GlobVar(S)$: the set of global variables of S, i.e., the set of variables which are declared in the main program of S. They are accessible in each procedure of S.
- $Var(t)$: the set of variables occurring in t.
- $LhsVar(e)$: the left hand side variable of the assignment of edge e.
- $GlobId(t)$ and $LocId(t)$: abbreviations of $GlobVar(S) \cap Var(t)$ and $Var(t) \backslash GlobVar(S)$.

Useful Notations (Cont'd)

- *NoGlobalChanges* $: E^* \to \mathbb{B}$: indicates that a modification of variable $v \in Var(t)$ by e will not be visible after finishing the call as the relevant memory location of v is local for the currently active call.
- *PotAccessible* $: S \to \mathbb{B}$: 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.

Local Predicates

The definition of the preceding functions utilizes the predicates $Transp_{LocId}$ and $Transp_{GlobId}$ 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:

$$\forall\, e \in E^*.\ NoGlobalChanges(e) \\ =_{df} \begin{cases} true & \text{if } e \in E_c^* \cup E_r^* \\ Transp_{LocId}(n) \ \wedge\ Transp_{GlobId}(n) & \text{otherwise} \end{cases}$$

Parameterized Local Predicates

...parameterized wrt alias information:

$$\forall\, e \in E^*.\ \textit{A-Comp}_{\,e} =_{df} \textit{Comp}_e \ \vee \ \textit{Comp}_e^{\textit{MstAl}}$$

$$\forall\, e \in E^*.\ \textit{A-Transp}_{\,e} =_{df} \ \textit{Transp}_e \ \wedge \ \begin{cases} \textit{true} & \text{if } e \in E^*_{call} \\ \textit{Transp}_e^{\textit{MayAl}} & \text{otherwise} \end{cases}$$

Parameterized Local Predicates (Cont'd)

Intuitively

- *A-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-Transp* $_e$, $e \in E^* \backslash E^*_{call}$, is true for t, if neither an operand of t (i.e., $Transp_e$) nor one of its may-aliases is modified by the statement at edge e (i.e., $Transp_e^{MayAl}$).
- For call and return edges $e \in E^*_{call}$, *A-Transp* $_e$ is true for t, if no operand of t is modified (i.e., $Transp_e$). This makes the difference between ordinary assignments and reference parameters and parameter transfers to reference parameters; the latter are updates of pointers that leave the memory invariant except of that update.

Interprocedural Availability

Key Ingredients of the DFA Specification:

1. Data flow lattice:

$$(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\, \mathbb{B}^2, \wedge\,, \vee\,, \leq, (\mathit{false}, \mathit{false}), (\mathit{true}, \mathit{true})\,)$$

2. Data flow functional:

$[\![\ ]\!]^*_{av} : E^* \rightarrow (\, \mathbb{B}^2 \rightarrow \mathbb{B}^2\,)$ defined by

$$\forall\, e \in E^*\ \forall\, (b_1, b_2) \in \mathbb{B}^2.\ [\![\, e\,]\!]^*_{av}(b_1, b_2) =_{df} (b_1', b_2')$$

where

$$b_1' =_{df} \mathit{A\text{-}Transp}_{\,e} \wedge (\mathit{A\text{-}Comp}_{\,e} \vee b_1)$$

$$b_2' =_{df} \begin{cases} b_2 \wedge \mathit{NoGlobalChanges}_{\,e} & \text{if } e \in E^* \backslash E_c^* \\ \mathit{true} & \text{otherwise} \end{cases}$$

Interprocedural Availability (Cont'd)

3. Return functional:

   $\mathcal{R}_{av} : E_{call} \to (\, \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2\,)$ defined by

   $\forall\, e \in E_{call}\ \forall (\, (b_1, b_2),\ (b_3, b_4)\,) \in \mathbb{B}^2 \times \mathbb{B}^2.$

   $\mathcal{R}_{av}(e)(\, (b_1, b_2),\ (b_3, b_4)\,) =_{df} (b_5, b_6)$ where

$$b_5 =_{df} \begin{cases} b_3 & \text{if } PotAccessible(callee(e)) \\ (b_1 \vee A\text{-}Comp_{\,e}) \wedge b_4 & \text{otherwise} \end{cases}$$

$$b_6 =_{df} b_2 \wedge b_4$$

Findings on Interprocedural Availability

Lemma 12.5.1.1

1. The lattice $\mathbb{B}^2$ and the induced lattice of functions satisfy the descending chain condition.
2. The functionals $[\![\ ]\!]_{av}^*$ and $\mathcal{R}_{av}$ are distributive.

This means, the preconditions of the Interprocedural Coincidence Theorem 12.1.4.4 and the Termination Theorem 12.1.5.4 are satisfied.

Chapter 12.5.2

Interprocedural Constant Propagation

Interprocedural Simple Constants – Naively

Key Ingredients of the DFA Specification:

1. Data flow lattice:
   $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\Sigma, \sqcap, \sqcup, \sqsubseteq, \sigma_\bot, \sigma_\top)$
2. Data flow functional: $[\![\ ]\!]^*_{sc} : E \rightarrow (\Sigma \rightarrow \Sigma)$ defined by
   $\forall\, e \in E.\ [\![\, e\,]\!]^*_{sc} =_{df} \theta_e$
3. Return functional: $\mathcal{R}_{sc} : E_{call} \rightarrow (\Sigma \times \Sigma \rightarrow \Sigma)$ defined by

$$\forall\, e \in E_{call}\ \forall\, (\sigma_1, \sigma_2)) \in \Sigma \times \Sigma.\ \mathcal{R}_{sc}(e)(\sigma_1, \sigma_2) =_{df} \sigma_3$$

where

$$\forall\, x \in Var.\ \sigma_3(x) =_{df} \begin{cases} \sigma_2(x) & \text{if } x \in GlobVar(S) \\ \sigma_1(x) & \text{otherwise} \end{cases}$$

Problems, Consequences

Unfortunately

- The preceding DFA specification for interprocedural simple constants does not induce a terminating analysis since the lattice of functions on Σ does not satisfy the descending chain condition.

In practice, thus

- simpler variants of the constant propagation problem are considered interprocedurally, e.g., interprocedural copy constants and linear constants.

Copy Constants, Linear Constants Recalled

Intuitively, a term is a

- copy constant at a program point, if it is a source-code constant or an operator-less term that is itself a copy constant (cf. Chapter 5.5)
- linear constant at a program point, if it is a source-code constant or of the form $a * x + b$ with a, b source-code constants and x a linear constant (cf. Chapter 5.4).

Interprocedural Copy Constants: Findings (1)

We have:

- The number of source-code constants and program variables are finite.
- Hence, the lattice of functions induced by the relevant sublattice of Σ for copy constants is finite satisfying thus the descending chain condition.
- Hence, the generic 2nd order and 1st order DFA algorithms for copy constants terminate with the the $IMaxFP_{Stk}$ solution for copy constants.
- Last but not least, the computable $IMaxFP_{Stk}$ solution for copy constants coincides with the specifying $IMOP_{Stk}$ solution, since the DFA functions $[\![\cdot]\!]^*_{cc}$ and the return functions $\mathcal{R}_{cc.}$ for copy constants are distributive.

Interprocedural Copy Constants: Findings (2)

Lemma 12.5.2.1

1. The lattice $\Sigma_{cc} \subseteq \Sigma$ and its induced lattice of functions satisfy the descending chain condition.
2. The functionals $[\![\ ]\!]^*_{cc}$ and $\mathcal{R}_{cc}$ are distributive.

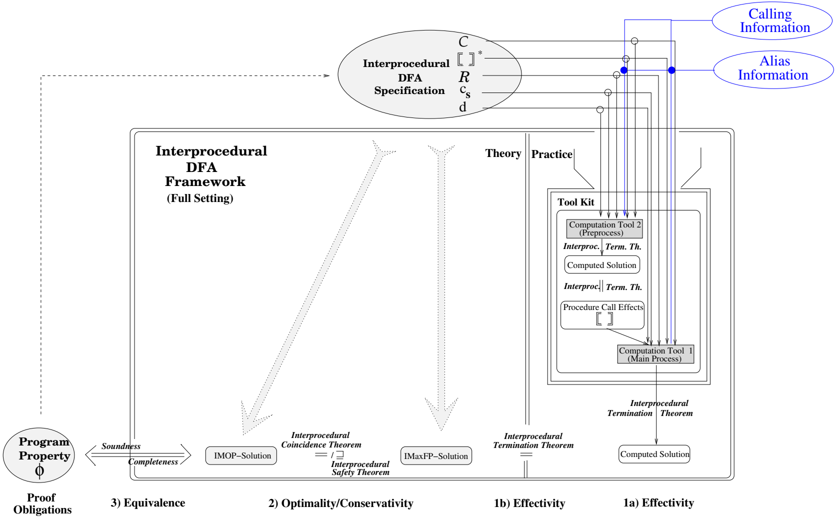
This means, the preconditions of the Interprocedural Coincidence Theorem 12.1.4.4 and the Termination Theorem 12.1.5.4 are satisfied.

Chapter 12.6

Summary, Looking Ahead

...considered from a holistic angle yields a

- Uniform Framework and Toolkit View

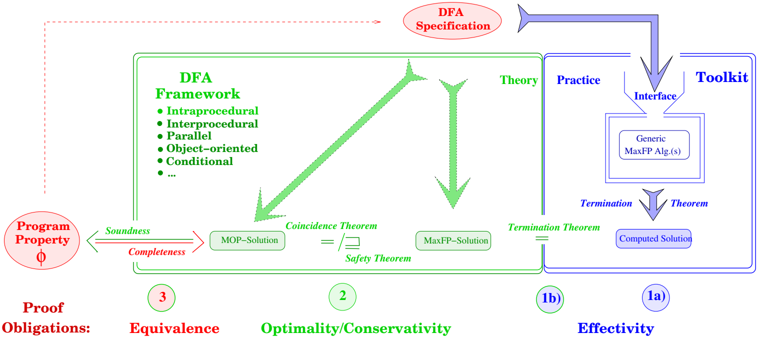


Overall, this provides

...further evidence for the claim of Chapter 3.8 that

- The Uniform Framework and Toolkit View of DFA

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



Chapter 12.7

References, Further Reading

Further Reading for Chapter 12 (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 12, Interprocedural Analysis)
- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 11, Interprocedural Analysis and Optimization)
- Thomas Ball, Sriram K. Rajamani. *Bebop: A Path-Sensitive Interprocedural Dataflow Engine*. In Proceedings of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), 97-103, 2001.

Further Reading for Chapter 12 (2)

- Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 7, Introduction to Interprocedural Data Flow Analysis, Chapter 8, Functional Approach to Interprocedural Data Flow Analysis; Chapter 9, Value-Based Approach to Interprocedural Data Flow Analysis)
- Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998. (Chapter 10, Interprocedural Code Motion: The Transformations, Chapter 11, Interprocedural Code Motion: The IDFA-Algorithms)

Further Reading for Chapter 12 (3)

- Jens Knoop. *Formal Callability and its Relevance and Application to Interprocedural Data Flow Analysis*. In Proceedings of the 6th IEEE International Conference on Computer Languages (ICCL'98), 252-261, 1998.
- Jens Knoop. *From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Springer-V., LNCS 1579, 360-374, 1999.
- Jens Knoop, Bernhard Steffen. *The Interprocedural Coincidence Theorem*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-V., LNCS 641, 125-140, 1992.

Further Reading for Chapter 12 (4)

- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 19, Interprocedural Analysis and Optimization)
- Tom Reps, Susan Horwitz, Mooly Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 49-61, 1995.
- Mooly Sagiv, Tom Reps, Susan Horwitz. *Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation*. In Proceedings of the 6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95), Springer-V., LNCS 915, 651-665, 1995.

Further Reading for Chapter 12 (5)

- Helmut Seidl, Christian Fecht. *Interprocedural Analyses: A Comparison*. The Journal of Logic Programming 43:123-156, 2000.
- Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.3, The Functional Approach to Interprocedural Analysis, 196-209.

Chapter 13

The Context Information Approach

Motivation

In this chapter, we complement the functional approach for IDFA by sketching a selection of so-called

- context information approaches.

Context information approaches

- allow the user to control the trade-off between power and performance
- promise to be more efficient in practice
- are heuristic in nature.

Outline

The presentation follows the one of Nielson, Nielson, and Hankin (2005) using their (extended) setting and notation of Chapter 2.

We start by extending the programming language WHILE by introducing programs with

- top-level declarations of global mutually recursive procedures and
- a call-by-value and a call-by-result parameter.

  Note: Extensions to multiple call-by-value, call-by-result, and call-by-value-result parameters are straightforward.

Chapter 13.1

Preliminaries, the Setting

Syntax: Introducing Procedures

Extended WHILE-Language WHILE$_\pi$:

$$\begin{array}{lcl} P_\star & ::= & \text{begin } D_\star \; S_\star \text{ end} \\ D & ::= & D; D \;\mid\; \text{proc } p(\text{val } x;\ \text{res } y) \text{ is}^{\ell_n} \; S \text{ end}^{\ell_x} \\ S & ::= & \ldots \;\mid\; [\text{call } p(a, z)]^{\ell_c}_{\ell_r} \end{array}$$

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

Assumptions

We assume that

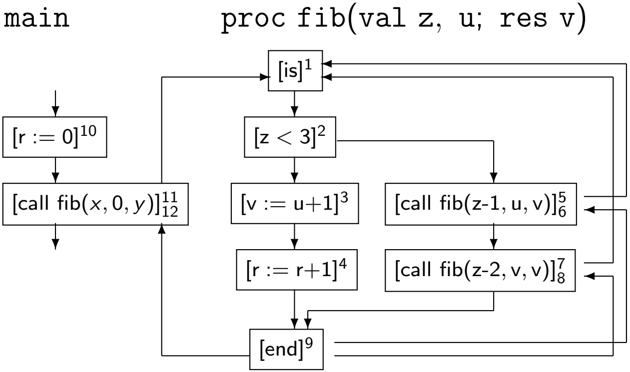
- WHILE_π is statically scoped.
- The parameter mechanism is
  - call-by-value for the first parameter
  - call-by-result for the second parameter.
- Procedures may be mutually recursive.
- Programs are uniquely labelled.
- There are no procedures of the same name.
- Only procedures may be called by a program that have been declared in it.

Illustrating Example

The procedure `proc fib` computing the Fibonacci numbers:

```
begin
  proc fib(val z,u; res v) is
    if z<3 then
      (v:=u+1; r:=r+1)
    else (
      call fib (z-1,u,v);
      call fib (z-2,v,v)
      )
  end;
r:=0;
call fib(x,0,y)
end
```

The Flow Graph of Procedure proc fib



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 ';'.

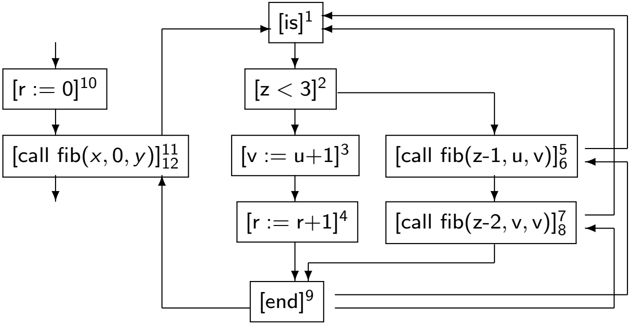
Notions and Notations for Flow Graphs (2)

...for (whole) programs:

	$P_\star$
$\text{init}_\star$	$\text{init}(S_\star)$
$\text{final}_\star$	$\text{final}(S_\star)$
$\text{blocks}_\star$	$\bigcup\{\text{blocks}\ (p) \mid \text{proc}\ p(\text{val}\ x;\ \text{res}\ y)\ \text{is}^{\ell_n}\ S\ \text{end}^{\ell_x}\ \textit{is in}\ D_\star\} \cup\ \text{blocks}(S_\star)$
$\text{labels}_\star$	$\bigcup\{\text{labels}\ (p) \mid \text{proc}\ p(\text{val}\ x;\ \text{res}\ y)\ \text{is}^{\ell_n}\ S\ \text{end}^{\ell_x}\ \textit{is in}\ D_\star\} \cup\ \text{labels}(S_\star)$
$\text{flow}_\star$	$\bigcup\{\text{flow}\ (p) \mid \text{proc}\ p(\text{val}\ x;\ \text{res}\ y)\ \text{is}^{\ell_n}\ S\ \text{end}^{\ell_x}\ \textit{is in}\ D_\star\} \cup\ \text{flow}(S_\star)$
$\mathbf{Lab}_\star$	$\text{labels}_\star$

$$\begin{aligned} \textit{inter-flow}_\star \quad = \quad & \{(\ell_c, \ell_n, \ell_x, \ell_r) \mid P_\star\ \textit{contains}\ [\text{call}\ p(a,z)]^{\ell_c}_{\ell_r}\ \textit{as well as} \\ & \text{proc}\ p(\text{val}\ x;\ \text{res}\ y)\ \text{is}^{\ell_n}\ S\ \text{end}^{\ell_x}\} \end{aligned}$$

Illustrating Example



$$
\begin{array}{rcl}
\mathit{flow}_\star & = & \{(1,2),(2,3),(3,4),(4,9),\\
& & (2,5),(5;1),(9;6),(6,7),(7;1),(9;8),(8,9),\\
& & (11;1),(9;12),(10,11)\}\\
\mathit{inter\text{-}flow}_\star & = & \{(11,1,9,12),(5,1,9,6),(7,1,9,8)\}
\end{array}
$$

Metavariables for Forward/Backward Analyses

Forward Analyses:

- $F = \mathit{flow}_\star$
- $E = \mathit{init}_\star$
- $IF = \mathit{inter\text{-}flow}_\star$

Backward Analyses:

- $F = \mathit{flow}_\star^R$
- $E = \mathit{final}_\star$
- $IF = \mathit{inter\text{-}flow}_\star^R$

Towards Interprocedural DFA

New transfer functions dealing with interprocedural flow are required:

- For each procedure call $[\text{call } p(a,z)]^{\ell_c}_{\ell_r}$ 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.

Chapter 13.1.1

Naive Interprocedural DFA

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

$$\iota_E^\ell =_{df} \begin{cases} \iota & \text{if } l \in E \\ \perp & \text{if } l \notin E \end{cases}$$

Interprocedural DFA: Naive Formulation (2)

Given the previous assumptions we have:

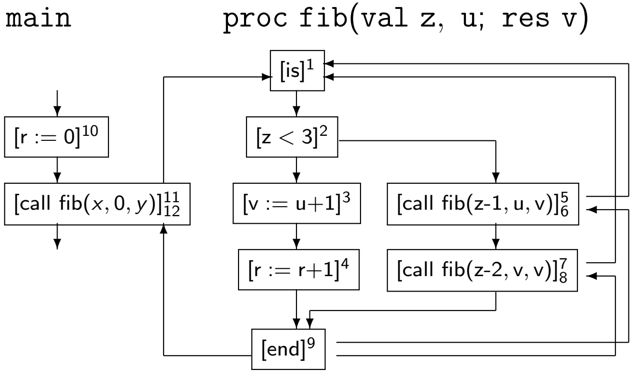
- Both procedure calls $(\ell_c; \ell_n)$ and procedure returns $(\ell_x; \ell_r)$ are treated like "goto's".
- There is no mechanism for ensuring that information flowing along $(\ell_c; \ell_n)$ flows back along $(\ell_x; \ell_r)$ to the *same* call
- Intuitively, the equation system considers a much too large set of "paths" through the program and hence will be grossly imprecise (although formally on the safe side)

Chapter 13.1.2

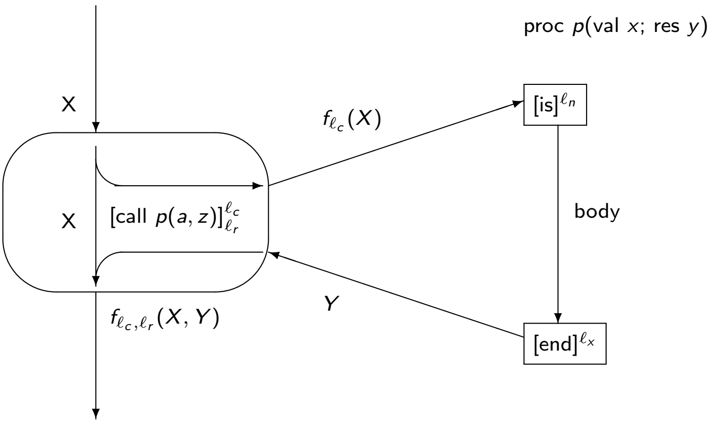
Interprocedurally Valid and Complete Paths

Interprocedurally Valid Program Paths

We want to overcome the shortcoming of the naive formulation by restricting attention to paths that have the proper nesting of procedure calls and exits. Important are the notions of matching procedure entries and exits and of complete and valid paths.



Matching Procedure Entries and Exits



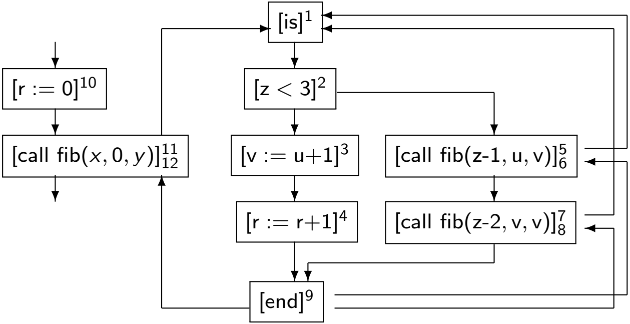
Complete Paths

A complete path from ℓ_1 to ℓ_2 in $P_\star$ has proper nesting of procedure entries and exits; and a procedure returns to the point where it was called:

$$
\begin{array}{ll}
CP_{\ell_1,\ell_2} \longrightarrow \ell_1 & \text{whenever } \ell_1 = \ell_2 \\
CP_{\ell_1,\ell_3} \longrightarrow \ell_1, CP_{\ell_2,\ell_3} & \text{whenever } (\ell_1,\ell_2) \in \text{flow}_\star \\
CP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, CP_{\ell_r,\ell} & \text{whenever } (\ell_c,\ell_n,\ell_x,\ell_r) \in \text{inter-flow}_\star
\end{array}
$$

Recall:
$(\ell_c,\ell_n,\ell_r,\ell_x) \in$ inter-flow$_\star$, if $P_\star$ contains $[\text{call } p(a,z)]^{\ell_c}_{\ell_r}$ as well as proc $p(\text{val } x;\ \text{res } y)$ is$^{\ell_n}$ S end$^{\ell_x}$.

Illustrating Example: Complete Paths



$$
\begin{aligned}
CP_{10,12} &\to 10,\ CP_{11,12}\\
CP_{11,12} &\to 11,\ CP_{1,9},\ CP_{12,12}\\
CP_{1,9} &\to 1,\ CP_{2,9}\\
CP_{2,9} &\to 2,\ CP_{3,9}\\
CP_{2,9} &\to 2,\ CP_{5,9}
\end{aligned}
$$

$$
\begin{aligned}
CP_{3,9} &\to 3,\ CP_{4,9}\\
CP_{4,9} &\to 4,\ CP_{9,9}\\
CP_{5,9} &\to 5,\ CP_{1,9},\ CP_{6,9}\\
CP_{6,9} &\to 6,\ CP_{7,9}\\
CP_{7,9} &\to 7,\ CP_{1,9},\ CP_{8,9}\\
CP_{8,9} &\to 8,\ CP_{9,9}
\end{aligned}
$$

$$
\begin{aligned}
CP_{12,12} &\to 12\\
CP_{9,9} &\to 9
\end{aligned}
$$

Chapter 13.2

MVP Approach and *MVP* Solution

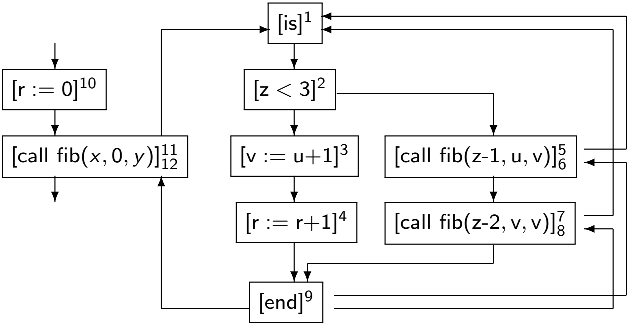
Valid Paths

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:

$$
\begin{array}{ll}
VP_\star \longrightarrow VP_{\mathsf{init}_\star,\ell} & \text{whenever } \ell \in \mathsf{Lab}_\star \\
VP_{\ell_1,\ell_2} \longrightarrow \ell_1 & \text{whenever } \ell_1 = \ell_2 \\
VP_{\ell_1,\ell_3} \longrightarrow \ell_1, VP_{\ell_2,\ell_3} & \text{whenever } (\ell_1,\ell_2) \in \mathsf{flow}_\star \\
VP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, VP_{\ell_r,\ell} & \text{whenever } (\ell_c,\ell_n,\ell_x,\ell_r) \in \mathsf{inter\text{-}flow}_\star \\
VP_{\ell_c,\ell} \longrightarrow \ell_c, VP_{\ell_n,\ell} & \text{whenever } (\ell_c,\ell_n,\ell_x,\ell_r) \in \mathsf{inter\text{-}flow}_\star
\end{array}
$$

Note: The valid paths are generated by the non-terminal $VP_\star$.

Illustrating Example: Valid Paths



$CP_{10,12} \rightarrow 10,\ CP_{11,12}$

$CP_{11,12} \rightarrow 11,\ CP_{1,9},\ CP_{12,12}$

$CP_{1,9} \rightarrow 1,\ CP_{2,9}$

$CP_{2,9} \rightarrow 2,\ CP_{3,9}$

$CP_{2,9} \rightarrow 2,\ CP_{5,9}$

$CP_{3,9} \rightarrow 3,\ CP_{4,9}$

$CP_{4,9} \rightarrow 4,\ CP_{9,9}$

$CP_{5,9} \rightarrow 5,\ CP_{1,9},\ CP_{6,9}$

$CP_{6,9} \rightarrow 6,\ CP_{7,9}$

$CP_{7,9} \rightarrow 7,\ CP_{1,9},\ CP_{8,9}$

$CP_{8,9} \rightarrow 8,\ CP_{9,9}$

$CP_{12,12} \rightarrow 12$

$CP_{9,9} \rightarrow 9$

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]

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

$vpath_\circ(\ell) =$
$\{[\ell_1, \ldots, \ell_{n-1}] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1, \ldots, \ell_n]$ is valid path$\}$

$vpath_\bullet(\ell) =$
$\{[\ell_1, \ldots, \ell_n] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1, \ldots, \ell_n]$ is valid path$\}$

Discussing the MVP Solution (1)

The MVP solution may be undecidable (even) for lattices satisfying the descending chain condition, just as was the case for the MOP solution.

Therefore, we need to reconsider the maximal fixed point approach and adapt it to

- avoid considering too many paths
- taking call context information into account.

Discussing the MVP Solution (2)

In more detail:

We have to

- reconsider the MFP solution and avoid taking too many invalid paths into account.

An obvious approach is to

- encode information about the paths taken into the data flow properties themselves.

This can be achieved by

- introducing context information $\delta \in \Delta$.

Chapter 13.3

Call Strings, Assumption Sets

Towards the MFP Counterpart of MVP

- Context insensitive analysis: No context information is used.
- Context sensitive analysis: Context information $\delta \in \Delta$ is used.
  - Call strings:
    - An abstraction of the sequences of procedure calls that have been performed so far.
    - Example: The program point where the call was initiated.
  - Assumption sets:
    - An abstraction of the states in which previous calls have been performed.
    - Example: An abstraction of the actual parameters of the call.

Chapter 13.3.1

Call Strings

Call Strings δ as Context Information Δ

- Encode the path taken.
- Only record flows of the form $(\ell_c; \ell_n)$ corresponding to a procedure call.
- we take as context $\Delta = \text{Lab}_*^*$ where the most recent label ℓ_c of a procedure call is at the right end.
- Elements of Δ are called call strings.
- The sequence of labels $\ell_c^1, \ell_c^2, \ldots, \ell_c^m$ 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:

$$\Lambda, [11], [11, 5], [11, 7], [11, 5, 5], [11, 5, 7], [11, 7, 5], [11, 7, 7], \ldots$$

The Adapted MFP Equation System

The Adapted MFP-Equation System:

$$\begin{array}{rcl} A_\circ(\ell) & = & \bigsqcap\{A_\bullet(\ell') \mid (\ell',\ell) \in F \vee (\ell';\ell) \in F\} \sqcap \widehat{\iota_E^\ell} \\ A_\bullet(\ell) & = & \widehat{f_\ell^A}(A_\circ(\ell)) \end{array}$$

where

- $\hat{L} = \Delta \rightarrow\ L$ maps a context to a data flow property (i.e., a data flow lattice element)
- each transfer function $\widehat{f_l}$ 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)
- $$\widehat{\iota_E^\ell} =_{df} \begin{cases} \iota_E^\ell & \text{if } \delta = \Lambda \\ \perp & \text{otherwise} \end{cases}$$

Making it Practical: Bounding Call Strings

Problem: Call strings can be arbitrarily long (recursive calls)

Solution: Truncate the call strings to have length of at most k for some fixed number k

In practice:

- $\Delta = \mathsf{Lab}_*^{\leq k}$, 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
  - $\Lambda, [11], [11, 5], [11, 7], [5, 5], [5, 7], [7, 5], [7, 7]$
- ...

Chapter 13.3.2

Assumption Sets

Assumption Sets δ as Context Information Δ

Instead of describing a path directly in terms of the calls being performed (as a call string does), information about the state in which a call was made can be stored (as an assumption set does).

For a more detailed account of the assumption set approach refer to

- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Chapter 2.5.5, Assumption Sets as Context)

Chapter 13.3.3

Advanced Topics

Advanced Topics

... of interprocedural program analysis and a glimpse on how they can be addressed by static program analysis.

- Function pointers
- Virtual function calls
- Overloaded functions

Function Pointers

Values of function pointer variables

The value of a function pointer variable is the address of a function. At run-time different values can be assigned to pointer variables.

Interprocedural Control Flow

Any function with the same signature (=parameter types) can be potentially called by using a function pointer.

Program analysis can reduce the number of functions that may be called at run-time by computing the set of possible pointer values assigned to function pointer variables in a given program.

Virtual Function Calls & Overloaded Functions

...in object-oriented programming.

Virtual function calls

By taking the class hierarchy into account, we can limit the methods that can be called to the set of overriding methods of subclasses. Program analysis can further reduce the number of methods that may be called at run-time.

Overloaded functions

Calls to overloaded functions are resolved at compile time.

Chapter 13.4

The Cloning-based Approach

Cloning-based Approaches

Especially popular

...for object-oriented and points-to analyses.

Key idea

...distinguishing contexts via cloning.

Applications

- *k*-object sensitive analysis for object-oriented programs (e.g., [MRR'02,SBL'11]).
- Pointer analyses (e.g., [BLQ'03,WL'04,ZC'04,Wha'07, BS'09])
  - Cloning-based pointer analyses are often expressed in Datalog solved using specialized Datalog solvers exploiting redundancy arising from large numbers of similar contexts for high *k* values ([Wha'07,BS'09]).
  - Contexts are typically represented by binary decision diagrams (BDDs) ([BLQHU'03,WL'04,ZC'04]) or explicit representations from the database literature ([BS'09]).
  - Recursion is typically approximated in an ad hoc manner. Exceptions are the approaches of [KK'08,KMR'12].

Chapter 13.5

References, Further Reading

Further Reading for Chapter 13 (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 12.1.3, Call Strings)

- Uday P. Khedker, Bageshri Karkare. *Efficiency, Precision, Simplicity, and Generality in Interprocedural Dataflow Analysis: Resurrecting the Classical Call Strings Method*. In Proceedings of the 17th International Conference on Compiler Construction (CC 2008), Springer-V., LNCS 4959, 213-228, 2008.

Further Reading for Chapter 13 (2)

- Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. (Chapter 7, Introduction to Interprocedural Data Flow Analysis, Chapter 8, Functional Approach to Interprocedural Data Flow Analysis; Chapter 9, Value-Based Approach to Interprocedural Data Flow Analysis)

- Ravi Mangal, Mayur Naik, Hongseok Yang. *A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join*. In Proceedings of the 23rd European Symposium on Programming (ESOP 2014), Springer-V., LNCS 8410, 513-533, 2014.

Further Reading for Chapter 13 (3)

- Matthew Might, Yannis Smaragdakis, David Van Horn. *Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. OO Program Analysis*. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), ACM SIGPLAN Notices 45(6):305-315, 2010.
- 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*. Springer-V., 2nd edition, 2005. (Chapter 2.5, Interprocedural Analysis)

References for Chapter 13 (4)

Yannis Smaragdakis, Martin Bravenboer, Ondřej Lhoták. *Pick Your Contexts Well: Understanding Object-Sensitivity*. In Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), 17-30, 2011.

Selected References for Chapter 13.4

Marc Berndl, Ondřej Lhoták, Feng Qian. Laurie Hendren, Navindra Umanee. *Points-to Analysis using BDDs*. In Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003), ACM SIGPLAN Notices 38:103-114, 2003.

References for Chapter 13 (5)

- Martin Bravenboer, Yannis Smaragdakis. *Strictly Declarative Specification of Sophisticated Points-to Analyses*. In Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA 2009), 243-262, 2009.
- Uday P. Khedker, Bageshri Karkare. *Efficiency, Precision, Simplicity, and Generality in Interprocedural Dataflow Analysis: Resurrecting the Classical Call Strings Method*. In Proceedings of the 17th International Conference on Compiler Construction (CC 2008), Springer-V., LNCS 4959, 213-228, 2008.

References for Chapter 13 (6)

- Uday P. Khedker, Alan Mycroft, Prashant Singh Rawat. *Liveness-based Pointer Analysis*. In Proceedings of the 19th Static Analysis Symposium (SAS 2012), Springer-V., LNCS 7460, 265-282, 2012.
- Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to and Side-effect Analyses for JAVA*. In Proceedings of the 6th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), 1-11, 2002.
- Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to Analysis for JAVA*. ACM Transactions on Software Engineering and Methodology 14(4):431-477, 2005.

References for Chapter 13 (7)

- Olin Shivers. *Control-Flow Analysis in Scheme*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88), ACM SIGPLAN Notices 23:164-174, 1988.
- Yannis Smaragdakis, Martin Bravenboer, Ondřej Lhoták. *Pick Your Contexts Well: Understanding Object-Sensitivity*. In Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), 17-30, 2011.
- John Whaley. *Context-sensitive Pointer Analysis using Binary Decision Diagrams*. PhD Thesis, Stanford University, CA, USA, 2007.

References for Chapter 13 (8)

- John Whaley, Monica S. Lam. *Cloning-based Context-sensitive Pointer Alias Analysis using Binary Decision Diagrams*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:131-144, 2004.
- Jianwen Zhu, Silvian Calman. *Symbolic Pointer Analysis Revisited*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:145-157, 2004.

Part IV

Extensions, Other Settings

Chapter 14

Aliasing

Pointer/Alias/Shape Analysis (1)

Problem

- Ambiguous memory references interfere with an optimizer's ability to improve code.
- One major source of ambiguity is the use of pointer-based values.

Goal of Pointer/Alias/Shape Analysis

- determine for each pointer the set of memory locations to which it may refer.

Pointer/Alias/Shape Analysis (2)

Without such analysis the compiler must assume that each pointer can refer to any addressable value, including

- any space allocated in the run-time heap.
- any variable whose address is explicitly taken.
- any variable passed as a call-by-reference parameter.

Forms of Pointer Analysis

- points-to sets
- alias pairs
- shape analysis

Chapter 14.1

Sources of Aliasing

Aliasing Everywhere: Answers

...to the question "What is an alias?" in different areas:

- A short, easy to remember name created for use in place of a longer, more complicated name; commonly used in e-mail applications. Also referred to as a "nickname".
- A hostname that replaces another hostname, such as an alias which is another name for the same Internet address. For example, www.company.com could be an alias for server03.company.com.
- A feature of UNIX shells that enables users to define program names (and parameters) and commands with abbreviations. (e.g. alias ls 'ls -l')
- In MGI (Mouse Genome Informatics), an alternative symbol or name for part of the sequence of a known gene that resembles names for other anonymous DNA segments. For example, D6Mit236 is an alias for Cftr.

Aliasing in Programs

In programs aliasing occurs when there exists more than one access path to a storage location.

An access path is the l-value of an expression that is constructed from variables, pointer dereference operators, and structure field operation operators.

Java (References)

```
A a,b;
a = new A();
b = a;
b.val = 0;
```

C++ (References)

```
A& a = *new A();
A& b = a;
b.val = 0;
```

C++ (Pointers)

```
A* a; A* b;
a = new A();
b = a;
b->val = 0;
```

C (Pointers)

```
A *a, *b;
a = (A*)malloc(sizeof(A));
b = a;
b->val = 0;
```

Examples of Different Forms of Aliasing (1)

Fortran 77

EQUIVALENCE statement can be used to specify that two or more scalar variables, array variables, and/or contiguous portions of array variables begin at the same storage location.

Pascal, Modula 2/3, Java

- Variable of a reference type is restricted to have either the value nil/null or to refer to objects of a particular specified type.
- An object may be accessible through several references at once, but it cannot both have its own variable name *and* be accessible through a pointer.

Examples of Different Forms of Aliasing (2)

C/C++

- The union type specifier allows to create static aliases. A union type may have several fields declared, all of which overlap in (= share) storage.
- It is legal to compute the address of an object with the & operator (statically, automatically, or dynamically allocated).
- Allows arithmetic on pointers and considers it equivalent to array indexing

Chapter 14.2

Relevance of Aliasing for Program Optimization

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.

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 $|Loc| \leq 1$.

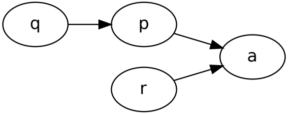
Representation of Alias Information

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
14.1
14.2
14.3
14.4
Chap. 15

Chapter 14.3

Shape Analysis

Questions about Heap Contents (1)

Execution State

Let execution state mean the set of cells in the heap, the connections between them (via pointer components of heap cells) and the values of pointer variables in the store.

- NULL pointers (Question 1): Does a pointer variable or a pointer component of a heap cell contain NULL at the entry to a statement that dereferences the pointer or component?
  - Yes (for every state): Issue an error message.
  - No (for every state): Eliminate a check for NULL.
  - Maybe: Warn about the potential NULL dereference.
- Memory leak (Question 2): Does a procedure or a program leave behind unreachable heap cells when it returns?
  - Yes (in some state): Issue a warning.

Questions about Heap Contents (2)

- Aliasing (Question 3): Do two pointer expressions reference the same heap cell?
  - Yes (for every state):
    - trigger a prefetch to improve cache performance
    - predict a cache hit to improve cache behavior prediction
    - increase the sets of uses and definitions for an improved liveness analysis
  - No (for every state): Disambiguate memory references and improve program dependence information.
- Sharing (Question 4): Is a heap cell shared? (within the heap)
  - Yes (for some state): Warn about explicit deallocation, because the memory manager may run into an inconsistent state.
  - No (for every state): Explicitly deallocate the heap cell when the last pointer to ceases to exist.

Questions about Heap Contents (3)

- Reachability (Question 5): Is a heap cell reachable from a specific variable or from any pointer variable?
  - Yes (for every state): Use this information for program verification.
  - No (for every state): Insert code at compile time that collects unreachable cells at run-time.
- Disjointness (Question 6): Do two data structures pointed to by two distinct pointer variables ever have common elements?
  - No (for every state): Distribute disjoint data structures and their computations to different processors.
- Cyclicity (Question 7): Is a heap cell part of a cycle?
  - No (for every state): Perform garbage collection of data structures by reference counting. Process all elements in an acyclic linked list in a doall-parallel fashion.

Shape Analysis

Aim of Shape Analysis (SA)

The aim of shape analysis is to determine a finite representation of heap allocated data structures which can grow arbitrarily large.

SA can determine the possible shapes data structures may take such as:

- lists, trees
- directed acyclic graphs, arbitrary graphs
- properties such as whether a data structure is or may be cyclic.

As example we shall discuss a precise shape analysis (from Nielson/Nielson/Hankin, PoPA, Chap. 2.6) that performs strong update and uses shape graphs to represent heap allocated data structures. It emphasises the analysis of list like data structures.

Strong Update

Here "strong" means that an update or nullification of a pointer expression allows one to remove (kill) the existing binding before adding a new one (gen).

We shall study a powerful analysis that achieves

- Strong nullification
- Strong update

for destructive updates that destroy (overwrite) existing values in pointer variables and in heap allocated data structures in general.

Examples:

- $[x := nil]^\ell$
- $[x.sel_1 := y.sel_2]^\ell$

Extending the WHILE Language

We extend the WHILE-language syntax with constructs that allow to create cells in the heap.

- the cells are structured and may contain values as well as pointers to other cells.
- the data stored in cells is accessed via selectors; we assume that a finite and non-empty set Sel of selector names is given:

$$sel \in \mathsf{Sel} \qquad \text{selector names}$$

- 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.

Abstract Syntax of Pointer Language

The syntax of the WHILE-language is extended to have:

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

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.

Shape Graphs

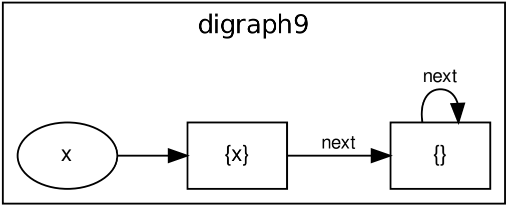
We shall introduce a method for combining the locations of the semantics into a finite number of abstract locations.

The analysis operates on shape graphs $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ consisting of:

- an abstract state, S (mapping variables to abstract locations).
- an abstract heap, H (specifying links between abstract locations).
- sharing information, is, for the abstract locations.

The last component allows us to recover some of the imprecision introduced by combining many locations into one abstract location.

Example



$$
g_9 = (\mathsf{S}, \mathsf{H}, \mathsf{is}) \text{ where}
$$

$$
\begin{aligned}
\mathsf{S} &= \{(\mathtt{x}, n_{\{\mathtt{x}\}})\} \\
\mathsf{H} &= \{(n_{\{\mathtt{x}\}}, \mathtt{next}, n_{\emptyset}), (n_{\emptyset}, \mathtt{next}, n_{\emptyset})\} \\
\mathsf{is} &= \emptyset
\end{aligned}
$$

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 locations of ALoc.

The abstract location $n_\emptyset$ is called the abstract summary location and represents all the locations that cannot be reached directly from the state without consulting the heap.

Clearly, n_X and $n_\emptyset$ represent disjoint sets of locations when $X \neq \emptyset$.

Invariant 1: If two abstract locations n_X and n_Y occur in the same shape graph then either $X = Y$ or $X \cap Y = \emptyset$. (i.e., two distinct abstract locations n_X and n_Y always represent disjoint sets of locations)

Abstract State

The abstract state, S, maps variables to abstract locations.

To maintain the naming convention for abstract locations we shall ensure that:

Invariant 2: If x is mapped to n_X by the abstract state then $x \in X$.

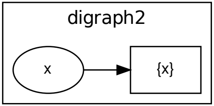
From Invariant 1 it follows that there will be at most one abstract location in the (same) shape graph containing a given variable.

We shall only be interested in the shape of heap so we shall not distinguish between integer values, nil-pointers, and uninitialized fields; hence we can view the abstract state as an element of

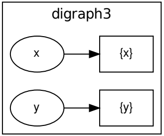
$$\mathsf{S} \in \mathsf{AState} = \mathcal{P}\mathsf{Var}_\star \times \mathsf{ALoc}$$

Example: Creating Linked Data Structures

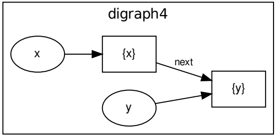
$[new(x)]^2$



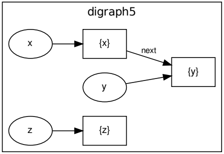
$[new(y)]^3$



$[\text{x.next} := \text{y}]^4$



$[new(z)]^5$



Abstract Heap

The abstract heap, H, specifies the links between the abstract locations.

The links will be specified by triples (n_V, sel, n_W) and formally we take the abstract heap as an element of

$$\mathsf{H} \in \mathsf{AHeap} = \mathcal{P}\mathsf{ALoc} \times \mathsf{Sel} \times \mathsf{ALoc}$$

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 $n_\emptyset$.

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:*

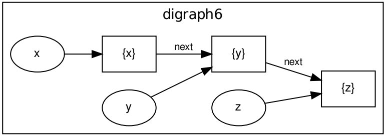
- an abstract location n_X will be included in is if it represents a location that is the target of more than one pointer in the heap.

In the case of the abstract summary location, $n_\emptyset$, the explicit sharing information clearly gives extra information:

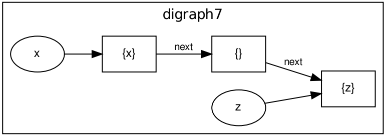
- if $n_\emptyset \in$ is then there might be a location represented by $n_\emptyset$ that is the target of two or more heap pointers.
- if $n_\emptyset \notin$ is then all the locations of represented by $n_\emptyset$ will be the target of at most one heap pointer.

Maintaining Sharing Information (1)

$[y.next := z]^6$

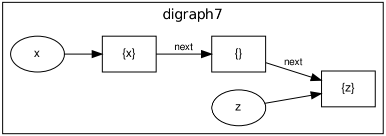


$[y := null]^7$

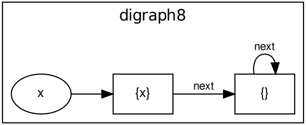


Maintaining Sharing Information (2)

$[y := \text{null}]^7$



$[z := \text{null}]^8$



Sharing Information Invariants (1)

We shall impose two invariants to ensure that information in the sharing component is also reflected in the abstract heap, and vice versa.

The first invariant, Invariant 4, ensures that information in the sharing component is also reflected in the abstract heap:

Invariant 4: If $n_X \in$ is then either

a) $(n_\emptyset, sel, n_X)$ is in the abstract heap for some sel, or

b) there exist two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap (that is either $sel_1 \neq sel_2$ or $V \neq W$).

- Case 4a) means that there might be several locations represented by $n_\emptyset$ that point to n_X
- Case 4b) means that two distinct pointers (with different source or different selectors) point to n_X.

Sharing Information Invariants (2)

The second invariant, Invariant 5, ensures that sharing information present in the abstract heap is also reflected in the sharing component:

Invariant 5: Whenever there are two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap and $n_X \neq n_\emptyset$ then $n_X \in$ is.

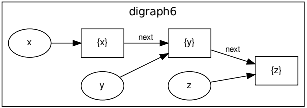
This invariant takes care of the situation where n_X represents a single location being the target of two or more heap pointers.

Note that Invariant 5 is the "inverse" of Invariant 4(b).

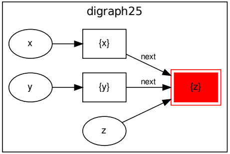
We have no "inverse" of Invariant 4(a) - the presence of a pointer from $n_\emptyset$ to n_X gives no information about sharing properties of n_X that are represented in is.

Sharing Component: Example 1

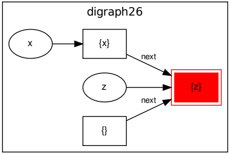
$[y.next := z]^6$



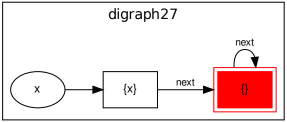
$[x.next := z]^{7'}$



$[y := null]^{8'}$

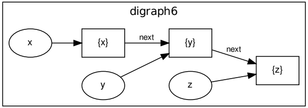


$[z := null]^{9'}$

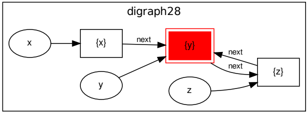


Sharing Component: Example 2

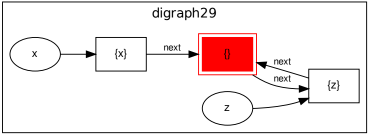
$[y.next := z]^6$



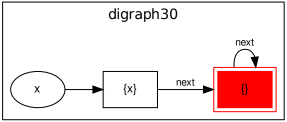
$[z.next := y]^{7''}$



$[y := null]^{8''}$



$[z := null]^{9''}$



Compatible Shape Graphs

A shape graph is a triple $(\mathsf{S}, \mathsf{H}, \mathsf{is})$:

$$\begin{aligned} \mathsf{S} \in \mathsf{AState} &= \mathcal{P}(\mathsf{Var}_\star \times \mathsf{ALoc}) \\ \mathsf{H} \in \mathsf{AHeap} &= \mathcal{P}(\mathsf{ALoc} \times \mathsf{Sel} \times \mathsf{ALoc}) \\ \mathsf{is} \in \mathsf{IsShared} &= \mathcal{P}(\mathsf{ALoc}) \end{aligned}$$

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 by

$$\mathsf{SG} = \{(\mathsf{S}, \mathsf{H}, \mathsf{is}) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \text{ is compatible}\}$$

Complete Lattice of Shape Graphs

The analysis, to be called *Shape*, will operate over *sets* of compatible shape graphs, i.e. elements of $\mathcal{P}$SG.

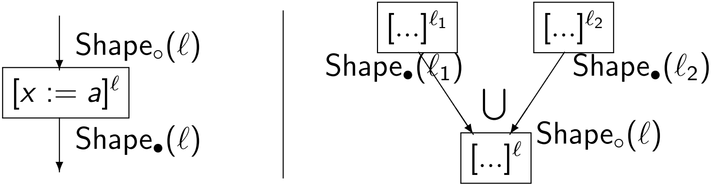
Since $\mathcal{P}$SG is a power set, it is trivially a complete lattice with

- ordering relation $\sqsubseteq$ being $\subseteq$
- combination operator $\sqcap$ being $\cup$ (may analysis)

$\mathcal{P}$SG is finite because SG $\subseteq$ AState $\times$ AHeap $\times$ IsShared and all of AState, AHeap, IsShared are finite.

The analysis will be specified as an instance of a Monotone Framework with the complete lattice of properties being $\mathcal{P}$SG, and as a forward analysis.

Analysis



$$
\begin{array}{rcl}
\text{Shape}_\circ(\ell) & = & \left\{ \begin{array}{ll} \iota & : \quad \text{if } \ell = \text{init}(S_\star) \\ \bigcup\{\text{Shape}_\bullet(\ell') \mid (\ell', \ell) \in \text{flow}(S_\star)\} & : \quad \text{otherwise} \end{array} \right. \\
\text{Shape}_\bullet(\ell) & = & f_\ell^{SA}(\text{Shape}_\circ(\ell))
\end{array}
$$

where $\iota \in \mathcal{P}SG$ is the extremal value holding at entry to $S_\star$.

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 $\mathsf{Shape}_\bullet(\ell)$).

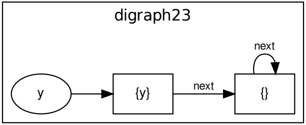
The functions ϕ_ℓ^{SA} for the statements (illustrated by example)

x := a	x := y	x := y.sel
x.sel := a	x.sel := y	x.sel := y.sel

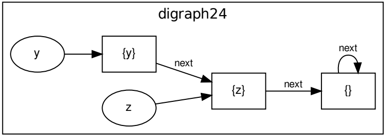
transform a shape graph into a set of different shape graphs.

The transfer functions for other statements and expressions are specified by the identity function.

Example: Materialization



$[z := y.next]^7$

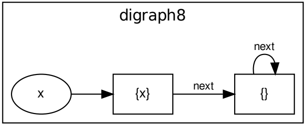


Example: Reverse List

```
[y := null]¹;
while [not isnull(x)]² do
  [t := y]³;
  [y := x]⁴;
  [x := x.next]⁵;
  [y.next := t]⁶;
od
[t := null]⁷
```

The program reverses the list pointed to by x and leaves the result in y.

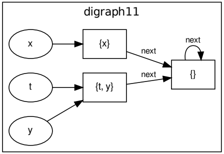
Reverse List: Extremal Value



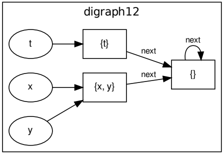
The extremal value ι is a set of graphs. The above graph is an element of this set for our example analysis of the list reversal program.

Shape Graphs in Shape$_\bullet(\ell)$ (1)

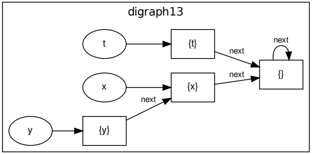
$[\mathrm{t} := \mathrm{y}]^3$



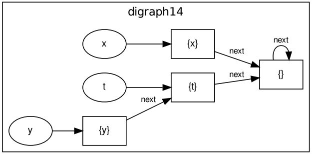
$[\mathrm{y} := \mathrm{x}]^4$



$[\mathrm{x} := \mathrm{x.next}]^5$

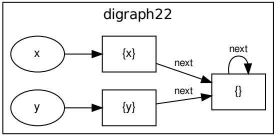


$[\mathrm{y.next} := \mathrm{t}]^6$

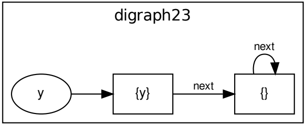


Shape Graphs in Shape$_\bullet(\ell)$ (2)

$[t := \text{null}]^7$



$[x := \text{null}]^7$



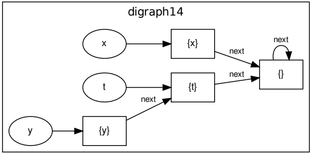
Reverse List: Established Properties

For the list reversal program shape analysis can detect that at the beginning of each iteration of the loop the following properties hold:

Invariant 1: Variable x points to an unshared, acyclic, singly linked list.

Invariant 2: Variable y points to an unshared, acyclic, singly linked list, and variable t may point to the second element of the y-list (if such an element exists).

Invariant 3: The lists pointed to by x and y are disjoint.



Drawbacks and Improvements

An improved version, on which the discussed analysis is based on, can be found in [SRW'98]:

- Operates on a single shape graph instead of sets of shape graphs.
- Merges sets of compatible shape graphs in one summary shape graph.
- Uses various mechanisms for extracting parts of individual compatible shape graphs.
- Avoids the exponential factor in the cost of the discussed analysis.

The sharing component of the shape graphs is designed to detect list-like properties:

- It can be replaced by other components detecting other shape properties [SRW'02; Compiler Design Handbook, Chap. 5].

Chapter 14.4

References, Further Reading

Further Reading for Chapter 14 (1)

- David Chase, Mark N. Wegmann, F. Kenneth Zadeck. *Analysis of Pointers and Structures*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), ACM SIGPLAN Notices 25:296-310, 1990.
- Maryam Emami, Rakesh Ghiya, Laurie J. Hendren. *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):242-256, 1994.

Further Reading for Chapter 14 (2)

- Ben Hardekopf, Calvin Lin. *Semi-sparse Flow-sensitive Pointer Analysis*. In Conference Record of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009), 226-238, 2009.
- Rakesh Ghiya, Laurie J. Hendren. *Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 1-15, 1996.
- Bertrand Jeannet, Alexey Logivon, Thomas W. Reps, Mooly Sagiv. *A Relational Approach to Interprocedural Shape Analysis*. In Proceedings of the 11th Static Analysis Symposium (SAS 2004), Springer-V., LNCS 3248, 246-264, 2004.

Further Reading for Chapter 14 (3)

- Roman Manevich, Mooly Sagiv, Ganesan Ramalingam, John Field. *Partially Disjunctive Heap Abstraction*. In Proceedings of the 11th Static Analysis Symposium (SAS 2004), Springer-V., LNCS 3248, 265-279, 2004.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 10, Alias Analysis)
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 2.6, Shape Analysis)

Further Reading for Chapter 14 (4)

- Viktor Pavlu, Markus Schordan, Andreas Krall. *Computation of Alias Sets from Shape Graphs for Comparison of Shape Analysis Precision*. In Proceedings of the 11th International IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2011), 25-34, 2011. [Best Paper Award SCAM 2011]
- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-analysis Problems in Languages with Destructive Updating*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 16-31, 1996.

Further Reading for Chapter 14 (5)

- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-analysis Problems in Languages with Destructive Updating*. ACM Transactions on Programming Languages and Systems 20(1):1-50, 1998.
- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. In Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), 105-118, 1999.
- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-valued Logic*. ACM Transactions on Programming Languages and Systems 24(3):217-298, 2002.

Further Reading for Chapter 14 (6)

- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations & Machine Code Generation*. CRC Press, 2002. (Chapter 5, Shape Analysis and Application)
- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2nd edition, 2008. (Chapter 5, Shape Analysis and Application)
- Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn. *Scalable Shape Analysis for Systems Code*. In Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), Springer-V., LNCS 5123, 385-398, 2008.

Chapter 15

Optimizations for Object-Oriented Languages

Optimizations f. Object-Oriented Languages (1)

...related to method invocation.

Invoking a method in an object-oriented language requires looking up the address of the block of code which implements that method and passing control to it.

Opportunities for optimization

- Look-up may be performed at compile time.
- There is only one implementation of the method in the class and in its subclasses.
- Language provides a declaration which forces the call to be non-virtual.
- Compiler performs static analysis which can determine that a unique implementation is always called at a particular call-site.

Optimizations f. Object-Oriented Languages (2)

Related optimizations for exploiting these opportunities:

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

Overview

- Object Layout and Method Invocation (cf. Chapter 15.1)
  - Single inheritance
  - Multiple inheritance
- Devirtualization of Method Calls (cf. Chapter 15.2)
  - Class hierarchy analysis
  - Rapid type analysis
  - Inlining
- Escape Analysis (cf. Chapter 15.3)
  - Connection graphs
  - Intra-procedural
  - Inter-procedural

Chapter 15.1

Object Layout and Method Invocation

Object Layout and Method Invocation

The memory layout of an object and how the layout supports dynamic dispatch are crucial factors for performance.

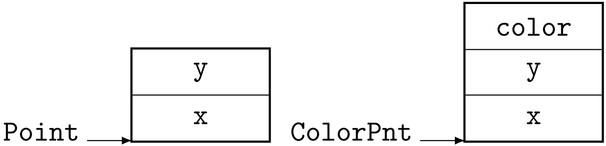
- Single Inheritance
  - with and without virtual dispatch table (i.e., direct calling guarded by a type test)
- Multiple Inheritance
  ...various techniques with different compromises
  - embedding superclasses
  - trampolines
  - table compression

Chapter 15.1.1
Single Inheritance

Single Inheritance Layout

```
class Point {
    int x, y;
}
```

```
class ColorPnt extends Point {
    int color;
}
```

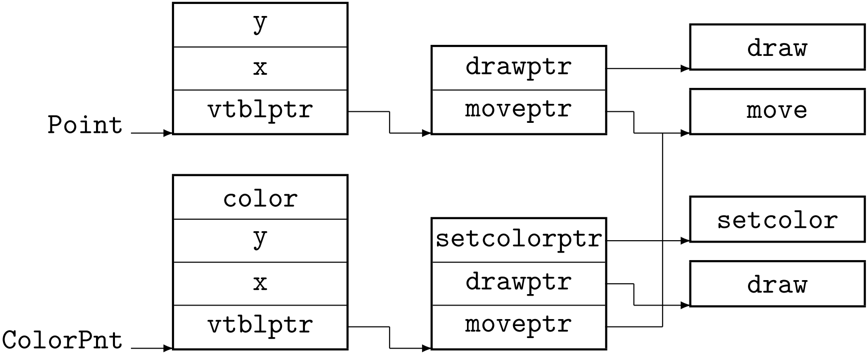


- Memory layout of an object of a superclass is a prefix of the memory layout of an object of the subclass.
- Instance variables access requires just one load or store instruction.

Single Inheritance Layout with vtbl

```
class Point {
   int x, y;
   void move(int x, int y) {...}
   void draw() {...}
}
```

```
class ColorPnt extends Point {
   int color;
   void draw() {...}
   void setcolor(int c) {...}
}
```



Invocation of Virtual Methods with vtbl

- Dynamic dispatching using a virtual method table (vtbl) has the advantage of being fast and executing in constant time.
- It is possible to add new methods and to override methods.
- Each method is assigned a fixed offset in the virtual method table (vtbl).
- Method invocation is just three machine code instructions:

```
LDQ vtblptr,(obj)          ; load vtbl pointer
LDQ mptr,method(vtblptr)   ; load method pointer
JSR (mptr)                 ; call method
```

- One extra word of memory is needed in each object for the pointer to the virtual method table (vtbl).

Dispatch Without Virtual Method Tables

Despite the use of branch target caches, indirect branches are expensive on modern architectures.

The pointer to the class information and virtual method table is replaced by a type identifier:

- A type identifier is an integer representing the type the object.
- It is used in a dispatch function which searches for the type of the receiver.
- Example: SmallEiffel (binary search).
- Dispatch functions are shared between calls with the same statically determined set of concrete types.
- In the dispatch function a direct branch to the dispatched method is used (or it is inlined).

Example

Let type identifiers T_A, T_B, T_C, and T_D be sorted by increasing number. The dispatch code for calling $x.f$ is:

if $id_x \leq T_B$ **then**
  if $id_x \leq T_A$ **then** $f_A(x)$
  else $f_B(x)$
else if $id_x \leq T_C$ **then** $f_C(x)$
  else $f_D(x)$

Comparison with dispatching using a virtual method table:

- Empirical study showed that for a method invocation with three concrete types, dispatching with binary search is between 10% and 48% faster.
- For a megamorphic call with 50 concrete types, the performance is about the same.

Chapter 15.1.2

Multiple Inheritance

Multiple Inheritance

...extending the superclasses as in single inheritance does not work anymore.

Instead

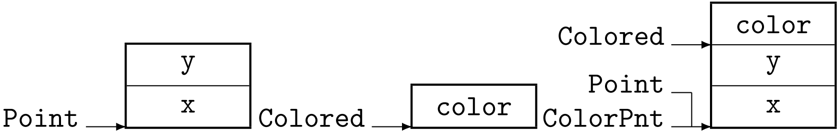
- Fields of superclass are embedded as contiguous block.
- Embedding allows fast access to instance variables exactly as in single inheritance.
- Garbage collection becomes more complex because pointers also point into the middle of objects.

Object Memory Layout (without vtbl)

```
class Point {
    int x, y;
}
```

```
class Colored {
    int color;
}
```

```
class ColorPnt extends Point, Colored {
}
```



Dynamic Dispatching for Embedding

- Allows fast access to instance variables exactly as with single inheritance.
- For every superclass
  - virtual method tables (vbtl) 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).

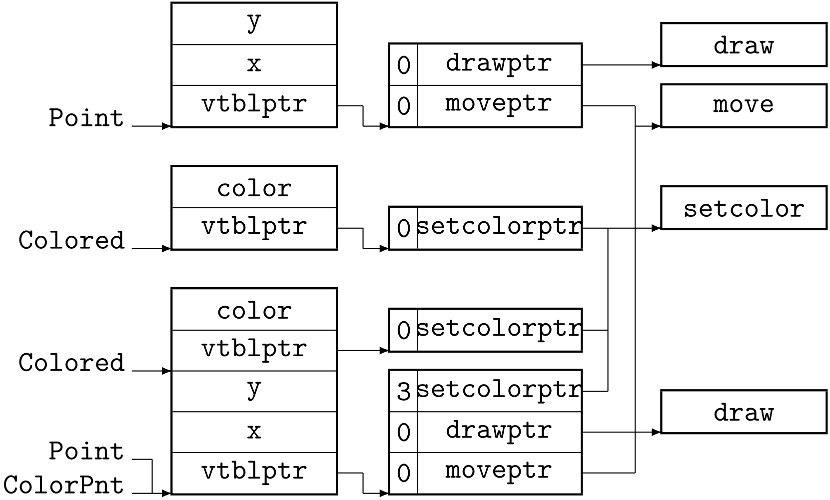
Multiple Inheritance with vtbl (1)

```
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() {...}
    }
```

Multiple Inheritance with vtbl (2)



Pointer Adjustment and Adjustment Offset

Pointer adjustment has to be suppressed for casts of null pointers:

```
Colored col; ColorPnt cp; ...;
col = cp;  // if (cp!=null)col=(Colored)((int*)cp+3)
```

Problem w/ implicit casts from actual receiver to formal receiver:

- Caller has no type info of formal receiver in the callee.
- Callee has no type info of actual receiver of the caller.
- Therefore this type info has to be stored as an adjustment offset in the vtbl.

Method Invocation with vtbl

Method invocation now takes 4 to 5 machine instructions (depending on the architecture).

```
LD  vtblptr,(obj)              ; load vtbl pointer
LD  mptr,method_ptr(vtblptr)   ; load method pointer
LD  off,method_off(vtblptr)    ; load adjustment offset
ADD obj,off,obj                ; adjust receiver
JSR (mptr)                     ; call method
```

This overhead in table space and program code is even necessary when multiple inheritance is not used (in the code).

Furthermore, adjustments to the remaining parameters and the result are not possible.

Trampoline

To eliminate much of the overhead a small piece of code, called trampolin is inserted that performs the pointer adjustments and the jumps to the original code.

The advantages are

- smaller table size (no storing of an offset)
- fast method invocation when multiple inheritance is not used
  - the same dispatch code as in single inheritance

The method pointer `setcolorptr` in the virtual method table of `Colorpoint` would (instead) point to code which adds 3 to the receiver before jumping to the code of method `setcolor`:

```
ADD obj,3,obj                   ; adjust receiver
BR  setcolor                    ; call method
```

Lookup at Compile-Time

Invoking a method requires looking up the address of the method and passing control to it.

In some cases, the lookup may be performed at compile-time:

- There is only one implementation of the method in the class and its subclasses.
- The language provides a declaration that forces the call to be non-virtual.
- The compiler has performed static analysis that can determine that a unique implementation is *always* called at a particular call site.

In other cases, a runtime lookup is required.

Dispatch Table

In principle the lookup can be implemented as indexing a two-dimensional table. A number is given to

- each method in the program
- each class in the program

The method call

```
result = obj.m(a1,a2);
```

can be implemented by the following three actions:

1. Fetch a pointer to the appropriate row of the dispatch table from the object `obj`.
2. Index the dispatch table row with the method number.
3. Transfer control to the address obtained.

Dispatch Table Compression (1)

- Virtual Tables
  - Effective method for statically typed languages.
  - Methods can be numbered compactly for each class hierarchy to leave no unused entries in each vtbl.
- Row Displacement Compression
  - Idea: combine all rows into a single very large vector.
  - It is possible to have rows overlapping as long as an entry in one row corresponds to empty entries in the other rows.
  - Greedy algorithm: place first row; for all subsequent rows: place on top and shift right if conflicts exist.
  - Unchanged: implementation of method invocation.
  - Penalty: verify class of current object at the beginning of any method that can be accessed via more than one row.

Dispatch Table Compression (2)

- Selector Coloring Compression
  - Graph coloring: two rows can be merged if no column contains different method addresses for the two classes.
  - Graph: one node per class; an edge connects two nodes if the corresponding classes provide different implementations for the same method name.
  - Coloring: each color corresponds to the index for a row in the compressed table.
  - Each object contains a reference to a possibly shared row.
  - Unchanged: implementation of method invocation code.
  - Penalty: if classes C1 and C2 share the same row and C1 implements method m whereas C2 does not, then the code for m should begin with a check that control was reached via dispatching on an object of type C1.

Chapter 15.2

Devirtualization of Method Invocations

Devirtualization

Devirtualization is a technique to reduce the overhead of virtual method invocation.

The aim of this technique is to statically determine which methods can be invoked by virtual method calls.

- If exactly one method is resolved for a method call, the method can be inlined or the virtual method call can be replaced by a static method call.

The analyses necessary for devirtualization also improve the accuracy of the call graph and the accuracy of subsequent interprocedural analyses.

Chapter 15.2.1

Class Hierarchy Analysis

Class Hierarchy Analysis

The simplest devirtualization technique is class hierarchy analysis (CHA), which determines the class hierarchy used in a program.

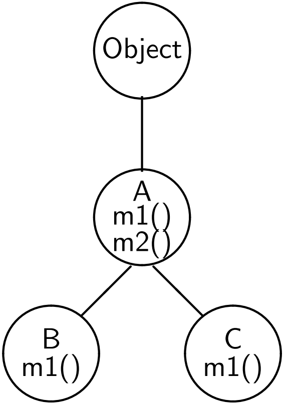
The information about all referenced classes is used to create a conservative approximation of the class hierarchy.

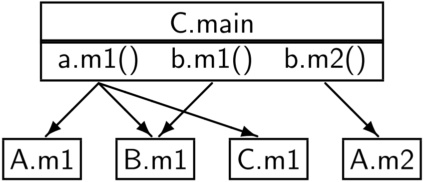
- The transitive closure of all classes referenced by the class containing the main method is computed.
- The declared types of the receiver of a virtual method call are used for determining all possible receivers.

Example: Class Hierarchy Analysis

```
class A extends Object {
    void m1() {...}
    void m2() {...}
    }
class B extends A {
    void m1() {...}
    }
class C extends A {
    void m1() {...}
    public static void main(...) {
        A a = new A();
        B b = new B();
        ...
        a.m1(); b.m1(); b.m2();
        }
    }
```

Example: Class Hierarchy and Call Graph





The CHA Algorithm

main	// the main method in a program
$x()$	// call of static method x
$type(x)$	// the declared type of the expression x
$x.y()$	// call of virtual method y in expression x
$subtype(x)$	// x and all classes which are a subtype of class x
$method(x, y)$	// the method y which is defined for class x

$callgraph := main$
$hierarchy := \{\}$
for each $m \in callgraph$ **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 $hierarchy$

Chapter 15.2.2
Rapid Type Analysis

Rapid Type Analysis (1)

Rapid type analysis (RTA) uses the fact that a method *m* of a class *c* can be invoked only if an object of type *c* is created during the execution of the program.

- RTA refines the class hierarchy (compared to CHA) by only including classes for which objects can be created at runtime.

Based on this idea

- pessimistic
- optimistic

algorithms are possible.

Rapid Type Analysis (2)

1. The pessimistic algorithm

...includes all classes in the class hierarchy for which instantiations occur in methods of the call graph from CHA.

2. The optimistic algorithm

- Initially assumes that no methods besides *main* are called and that no objects are instantiated.
- It traverses the call graph initially ignoring virtual calls (marking them in a mapping as potential calls only) following static calls only.
- When an instantiation of an object is found during analysis, all virtual methods of the corresponding objects that were left out previously are then traversed as well.
- The live part of the call graph and the set of instantiated classes grow interleaved as the algorithm proceeds.

Chapter 15.2.3
Inlining

Using Devirtualization Information

Inlining is an important usage of devirtualization information.

If a virtual method call can be devirtualized

- it might completely be replaced by inlining the call (supposed it is not recursive).

Chapter 15.3

Escape Analysis

Escape Analysis

The goal of escape analysis is to determine which objects have lifetimes which do not stretch outside the lifetime of their immediately enclosing scopes.

- The storage for such objects can be safely allocated as part of the current stack frame – that is, their storage can be allocated on the run-time stack.
- At method return, deallocation of the memory space used by non-escaping objects is automatic. No garbage collection is required.
- The transformation also improves the data locality of the program and, depending on the computer's cache, can significantly reduce execution time. Objects not escaping a thread can be allocated in the processor where that thread is scheduled.

Using Escape Information

Objects whose lifetimes are confined to within a single scope cannot be shared between two threads.

- Synchronization actions for these objects can be eliminated.

Escape Analysis by Abstract Interpretation

A prototype implementation of escape analysis was included in the IBM High Performance Compiler for Java.

The approach of Choi et al. (OOPSLA'99) attempts to determine whether the object

- escapes from a method (i.e., from the scope where it is allocated).
- escapes from the thread that created it
  - the object can escape a method but does not escape from the thread.

    Note: The converse is not possible (if it does not escape the method then it cannot escape the thread).

Essence of Choi et al.'s Approach

- Introducing of a simple program abstraction called connection graph:

  Intuitively, a connection graph captures the connectivity relationship between heap allocated objects and object references.

- Demonstrating that escape analysis boils down to a reachability problem within connections graphs:

  If an object is reachable from an object that might escape, it might escape as well.

Experimental Results Reported by Choi et al.

…based on 10 benchmark programs:

- Percentage of objects that may be allocated on the stack:
  Up to $70+\%$, with a median of 19%.
- Percentage of all lock operations eliminated:
  From 11% to 92%, with a median of 51%.
- Overall execution time reduction:
  From 2% to 23%, with a median of 7%.

These results make escape analysis and the optimizations based theron whorthwhile.

Escape States

The analysis uses a simple lattice to represent different escape states:

$$
\begin{array}{c}
\mathit{NoEscape}\ (\top) \\
| \\
\mathit{ArgEscape} \\
| \\
\mathit{GlobalEscape}\ (\bot)
\end{array}
$$

State	Escapes the method	Escapes the thread
NoEscape	no	no
ArgEscape	may (via args)	no
GlobalEscape	may	may

Using Escape Information

All objects which are marked

- NoEscape: are stack-allocatable in the method where they are created.
- NoEscape or ArgEscape: are local to the thread in which they are created; hence synchronization statements in accessing these objects can be eliminated.

Chapter 15.3.1

Connection Graphs

Connection Graphs

We are interested only in

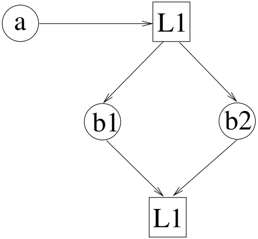
- following the object O from its point of allocation.
- knowing which variables reference O.
- and which other objects are referenced by O fields.

We "abstract out" the referencing information, using a graph structure where

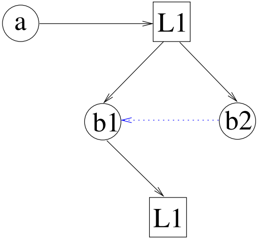
- a circle node represents a variable.
- a square node represents objects in the heap.
- an edge from circle to square represents a reference.
- an edge from square to circle represents ownership of fields.

Example: Connection Graphs

```
A a  = new A();  // line L1
a.b1 = new B();  // line L2
a.b2 = a.b1;     // line L3
```



Simple Version



Using Deferred Edges

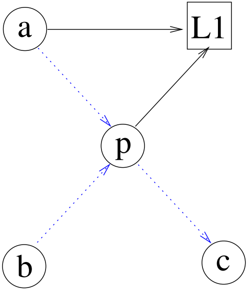
An edge drawn as a dotted arrow is called a deferred edge and shows the effect of an assignment from one variable to another (example: created by the assignment in line 3) $\leadsto$ improves efficiency of the approach.

Chapter 15.3.2
Intraprocedural Setting

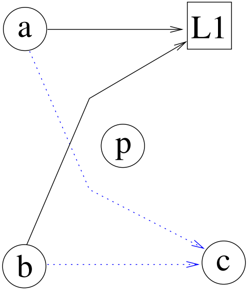
Intraprocedural Abstract Interpretation

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*):



Before



After

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.

Analyzing Statements (2)

p = q.f; If q does not point at any object node then a phantom node is created and an edge from q to the new node is added. Then *byPass*(p) is applied and deferred edges are added from p to all the f nodes that q is connected to by field edges.

For each statement one graph represents the state of the program at the statement.

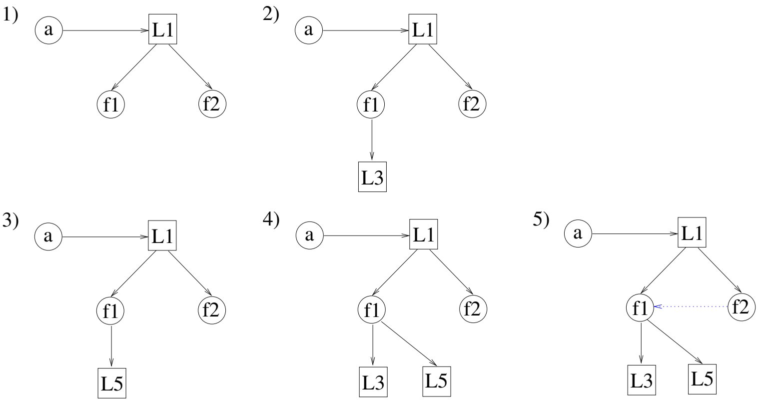
At a point where two or more control paths converge, the connection graphs from each predecessor statements are merged.

Example: Connection Graphs (1)

Suppose that the code inside some method is as follows. The declarations of classes `A`, `B1` and `B2` are omitted.

```
A a = new A();       // line L1
if (i > 0)
   a.f1 = new B1(); // line L3
else
   a.f1 = new B2(); // line L5
a.f2 = a.f1;         // line L6
```

Example: Connection Graphs (2)



G_1: out: A a = new A(); // line L1
G_2: out: a.f1 = new B1(); // line L3
G_3: out: a.f1 = new B2(); // line L5
G_4: out: $G_2 \cup G_3$
G_5: out: a.f2 = a.f1; // line L6

Chapter 15.3.3
Interprocedural Setting

Interprocedural Abstract Interpretation (1)

Analyzing methods:

- It is necessary to analyze each method in the reverse order implied by the call graph.
- If method A may call methods B and C, then B and C should be analyzed before A.
- Recursive edges in the call graph are ignored when determining the order.
- Java has virtual method calls – at a method call site where it is not known which method implementation is being invoked, the analysis must assume that all of the possible implementations are called, combining the effects from all the possibilities.
- The interprocedural analysis iterates over all the methods in the call graph until the results converge (fixed point).

Interprocedural Abstract Interpretation (2)

- A call to a method M is equivalent to copying the actual parameters (i.e. the arguments being passed in the method call) to the formal parameters, then executing the body of M, and finally copying any value returned by M as its result back to the caller.
- If M has already been analyzed intraprocedurally following the approach described above, the effect of M can be summarized with a connection graph. That summary information eliminates the need to re-analyze M for each call site in the program.

Analysis Results (1)

After the operation *byPass* has been used to eliminate all deferred edges, the connection graph can be partitioned into three subgraphs:

Global escape nodes: All nodes reachable from a node whose associated state is *GlobalEscape* are themselves considered to be global escape nodes (Subgraph 1)

- the nodes initially marked as *GlobalEscape* are the static fields of any classes and instances of any class that implements the `Runnable` interface.

Argument escape nodes: All nodes reachable from a node whose associated state is *ArgEscape*, but are not reachable from a *Global Escape* node. (Subgraph 2)

- the nodes initially marked as *ArgEscape* are the argument nodes $a_1, \ldots, a_n$.

Analysis Results (2)

No escape nodes: All other nodes have *NoEscape* status. (Subgraph 3).

The third subgraph represents the summary information for the method because it shows which objects can be reached via the arguments passed to the method.

All objects created within a method M and that have the *NoEscape* status after the three subgraphs have been determined can be safely allocated on the stack.

Chapter 15.4

References, Further Reading

Further Reading for Chapter 15 (1)

- John-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. *Escape Analysis for Java*. In Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'99), 1-19, 1999.
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)
- Hemant D. Pande, Barbara Ryder. *Data-flow-based Virtual Function Resolution*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 238-254, 1996.

Further Reading for Chapter 15 (2)

- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 1st edition, CRC Press, 2002. (Chapter 6, Optimizations for Object-Oriented Languages)
- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 2nd edition, CRC Press, 2008. (Chapter 13, Optimizations for Object-Oriented Languages)

Part V
Conclusions and Perspectives

Chapter 16
Conclusions, Emerging and Future Trends

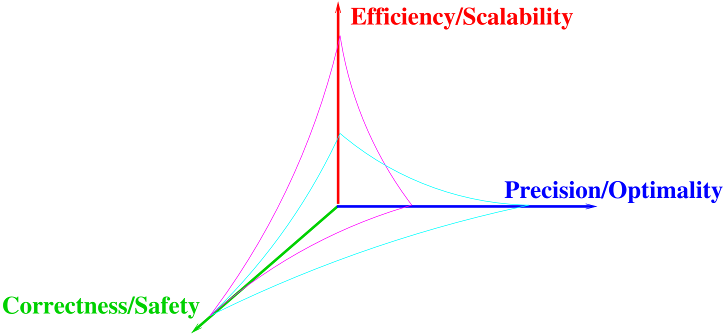
Chapter 16.1

Reconsidering Optimization

Program Analysis and Optimization (1)

...takes place in the area of conflict between

- Correctness, safety
- Precision, optimality
- Efficiency, scalability



Program Analysis and Optimization (2)

In principle

- Correctness/safety, precision/optimality, and efficiency/scalability can be traded for each other.

For example

- Iterative Compilation: Analytically, experimentally

  ...trades efficiency/scalability for precision/optimality.
- Adaptive Compilation: Experimentally

  ...trades efficiency/scalability for precision/optimality.
- Aggressive Optimization

  ...trades safety/correctness and/or efficiency/scalability for impact (rather than precision/optimality)

Program Analysis and Optimization (3)

Different fields also impose different performance demands:

- Compilation – trading precision/optimality for efficiency/scalability
  - Interactive, user: high
  - Batch, embedded systems compilation: moderate
  - Dynamic: extremely high
- Verification – trading efficiency/scalability for precision/optimality
  - moderate to low notwithstanding as fast as possible
- On-line monitoring/verification – trading precision/optimality for efficiency/scalability
  - real-time empowered (autonomous systems,...)

The characteristics and demands of an application scenario has a tremendous impact on the kind of analyses and transformations/optimizations which are considered reasonable.

Optimization worth the Effort?

...which options do we have if our program is too slow?

A radical view:

- Option 1: Buying new hardware!

  Moore's Law. Hardware performance gains double the computing power every 18 months.

- Option 2: Buying a new compiler!

  Proebsting's Law. Compiler optimizations gains double the computing power every 18 years.

Note: Proebsting's Law above is a corollary of his finding/ observation:

- *"Compiler optimizations have yielded annual performance gains an order of magnitude worse than hardware performance gains."*

Optimization of Little to No Relevance?

No, by contrast.

- Program analysis and optimization are more important than ever these days, and will so continue in the years to come.

Which evidence do we have?

Most importantly

- Moore's Law is vanishing: "The end" of Moore's Law due to physical limitations is foreseeable.
  - Waiting for the next processor generation with higher clock rate is no longer an option:

    There is no free lunch any longer!
- The improvement by compiler optimizations is always on top of any improvement by hardware advancements.

Optimization of the Highest Relevance

In fact, in response to the foreseeable "end" of Moore's Law

- All major chip vendors switched their focus from processors with higher and higher clock rates to many and multi-core processors.

Again

- There is no free lunch any longer!

In the words of a speaker at the CGO 2007 conference:

> *We asked for more computing power.*
> *We received more processor cores.*
>
> Speaker at CGO 2007

Hence

- New parallelization and optimization techniques are required!

Drivers of the Relevance of Optimization (1)

New advances in hardware and software demand strong compiler and optimizer support:

- Parallelism
  - Hardware/processors: Many/multi-core processors, CPUs, GPUs, GPGPUs, FPGAs, and other accelarators, heterogeneous hardware,...
  - Software: Parallel languages, parallelization of sequential programs (legacy software),...
  - New computing paradigms: Cloud computing, software-as-a-service,...
- Embedded and cyber-physical systems
  - Mobile systems: Laptops, tablets, smartphones,...
  - Autonomous mobile systems, (safety-critical) real-time systems: Robots in outer space, in co-working spaces with humans, fully autonomous cars, trains, subways, airplanes, ships,...) impose rigorous demands for safety and security, performance, power consumption, etc.

Drivers of the Relevance of Optimization (2)

Grand Challenges of Informatics pose new and strong demands on compilers and optimization, e.g.:

- The Verifying Compiler, Sir Tony Hoare.
  - Related Endeavours
    - Compiler verification: ProCoS, Verifix, CompCert,...
    - Translation/optimization validation: C3PRO, TVOC, CVT, VOC CovaC,...
    - ...
- Verlässlichkeit von Software, Gesellschaft für Informatik e.V. (GI), Fachbereich Softwaretechnik.

...contributions and advances in compiler construction, program analysis and optimization are crucial for successfully mastering these and other (grand) challenges.

Drivers of the Relevance of Optimization (3)

...research on optimization impacts other research fields and vice versa:

- Programm analysis and optimization
- Software engineering
  - Program understanding, program debugging, program re-engineering, program re-factoring, program (re-) specification,...
  - Model-driven code generation, model-driven code transformation,...
- Safety and security/privacy analysis
  - E.g., individual code generation for each compiler run to enhance security (Michael Franz, Stefan Brunthaler et al.)
- Language and compiler design
- Hardware/processor design
- ...

...mutually benefit of and challenge each other.

Chapter 16.2
Summary, Looking Ahead

Summing up, Looking ahead (1)

All this shows:

New topics in research on optimization pop up:

- Portable performance
  - Write once, run everywhere with the highest performance (CPUs, GPUs, GPGPUs, FPGAs, Multi-/Many-core architectures, Heterogeneous architectures,...)
- Power consumption
  - Mobile devices: laptops, tablets, smartphones (Pokémon Go), robots,...
  - Outer-space objects: spacecrafts, satellites, robots (MER-A Spirit, MER-B Opportunity, ESA Rosetta, ESA Philae,...),...
- Green IT
  - Power consumption (in the large)
- ...

Summing up, Looking ahead (2)

Established topics in research on optimization gain new momentum and experience a renaissance:

- Parallelization
  - Parallelism for the masses (parallelism is no longer a niche for the expert).
- Performance
  - Moore's law is nearing its end due to physical limits.
- Resource Analysis
  - Embedded and cyber-physical systems are ubiquitous.
    - Size, Power, Performance (WCET, ACET)
- ...

Summing up, Looking ahead (3)

For all these reasons, it is fair to say:

Compiler construction, program analysis, and optimization is

- a vibrant, theoretically and practically relevant field of research in informatics and will so remain in the years to come.
- among the most influential fields for the further progress and advancement of informatics.

Summing up, Looking ahead (4)

Key Issues

- Keeping pace with advances in software and hardware design.
- Impacting language and hardware design.
- Complementing the well-established and powerful theory of program analysis with an equally powerful theory of program transformations.

Summing up, Looking ahead (5)

Overall

- The future of the field of compiler construction, program analysis, and optimization is bright!

In particular

- Compiler construction, program analysis, and optimization are an unexhaustable source of challenging theoretically and practically relevant topics for PhD, master, and bachelor theses.

Chapter 16.3

References, Further Reading

Further Reading for Chapter 16 (1)

On Moore's and Proebsting's Laws

- Gordon E. Moore. *Cramming More Components onto Integrated Circuits*. Electronics 38(8), 114-117, 1965. http://web.eng.fiu.edu/npala/eee6397ex/gordon_moore_1965_article.pdf

- Gordon E. Moore. *Progress in Digital Integrated Electronics*. International Electron Devices Meeting, IEEE, IEDM Tech. Digest, 1113, 1975. http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf

- Tom Simonite. *Moore's Law Is Dead. Now What?* MIT Technology Review, 2016. https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/

Further Reading for Chapter 16 (2)

- M. Mitchell Waldrop. *More than Moore*. Nature 530(7589):144-147, 2016. `http://www.nature.com/polopoly_fs/1.19338!/menu/main/topColumns/topLeftColumn/pdf/530144a.pdf`
- Todd A. Proebsting. *Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years*. `http://proebsting.cs.arizona.edu/law.html`

Iterative Compilation

- Grigori Fursin, Michael F.P. O'Boyle, Peter M.W. Knijnenburg. *Evaluating Iterative Compilation*. In Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing (LCPC 2002), Revised Papers, Springer-V., LNCS 2481, 362-376, 2005.

Further Reading for Chapter 16 (3)

- Toru Kisuki, Peter M.W. Knijnenburg, Michael F.P. O'Boyle, François Bodin, Harry A.G. Wijshoff. *A Feasa- bility Study in Iterative Compilation*. In Proceedings of the 2nd International Symposium on High Performance Computing (ISHPC'99), Springer-V., LNCS 1615, 121-132, 1999.
- Peter M.W. Knijnenburg, Toru Kisuki, Michael F.P. O'Boyle. *Iterative Compilation*. In Proceedings of the 1st Workshop on Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS 2001), Springer-V., LNCS 2268, 171-187, 2002.

Further Reading for Chapter 16 (4)

Adaptive Compilation

- L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *Finding Effective Compilation Sequences*. In Proceedings of the 2004 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), 231-239, 2004.
- Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *ACME: Adaptive Compilation Made Efficient*. In Proceedings of the 2005 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005), 69-77, 2005.

Further Reading for Chapter 16 (5)

- Keith D. Cooper, Devika Subramanian, Linda Torczon. *Adaptive Optimizing Compilers for the 21st Century*. The Journal of Supercomputing 23(1):7-22, 2002.

Aggressive Optimization

- Li-Ling Chen, Youfeng Wu. *Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation*. In Proceedings of the 2003 IEEE International Conference on Parallel Processing (ICPP 2003), 607-614, 2003.

Further Reading for Chapter 16 (6)

- Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Leaf Petersen. *A Verifiable SSA Program Representation for Aggressive Compiler Optimization*. In Conference Record of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), 397-408, 2006.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 1.5, Placement of Optimizations in Aggressive Optimizing Compilers)

Further Reading for Chapter 16 (7)

Grand Challenges

- Charles A.R. Hoare. *The Verifying Compiler: A Grand Challenge for Computing Research*. Journal of the ACM 50(1):63-69, 2003.
- Gesellschaft für Informatik e.V. (GI), Fachbereich Softwaretechnik. *Verlässlichkeit von Software*, 2014. `https://www.gi.de/themen/grand-challenges/verlaesslichkeit-von-software.html`

Further Reading for Chapter 16 (8)

Compiler Verification

- Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, Jean Souyris. *Towards Formally Verified Optimizing Compilation in Flight Control Software*. In Proceedings of the Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011), 59-68, 2011.
- Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard von Karger, Yassine Lakhnech, Hans Langmaack, Markus Müller-Olm. *Provably Correct Compiler Development and Implementation*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-V., LNCS 641, 141-155, 1992.

Further Reading for Chapter 16 (9)

- Sabine Glesner, Gerhard Goos, Wolf Zimmermann. *Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers)*. it - Information Technology 46(5):265-276, 2004.

- Gerhard Goos, Wolf Zimmermann. *Verification of Compilers*. Correct System Design: Recent Insight and Advances. Springer-V., LNCS 1710, 201-230, 1999.

- Andreas Krall. *Correct Compilers for Correct Processors*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: `http://old.hipeac.net/system/files/hipeac14.pdf`

Further Reading for Chapter 16 (10)

- Hans Langmaack. *Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit*. it+ti - Informationstechnik und Technische Informatik 39(3):41-47, 1997.
- Hans Langmaack: *The ProCoS Approach to Correct Systems*. Real-Time Systems 13(3):253-275, 1997.
- Xavier Leroy. *Formal Verification of a Realistic Compiler*. Communications of the ACM 52(7):107-115, 2009.

Further Reading for Chapter 16 (11)

- Xavier Leroy. *Formally Verifying a Compiler: Why? How? How far?* Invited Talk, In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011), xxxi, 2011.
- Xavier Leroy. *Compiler Verification for Fun and Profit*. Invited Talk, In Proceedings of the 2014 IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD 2014), 9, 2014.
- John McCarthy, James Painter. *Correctness of a Compiler for Arithmetical Expressions*. Mathematical Aspects of Computer Science, ser. Proceedings of Symposia in Applied Mathematics, Vol. 19, American Mathematical Society, 33-41, 1967.

Further Reading for Chapter 16 (12)

Robin Milner, R. Weyrauch. *Proving Compiler Correctness in a Mechanized Logic*. In Proceedings of the 7th Annual Machine Intelligence Workshop, ser. Machine Intelligence, B. Meltzer and D. Michie, Eds., Vol. 7., Edinburgh University Press, 51-72, 1972.

Wolf Zimmermann. *On the Correctness of Transformations in Compiler Back-Ends*. In Proceedings of the 1st First International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004), Springer-V, LNCS 4313, 74-95, 2004.

Further Reading for Chapter 16 (13)

Translation Validation

- Clark W. Barret, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, Lenore Zuck. *TVOC: A Translation Validator for Optimizing Compilers*. In Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005), Springer-V., LNCS 3576, 291-295, 2005.
- Aditya Kanade, Amitabha Sanyal, Uday Khedker. *A PVS based Framework for Validating Compiler Optimizations*. In Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 108-117, 2006.

Further Reading for Chapter 16 (14)

- George C. Necula. *Translation Validation for an Optimizing Compiler*. In Proceedings of the 20th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000), ACM SIGPLAN Notices 35:83-95, 2000.
- Amir Pnueli, Michael Siegel, Eli Singerman. *Translation Validation*. In Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Springer-V., LNCS 1384, 151-166, 1998.

Further Reading for Chapter 16 (15)

- Amir Pnueli, Ofer Strichman, Michael Siegel. *Translation Validation for Synchronous Languages*. In Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98), Springer-V., LNCS 1443, 235-246, 1998.
- Amir Pnueli, Ofer Strichman, Michael Siegel. *The Code Validation Tool (CVT) Automatic Verification of a Compilation Process*. International Journal on Software Tools for Technology Transfer 2(2):192-201, 1998.
- Jean-Baptiste Tristan, Xavier Leroy. *Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations*. In Conference Record of the 35th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2008), 17-27, 2008.

Further Reading for Chapter 16 (16)

- Jean-Baptiste Tristan, Xavier Leroy. *Verified Validation of Lazy Code Motion*. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), ACM SIGPLAN Notices 44:316-326, 2009.
- Anna Zaks, Amir Pnueli. *CovaC: Compiler Validation by Program Analysis of the Cross-product*. In Proceedings of the 15th International Symposium on Formal Methods (FM 2008), Springer-V., LNCS 5014, 35-51, 2008.
- Lenore Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg. *VOC: A Methodology for Translation Validation of Optimizing Compilers*. Journal of Universal Computer Science 9(3):223-247, 2003.

Further Reading for Chapter 16 (17)

New Programming Models and Architectures

- David Kaeli. *The Road to new Programming Models and Architectures for Future Heterogeneous Systems*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: http://old.hipeac.net/system/files/David%20Kaeli_reduced.pdf

- Margaret Martonosi. *Power-Aware Computing: Then, Now, and into the Future*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: http://old.hipeac.net/system/files/MartonosiHIPEACfinal.pdf

Further Reading for Chapter 16 (18)

András Vajda. *Programming Many-Core Chips*. Springer-V., 2011. (Chapter 1.1, The End of Endless Scalability; Chapter 2, Multi-core and Many-core Processor Architecture; Chapter 3, State of the Art Multi-core Operating Systems; Chapter 10, Looking Ahead)

References

Recommended Reading

...for deepened and independent studies.

- I Textbooks
- II On-line Tutorials
- III On-line Resources of Compilers and Compiler Writing Tools
- IV Monographs and Volumes
- V Articles

I Textbooks (1)

- Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1997.
- Andrew W. Appel with Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.
- Andrew W. Appel with Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2. aktualisierte Auflage, 2008.

I Textbooks (2)

- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002.
- André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
- Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012.
- Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013.
- Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.

I Textbooks (3)

- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004.
- Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002.
- Marcel Erné. *Einführung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
- Shimon Even. *Graph Algorithms*. Pitman, 1979.
- C. Fischer, R. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Co., Inc. Menlo Park, CA, 1988.

I Textbooks (4)

- Helmuth Gericke. *Theorie der Verbände*. Bibliographisches Institut, 2. Auflage, 1967.
- George Grätzer. *General Lattice Theory*. Birkhäuser, 2nd edition, 2003.
- Dick Grune, Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-V., 2nd edition, 2008.
- Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, Koen G. Langendoen. *Modern Compiler Design*. Springer-V., 2nd edition, 2012.
- Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001.

I Textbooks (5)

- Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
- Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.
- Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009.
- Janusz Laski, William Stanley. *Software Verification and Analysis*. Springer-V., 2009.
- Uday P. Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

I Textbooks (6)

- Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998.
- David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008.
- 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.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007.

I Textbooks (7)

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

Peter Pepper, Petra Hofstedt. *Funktionale Programmie- rung: Sprachdesign und Programmiertechnik*. Springer-V., 2006.

Helmut Seidl, Reinhard Wilhelm, Sebastian Hack. *Compiler Design: Analysis and Transformation*. Springer-V., 2012.

Patrick D. Terry. *Compilers and Compiler Generators: An Introduction with C++*. International Thomson Computer Press, 1997.

I Textbooks (8)

- Patrick D. Terry. *Compiling with C# and Java*. Addison-Wesley, 2005.
- András Vajda. *Programming Many-Core Chips*. Springer-V., 2011.
- William M. Waite, Gerhard Goos. *Compiler Construction*. Springer-V., 1984.
- William M. Waite, Lynn R. Carter. *An Introduction to Compiler Construction*. HarperCollins College Publishers, 1993.
- Reinhard Wilhelm, Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

I Textbooks (9)

- Reinhard Wilhelm, Dieter Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-V., 2. Auflage, 1997.
- Reinhard Wilhelm, Helmut Seidl. *Compiler Design: Virtual Machines*. Springer-V., 2010.
- Reinhard Wilhelm, Helmut Seidl, Sebastian Hack. *Compiler Design: Syntactic and Semantic Analysis*. Springer-V., 2013.

II On-line Tutorials (1)

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

III On-line Resources of Compilers and Compiler Writing Tools (1)

- German National Research Center for Information Technology, Fraunhofer Institute for Computer Architecture and Software Technology. *The Catalog of Compiler Construction Tools*, 1996-2006. http://catalog.compilertools.net/
- Compilers.net Team. *Search Machine on Compilers and Programming Languages, Directory of Compiler and Language Resources*, 1997-2007. http://www.compilers.net

III On-line Resources of Compilers and Compiler Writing Tools (2)

- Nullstone Corporation. *The Compiler Connection: A Resource for Compiler Developers and Those who use Their Products and Services (Books, Tools, Techniques, Conferences, Jobs and more*, 2011-2012.
  `http://www.compilerconnection.com`
- Olaf Langmack. *Catalog of Compiler Construction Products 01-98*. 13th Issue, 1998.
  `http://compilers.iecc.com/tools.html`
- William M. Waite, Uwe Kastens et al. *Eli: Translator Construction made Easy*, 1989-today.
  `http://eli-project.sourceforge.net/`

III On-line Resources of Compilers and Compiler Writing Tools (3)

- Free Software Foundation (FSF). *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/
- LLVM Foundation. *The LLVM Compiler Infrastructure*. http://llvm.org/
- The SUIF Group (Monica S. Lam et al.), Stanford University. *The SUIF (Stanford University Intermediate Format) Compiler System*. http://suif.stanford.edu/

III On-line Resources of Compilers and Compiler Writing Tools (4)

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

IV Monographs and Volumes (1)

- Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-V., LNCS 1428, 1998.
- Yuri V. Matijasevic. *Hilbert's Tenth Problem*. MIT Press, 1993.
- Markus Müller-Olm. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*. Springer-V., LNCS 3800, 2006.
- Stephen S. Muchnick, Neil D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.
- Oliver Rüthing. *Interacting Code Motion Transformations: Their Impact and Their Complexity*. Springer-V., LNCS 1539, 1998.

IV Monographs and Volumes (2)

- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.
- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2nd edition, 2008.
- John Whaley. *Context-sensitive Pointer Analysis using Binary Decision Diagrams*. PhD Thesis, Stanford University, CA, USA, 2007.

V Articles (1)

- Frances E. Allen, John A. Cocke. *A Program Data Flow Analysis Procedure*. Communications of the ACM 19(3):137-147, 1976.
- Frances E. Allen, John Cocke, Ken Kennedy. *Reduction of Operator Strength*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 3, 79-101.
- L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *Finding Effective Compilation Sequences*. In Proceedings of the 2004 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004), 231-239, 2004.

V Articles (2)

- Bowen Alpern, Mark N. Wegman, F. Kenneth Zadeck. *Detecting Equality of Variables in Programs*. In Conference Record of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 1-11, 1988.
- Clement A. Baker-Finch, Kevin Glynn, Simon L. Peyton Jones. *Constructed Product Result Analysis for Haskell*. Journal of Functional Programing 14(2):211-245, 2004.
- Thomas Ball, Sriram K. Rajamani. *Bebop: A Path-Sensitive Interprocedural Dataflow Engine*. In Proceedings of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), 97-103, 2001.

V Articles (3)

- Clark W. Barret, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, Lenore Zuck. *TVOC: A Translation Validator for Optimizing Compilers*. In Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005), Springer-V., LNCS 3576, 291-295, 2005.
- Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, Jean Souyris. *Towards Formally Verified Optimizing Compilation in Flight Control Software*. In Proceedings of the Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011), 59-68, 2011.

V Articles (4)

Marc Berndl, Ondřej Lhoták, Feng Qian. Laurie Hendren, Navindra Umanee. *Points-to Analysis using BDDs*. In Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003), ACM SIGPLAN Notices 38:103-114, 2003.

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

V Articles (5)

- Ras Bodik, Rajiv Gupta. *Partial Dead Code Elimination using Slicing Transformations*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM SIGPLAN Notices 32:159-170, 1997.

- Ras Bodik, Rajiv Gupta. *Register Pressure Sensitive Redundancy Elimination*. In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 107-121, 1999.

V Articles (6)

- Ras Bodik, Rajiv Gupta, Mary Lou Soffa. *Complete Removal of Redundant Expressions*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), ACM SIGPLAN Notices 33(5):1-14, 1998.
- Martin Bravenboer, Yannis Smaragdakis. *Strictly Declarative Specification of Sophisticated Points-to Analyses*. In Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA 2009), 243-262, 2009.

V Articles (7)

Preston Briggs, Keith D. Cooper. *Effective Partial Redundancy Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):159-170, 1994.

Preston Briggs, Keith D. Cooper, L. Taylor Simpson. *Value Numbering*. Software: Practice and Experience 27(6):701-724, 1997.

Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard von Karger, Yassine Lakhnech, Hans Langmaack, Markus Müller-Olm. *Provably Correct Compiler Development and Implementation*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-V., LNCS 641, 141-155, 1992.

V Articles (8)

- Qiong Cai, Jingling Xue. *Optimal and Efficient Speculation-based Partial Redundancy Elimination*. In Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2003), 91-104, 2003.
- David Callahan, Steve Carr, Ken Kennedy. *Improving Register Allocation for Subscripted Variables*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), ACM SIGPLAN Notices 25:53-65, 1990.
- Steve Carr, Ken Kennedy. *Scalar Replacement in the Presence of Conditional Control Flow*. Software: Practice and Experience 24(1):51-77, 1994.

V Articles (9)

David R. Chase, Mark N. Wegmann, F. Kenneth Zadeck. *Analysis of Pointers and Structures*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), ACM SIGPLAN Notices 25:296-310, 1990.

Li-Ling Chen, Youfeng Wu. *Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation*. In Proceedings of the 2003 IEEE International Conference on Parallel Processing (ICPP 2003), 607-614, 2003.

John-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. *Escape Analysis for Java*. In Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'99), 1-19, 1999.

V Articles (10)

- Fred C. Chow, Sun Chan, Robert Kennedy, Shing-Ming Liu, Raymond Lo, Peng Tu. *A New Algorithm for Partial Redundancy Elimination based upon SSA Form*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM SIGPLAN Notices 32(5):273-286, 1997.
- Cliff Click. *Global Code Motion, Global Value Numbering*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):246-257, 1995.
- Cliff Click, Keith D. Cooper. *Combing Analyses, Combining Optimizations*. ACM Transactions on Programming Languages and Systems 17(2):181-196, 1995.

V Articles (11)

John Cocke, Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 2nd Revised Version, 771 pages, 1970.

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

Keith D. Cooper, Jason Eckhardt, Ken Kennedy. *Redundancy Elimination Revisited*. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008), 12-21, 2008.

V Articles (12)

- Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, Todd Waterman. *ACME: Adaptive Compilation Made Efficient*. In Proceedings of the 2005 ACM SIGPLAN- SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005), 69-77, 2005.
- Keith D. Cooper, L. Taylor Simpson, Christopher A. Vick. *Operator Strength Reduction*. ACM Transactions on Programming Languages and Systems 23(5):603-625, 2001.
- Keith D. Cooper, Devika Subramanian, Linda Torczon. *Adaptive Optimizing Compilers for the 21st Century*. The Journal of Supercomputing 23(1):7-22, 2002.

V Articles (13)

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. ACM Transactions on Programming Languages and Systems 13(4):451-490, 1991.

Dhananjay M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. International Journal of Computer Mathematics 27:1-14,31-32, 1989.

Dhananjay M. Dhamdhere. *Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise*. ACM Transactions on Programming Languages and Systems 13(2):291-294, 1991, Technical Correspondence.

V Articles (14)

Dhananjay M. Dhamdhere. *E-path_pre: Partial Redundancy Elimination Made Easy*. ACM SIGPLAN Notices 37(8):53-65, 2002.

Dhananjay M. Dhamdhere, J. R. Isaac. *A Composite Algorithm for Strength Reduction and Code Movement Optimization*. International Journal of Computer and Information Sciences 9(3):243-273, 1980.

Karl-Heinz Drechsler, Manfred P. Stadel. *A Solution to a Problem with Morel and Renvoise's "Global Optimization by Suppression of Partial Redundancies"*. ACM Transactions on Programming Languages and Systems 10(4):635-640, 1988, Technical Correspondence.

V Articles (15)

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

- Maryam Emami, Rakesh Ghiya, Laurie J. Hendren. *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):242-256, 1994.

- Andrei P. Ershov. *On Programming of Arithmetic Operations*. Communications of the ACM 1(8):3-6, 1958. (Three figures from this article are in CACM 1(9):16).

V Articles (16)

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

V Articles (17)

- Grigori Fursin, Michael F.P. O'Boyle, Peter M.W. Knijnenburg. *Evaluating Iterative Compilation*. In Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing (LCPC 2002), Revised Papers, Springer-V., LNCS 2481, 362-376, 2005.
- Gesellschaft für Informatik e.V. (GI), Fachbereich Softwaretechnik. *Verlässlichkeit von Software*, 2014. `https://www.gi.de/themen/grand-challenges/verlaesslichkeit-von-software.html`
- Alfons Geser, Jens Knoop, Gerald Lüttgen, Oliver Rü- thing, Bernhard Steffen. *Non-Monotone Fixpoint Iter- ations to Resolve Second Order Effects*. In Proceedings of the 6th International Conference on Compiler Construc- tion (CC'96), Springer-V., LNCS 1060, 106-120, 1996.

V Articles (18)

- Rakesh Ghiya, Laurie J. Hendren. *Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 1-15, 1996.
- Sabine Glesner, Gerhard Goos, Wolf Zimmermann. *Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers)*. it - Information Technology 46(5):265-276, 2004.
- Gerhard Goos, Wolf Zimmermann. *Verification of Compilers*. Correct System Design: Recent Insight and Advances. Springer-V., LNCS 1710, 201-230, 1999.

V Articles (19)

- Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, William M. Waite. *Eli: A Complete, Flexible Compiler Construction System*. Communications of the ACM 35(2):121-131, 1992.
- Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Redundancy Elimination Using Speculation*. In Proceedings of the 6th IEEE International Conference on Computer Languages (ICCL'98), 230-239, 1998.
- Rajiv Gupta, David A. Berson, Jesse Z. Fang. *Path Profile Guided Partial Dead Code Elimination using Predication*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 102-115, 1997.

V Articles (20)

- Ben Hardekopf, Calvin Lin. *Semi-sparse Flow-sensitive Pointer Analysis*. In Conference Record of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009), 226-238, 2009.
- Charles A.R. Hoare. *The Verifying Compiler: A Grand Challenge for Computing Research*. Journal of the ACM 50(1):63-69, 2003.
- R. Nigel Horspool, H. C. Ho. *Partial Redundancy Elimination Driven by a Cost-benefit Analysis*. In Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering (CSSE'97), 111-118, 1997.

V Articles (21)

- R. Nigel Horspool, David J. Pereira, Bernhard Scholz. *Fast Profile-based Partial Redundancy Elimination*. In Proceedings of the 7th Joint Modular Languages Conference (JMLC 2006), 362-376, 2006.

- Susan Horwitz, Alan J. Demers, Tim Teitelbaum. *An Efficient General Iterative Algorithm for Dataflow Analysis*. Acta Informatica 24(6):679-694, 1987.

- Bertrand Jeannet, Alexey Logivon, Thomas W. Reps, Mooly Sagiv. *A Relational Approach to Interprocedural Shape Analysis*. In Proceedings of the 11th Static Analysis Symposium (SAS 2004), Springer-V., LNCS 3248, 246-264, 2004.

V Articles (22)

- S. M. Joshi, Dhananjay M. Dhamdhere. *A Composite Hoisting- strength Reduction Transformation for Global Program Optimization – Part I and Part II*. International Journal of Computer Mathematics 11:21-41,111-126, 1982.

- David Kaeli. *The Road to new Programming Models and Architectures for Future Heterogeneous Systems*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: `http://old.hipeac.net/system/files/David%20Kaeli_reduced.pdf`

- John B. Kam, Jeffrey D. Ullman. *Global Data Flow Analysis and Iterative Algorithms*. Journal of the ACM 23:158-171, 1976.

V Articles (23)

- John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. Acta Informatica 7:305-317, 1977.
- Aditya Kanade, Amitabha Sanyal, Uday Khedker. *A PVS based Framework for Validating Compiler Optimizations*. In Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 108-117, 2006.
- Ken Kennedy. *Safety of Code Motion*. International Journal of Computer Mathematics 3(2-3):117-130, 1972.
- Robert Kennedy, Sun Chan, Shing-Ming Liu, Raymond Lo, Peng Tu, Fred C. Chow. *Partial Redundancy Elimination in SSA Form*. ACM Transactions of Pro- gramming Languages and Systems 32(3):627-676, 1999.

V Articles (24)

- Robert Kennedy, Fred C. Chow, Peter Dahl, Shing-Ming Liu, Raymond Lo, Mark Streich. *Strength Reduction via SSAPRE*. In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 144-158, 1998.
- Uday P. Khedker, Bageshri Karkare. *Efficiency, Precision, Simplicity, and Generality in Interprocedural Dataflow Analysis: Resurrecting the Classical Call Strings Method*. In Proceedings of the 17th International Conference on Compiler Construction (CC 2008), Springer-V., LNCS 4959, 213-228, 2008.

V Articles (25)

- Uday P. Khedker, Alan Mycroft, Prashant Singh Rawat. *Liveness-based Pointer Analysis*. In Proceedings of the 19th Static Analysis Symposium (SAS 2012), Springer-V., LNCS 7460, 265-282, 2012.

- Gary A. Kildall. *A Unified Approach to Global Program Optimization*. In Conference Record of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73), 194-206, 1973.

- Toru Kisuki, Peter M.W. Knijnenburg, Michael F.P. O'Boyle, François Bodin, Harry A.G. Wijshoff. *A Feasability Study in Iterative Compilation*. In Proceedings of the 2nd International Symposium on High Performance Computing (ISHPC'99), Springer-V., LNCS 1615, 121-132, 1999.

V Articles (26)

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

- Peter M.W. Knijnenburg, Toru Kisuki, Michael F.P. O'Boyle. *Iterative Compilation*. In Proceedings of the 1st Workshop on Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS 2001), Springer-V., LNCS 2268, 171-187, 2002.

V Articles (27)

- Kathleen Knobe, Vivek Sarkar. *Conditional Constant Propagation of Scalar and Array References Using Array SSA Form*. In Proceedings of the 5th Static Analysis Symposium (SAS'98), Springer-V., LNCS 1503, 33-56, 1998.
- Jens Knoop. *Parallel Constant Propagation*. In Proceedings of the 4th European Conference on Parallel Processing (Euro-Par'98), Springer-V., LNCS 1470, 445-455, 1998.
- Jens Knoop. *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.

V Articles (28)

- 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, Eduard Mehofer. *Optimal Distribution Assignment Placement*. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Springer-V., LNCS 1300, 364 - 373, 1997.

V Articles (29)

- Jens Knoop, Eduard Mehofer. *Interprocedural Distribution Assignment Placement: More than just Enhancing Intraprocedural Placing Techniques*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'97), 26-37, 1997.
- Jens Knoop, Oliver Rüthing. *Constant Propagation on the Value Graph: Simple Constants and Beyond*. In Proceedings of the 9th International Conference on Compiler Construction (CC 2000), Springer-V., LNCS 1781, 94-109, 2000.
- Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. In Proceedings of the 7th Brazilian Symposium on Programming Languages (SBLP 2003), 135-148, 2003.

V Articles (30)

- Jens Knoop, Oliver Rüthing. *Constant Propagation on Predicated Code*. Journal of Universal Computer Science 9(8):829-850, 2003. (Special issue for SBLP'03)
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. Journal of Programming Languages 1(1):71-91, 1993.

V Articles (31)

- 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 Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-V., LNCS 1381, 154-169, 1998.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Expansion-based Removal of Semantic Partial Redundancies.* In Proceedings of the 8th International Conference on Compiler Construction (CC'99), Springer-V., LNCS 1575, 91-106, 1999.

V Articles (32)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Partial Dead Code Elimination*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), ACM SIGPLAN Notices 29(6):147-158, 1994.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *The Power of Assignment Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), ACM SIGPLAN Notices 30(6):233-245, 1995.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retrospective: Lazy Code Motion*. In "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection", ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.

V Articles (33)

- 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.

- Kathleen Knobe, Vivek Sarkar. *Array SSA Form and its Use in Parallelization*. In Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98), 107-120, 1998.

V Articles (34)

- Donald E. Knuth. *An Empirical Study of Fortran Programs*. Software: Practice and Experience 1:105-133, 1971.
- Andreas Krall. *Correct Compilers for Correct Processors*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: `http://old.hipeac.net/system/files/hipeac14.pdf`
- Hans Langmaack. *Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit*. it+ti - Informationstechnik und Technische Informatik 39(3):41-47, 1997.
- Hans Langmaack: *The ProCoS Approach to Correct Systems*. Real-Time Systems 13(3):253-275, 1997.

V Articles (35)

Xavier Leroy. *Formal Verification of a Realistic Compiler*. Communications of the ACM 52(7):107-115, 2009.

Xavier Leroy. *Formally Verifying a Compiler: Why? How? How far?* Invited Talk, In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011), xxxi, 2011.

Xavier Leroy. *Compiler Verification for Fun and Profit*. Invited Talk, In Proceedings of the 2014 IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD 2014), 9, 2014.

Roman Manevich, Mooly Sagiv, Ganesan Ramalingam, John Field. *Partially Disjunctive Heap Abstraction*. In Proceedings of the 11th Static Analysis Symposium (SAS 2004), Springer-V., LNCS 3248, 265-279, 2004.

V Articles (36)

Roman Manevich, Boris Dogadov, Noam Rinetzky. *From Shape Analysis to Termination Analysis in Linear Time*. In Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), Springer-V., LNCS 9779, 426-446, 2016.

Ravi Mangal, Mayur Naik, Hongseok Yang. *A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join*. In Proceedings of the 23rd European Symposium on Programming (ESOP 2014), Springer-V., LNCS 8410, 513-533, 2014.

Thomas J. Marlowe, Barbara G. Ryder. *Properties of Data Flow Frameworks*. Acta Informatica 28(2):121-163, 1990.

V Articles (37)

- Stephen P. Masticola, Thomas J. Marlowe, Barbara G. Ryder. *Lattice Frameworks for Multisource and Bidirectional Data Flow Problems*. ACM Transactions on Programming Languages and Systems (TOPLAS) 17(5):777-803, 1995.
- Florian Martin. *PAG - An Efficient Program Analyzer Generator*. Journal of Software Tools for Technology Transfer 2(1):46-67, 1998.
- Margaret Martonosi. *Power-Aware Computing: Then, Now, and into the Future*. Invited Talk, 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2014), 2014. Slides: `http://old.hipeac.net/system/files/MartonosiHIPEACfinal.pdf`

V Articles (38)

- Yuri V. Matijasevic. *Enumerable Sets are Diophantine (In Russian)*. Dodl. Akad. Nauk SSSR 191, 279-282, 1970.
- Yuri V. Matijasevic. *What Should We Do Having Proved a Decision Problem to be Unsolvable?* Algorithms in Modern Mathematics and Computer Science 1979:441-448, 1979.
- Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Leaf Petersen. *A Verifiable SSA Program Representation for Aggressive Compiler Optimization*. In Conference Record of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), 397-408, 2006.

V Articles (39)

- Samuel P. Midkiff, José E. Moreira, Marc Snir. *A Constant Propagation Algorithm for Explicitly Parallel Programs*. International Journal of Computer Science 26(5):563-589, 1998.
- Matthew Might, Yannis Smaragdakis, David Van Horn. *Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. OO Program Analysis*. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), ACM SIGPLAN Notices 45(6):305-315, 2010.
- Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to and Side-effect Analyses for JAVA*. In Proceedings of the 6th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), 1-11, 2002.

V Articles (40)

Ana Milanova, Atanas Rountev, Barbara G. Ryder. *Parameterized Object Sensitivity for Points-to Analysis for JAVA*. ACM Transactions on Software Engineering and Methodology 14(4):431-477, 2005.

John McCarthy, James Painter. *Correctness of a Compiler for Arithmetical Expressions*. Mathematical Aspects of Computer Science, ser. Proceedings of Symposia in Applied Mathematics, Vol. 19, American Mathematical Society, 33-41, 1967.

Robin Milner, R. Weyrauch. *Proving Compiler Correctness in a Mechanized Logic*. In Proceedings of the 7th Annual Machine Intelligence Workshop, ser. Machine Intelligence, B. Meltzer and D. Michie, Eds., Vol. 7., Edinburgh University Press, 51-72, 1972.

V Articles (41)

- Gordon E. Moore. *Cramming More Components onto Integrated Circuits*. Electronics 38(8), 114-117, 1965. http://web.eng.fiu.edu/npala/eee6397ex/gordon_moore_1965_article.pdf

- Gordon E. Moore. *Progress in Digital Integrated Electronics*. International Electron Devices Meeting, IEEE, IEDM Tech. Digest, 1113, 1975. http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf

- Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.

V Articles (42)

- Markus Müller-Olm. *The Complexity of Copy Constant Detection in Parallel Programs*. In Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2001), Springer-V., LNCS 2010, 490-501, 2001.
- Markus Müller-Olm, Oliver Rüthing. *The Complexity of Constant Propagation*. In Proceedings of the European Symposium on Programming (ESOP 2001), Springer-V., LNCS 2028, 190-205, 2001.

V Articles (43)

- Markus Müller-Olm, Helmut Seidl. *Polynomial Constants are Decidable*. In Proceedings of the 9th Static Analysis Symposium (SAS 2002), Springer-V., LNCS 2477, 4-19, 2002.
- George C. Necula. *Translation Validation for an Optimizing Compiler*. In Proceedings of the 20th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000), ACM SIGPLAN Notices 35:83-95, 2000.
- Flemming Nielson. *Semantics-directed Program Analysis: A Tool-maker's Perspective*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 2-21, 1996.

V Articles (44)

- Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92), 96-108, 1992.
- Hemant D. Pande, Barbara Ryder. *Data-flow-based Virtual Function Resolution*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-V., LNCS 1145, 238-254, 1996.
- Viktor Pavlu, Markus Schordan, Andreas Krall. *Computation of Alias Sets from Shape Graphs for Comparison of Shape Analysis Precision*. In Proceedings of the 11th International IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2011), 25-34, 2011. [Best Paper Award SCAM 2011]

V Articles (45)

- Amir Pnueli, Ofer Strichman, Michael Siegel. *Translation Validation for Synchronous Languages*. In Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98), Springer-V., LNCS 1443, 235-246, 1998.
- Amir Pnueli, Michael Siegel, Eli Singerman. *Translation Validation*. In Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Springer-V., LNCS 1384, 151-166, 1998.
- Amir Pnueli, Ofer Strichman, Michael Siegel. *The Code Validation Tool (CVT) – Automatic Verification of a Compilation Process*. International Journal on Software Tools for Technology Transfer 2(2):192-201, 1998.

V Articles (46)

- Todd A. Proebsting. *Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years*. http://proebsting.cs.arizona.edu/law.html

- John H. Reif, Harry R. Lewis. *Symbolic Evaluation and the Global Value Graph*. In Conference Record of the 4th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), 104-118, 1977.

- John H. Reif, Harry R. Lewis. *Efficient Symbolic Analysis of Programs*. Aiken Computation Laboratory, Harvard University, TR-37-82, 1982.

V Articles (47)

- Tom Reps, Susan Horwitz, Mooly Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 49-61, 1995.
- Rishi Surendran, Rajkishore Barik, Jisheng Zhao, Vivek Sarkar. *Inter-iteration Scalar Replacement using Array SSA Form*. In Proceedings of the 23rd International Conference on Compiler Construction (CC 2014), Springer-V., LNCS 8409, 40-60, 2014.
- Barry K. Rosen. *High-level Data Flow Analysis*. Communications of the ACM 20(10):141-156, 1977.

V Articles (48)

- Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. *Global Value Numbers and Redundant Computations*. In Conference Record of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88), 12-27, 1988.
- Oliver Rüthing, Jens Knoop, Bernhard Steffen. *Detecting Equalities of Variables: Combining Efficiency with Precision*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Springer-V., LNCS 1694, 232-247, 1999.
- 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.

V Articles (49)

- Olin Shivers. *Control-Flow Analysis in Scheme*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88), ACM SIGPLAN Notices 23:164-174, 1988.

- Yannis Smaragdakis, Martin Bravenboer, Ondřej Lhoták. *Pick Your Contexts Well: Understanding Object-sensitivity*. In Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), 17-30, 2011.

- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-analysis Problems in Languages with Destructive Updating*. In Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 16-31, 1996.

V Articles (50)

- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Solving Shape-Analysis Problems in Languages with Destructive Updating*. ACM Transactions on Programming Languages and Systems 20(1):1-50, 1998.
- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. In Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), 105-118, 1999.
- Mooly Sagiv, Tom Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-valued Logic*. ACM Transactions on Programming Languages and Systems 24(3):217-298, 2002.

V Articles (51)

Helmut Seidl, Christian Fecht. *Interprocedural Analyses: A Comparison*. The Journal of Logic Programming 43:123-156, 2000.

Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.3, The Functional Approach to Interprocedural Analysis, 196-209.

V Articles (52)

- Micha Sharir, Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 7.4, The Call-String Approach to Interprocedural Analysis, 210-217; Chapter 7.6, An Approximative Call-String Approach, 225-230.
- 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.

V Articles (53)

- Tom Simonite. *Moore's Law Is Dead. Now What?* MIT Technology Review, 2016. https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/

- 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.

V Articles (54)

- Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. In Proceedings of the 14th International Conference on Mathematical Foundations of Computer Science (MFCS'89), Springer-V., LNCS 379, 481-490, 1989.
- Bernhard Steffen, Jens Knoop. *Finite Constants: Characterizations of a New Decidable Set of Constants*. Theoretical Computer Science 80(2):303-318, 1991.
- 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.

V Articles (55)

- 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 of the 6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95), Springer-V., LNCS 915, 651-665, 1995.

V Articles (56)

- Bernhard Steffen, Andreas Claßen, Marion Klein, Jens Knoop, Tiziana Margaria. *The Fixpoint Analysis Machine*. In Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95), Springer-V., LNCS 962, 72-87, 1995.
- Munehiro Takimoto, Kenichi Harada. *Effective Partial Redundancy Elimination based on Extended Value Graph*. Information Processing Society of Japan 38(11):2237-2250, 1990.
- Munehiro Takimoto, Kenichi Harada. *Partial Dead Code Elimination Using Extended Value Graph*. In Proceedings of the 6th Static Analysis Symposium (SAS'99), Sprin- ger-V., LNCS 1694, 179-193, 1999.

V Articles (57)

- Robert E. Tarjan. *Fast Algorithms for Solving Path Problems*. Journal of the ACM 28(3):594-614, 1981.
- Jean-Baptiste Tristan, Xavier Leroy. *Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations*. In Conference Record of the 35thAnnual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2008), 17-27, 2008.
- Jean-Baptiste Tristan, Xavier Leroy. *Verified Validation of Lazy Code Motion*. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), ACM SIGPLAN Notices 44:316-326, 2009.

V Articles (58)

M. Mitchell Waldrop. *More than Moore*. Nature 530(7589):144-147, 2016. http://www.nature.com/polopoly_fs/1.19338!/menu/main/topColumns/topLeftColumn/pdf/530144a.pdf

Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. In Conference Record of the 12th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'85), 291-299, 1985.

Mark N. Wegman, F. Kenneth Zadeck. *Constant Propagation with Conditional Constraints*. ACM Transactions on Programming Languages and Systems 13(2):181-210, 1991.

V Articles (59)

- John Whaley, Monica S. Lam. *Cloning-based Context-sensitive Pointer Alias Analysis using Binary Decision Diagrams*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:131-144, 2004.

- Jingling Xue, Qiong Cai. *A Lifetime Optimal Algorithm for Speculative PRE*. ACM Transactions on Architecture and Code Optimization 3(2):115-155, 2006.

- Jingling Xue, Jens Knoop. *A Fresh Look at PRE as a Maximum Flow Problem*. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006), Springer-V., LNCS 3923, 139-154, 2006.

V Articles (60)

- Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn. *Scalable Shape Analysis for Systems Code*. In Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), Springer-V., LNCS 5123, 385-398, 2008.
- Anna Zaks, Amir Pnueli. *CovaC: Compiler Validation by Program Analysis of the Cross-product*. In Proceedings of the 15th International Symposium on Formal Methods (FM 2008), Springer-V., LNCS 5014, 35-51, 2008.
- Jianwen Zhu, Silvian Calman. *Symbolic Pointer Analysis Revisited*. In Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), ACM SIGPLAN Notices 39:145-157, 2004.

V Articles (61)

- Hucheng Zhou, Wenguang Chen, Fred C. Chow. *An SSA-based Algorithm for Optimal Speculative Code Motion under an Execution Profile*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), ACM SIGPLAN Notices 46(6):98-108, 2011.
- Wolf Zimmermann. *On the Correctness of Transformations in Compiler Back-Ends*. In Proceedings of the 1st First International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004), Springer-V., LNCS 4313, 74-95, 2004.
- Lenore Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg. *VOC: A Methodology for Translation Validation of Optimizing Compilers*. Journal of Universal Computer Science 9(3):223-247, 2003.

Appendices

Appendix A
Mathematical Foundations

A.1
Relations

Relations

Let M_i, $1 \leq i \leq k$, be sets.

Definition A.1.1 (k-ary Relation)

A (k-ary) relation is a set R of ordered tuples of elements of M_1, ..., M_k, i.e., $R \subseteq M_1 \times \ldots \times M_k$ is a subset of the cartesian product of the sets M_i, $1 \leq i \leq k$.

Examples

- $M_1 \times \ldots \times M_k$ is the biggest relation on $M_1 \times \ldots \times M_k$.
- $\emptyset$ is the smallest relation on $M_1 \times \ldots \times M_k$.

Binary Relations

Let M, N be sets.

Definition A.1.2 (Binary Relation)

A (binary) relation is a set R of ordered pairs of elements of M and N, i.e., R is a subset of the cartesian product of M and N, $R \subseteq M \times N$, called a relation from M to N.

Examples

- $M \times N$ is the biggest relation from M to N.
- $\emptyset$ is the smallest relation from M to N.

Note

- If R is a relation from M to N, it is common to write $m\,R\,n$, $R(m,n)$, or $R\,m\,n$ instead of $(m,n) \in R$.

Between, On

Definition A.1.3 (Between, On)

A relation R from M to N is also called a relation between M and N or, synonymously, a relation on $M \times N$.

If M equals N, then R is called a relation on M, in symbols: (M, R).

Domain and Range of a Binary Relation

Definition A.1.4 (Domain and Range)

Let R be a relation from M to N.

The sets

- $dom(R) =_{df} \{m \mid \exists\, n \in N.\ (m,n) \in R\}$
- $ran(R) =_{df} \{n \mid \exists\, m \in M.\ (m,n) \in R\}$

are called the domain and the range of R, respectively.

Properties of Relations on a Set M

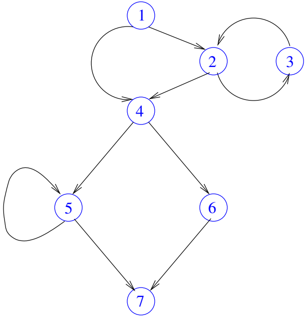
Definition A.1.5 (Properties of Relations on M)

A relation R on a set M is called

- reflexive iff $\forall\, m \in M.\ m\,R\,m$
- irreflexive iff $\forall\, m \in M.\ \neg m\,R\,m$
- transitive iff $\forall\, m, n, p \in M.\ m\,R\,n \ \wedge\ n\,R\,p \ \Rightarrow\ m\,R\,p$
- intransitive iff $\forall\, m, n, p \in M.\ m\,R\,n \wedge n\,R\,p \Rightarrow \neg m\,R\,p$
- symmetric iff $\forall\, m, n \in M.\ m\,R\,n \iff n\,R\,m$
- antisymmetric iff $\forall\, m, n \in M.\ m\,R\,n \wedge n\,R\,m \Rightarrow m = n$
- asymmetric iff $\forall\, m, n \in M.\ m\,R\,n \Rightarrow \neg n\,R\,m$
- linear iff $\forall\, m, n \in M.\ m\,R\,n \vee n\,R\,m \vee m = n$
- total iff $\forall\, m, n \in M.\ m\,R\,n \vee n\,R\,m$

(Anti-) Example

Let $G = (N, E, \mathbf{s} \equiv 1, \mathbf{e} \equiv 7)$ be the below (flow) graph, and let R be the relation '$\cdot$ is linked to $\cdot$ alongside an edge' on N of G (e.g., node 4 is linked to node 6 but not vice versa).



The relation R is not reflexive, not irreflexive, not transitive, not intransive, not symmetric, not antisymmetric, not asymmetric, not linear, and not total.

Equivalence Relation

Let $R \neq \emptyset$ be a relation on M.

Definition A.1.6 (Equivalence Relation)

R is an equivalence relation (or equivalence) iff R is reflexive, transitive, and symmetric.

A.2
Ordered Sets

Ordered Sets

Let $R \neq \emptyset$ be a relation on M.

Definition A.2.1 (Pre-Order)

R is a pre-order (or quasi-order) iff R is reflexive and transitive.

Definition A.2.2 (Partial Order)

R is a partial order iff R is reflexive, transitive, and antisymmetric.

Definition A.2.3 (Strict Partial Order)

R is a strict partial order iff R is asymmetric and transitive.

Examples of Ordered Sets

Pre-order (reflexive, transitive)

- The relation $\Rightarrow$ on logical formulas.

Partial order (reflexive, transitive, antisymmetric)

- The relations $\leq$ and $\geq$ on $\mathbb{N}$.
- The relation $m \mid n$ (m is a divisor of n) on $\mathbb{N}$.

Strict partial order (asymmetric, transitive)

- The relations $<$ and $>$ on $\mathbb{N}$.
- The relations $\subset$ and $\supset$ on sets.

Equivalence relation (reflexive, transitive, symmetric)

- The relation $\iff$ on logical formulas.
- The relation 'have the same prime number divisors' on $\mathbb{N}$.
- The relation 'are citizens of the same country' on people.

Note

- An antisymmetric pre-order is a partial order; a symmetric pre-order is an equivalence relation.
- For convenience and simplicity, also the pair (M, R) is called a pre-order, partial order, and strict partial order, respectively.
- More accurately, we could speak of the pair (M, R) as of a set M which is pre-ordered, partially ordered, and strictly partially ordered by R, respectively.
- Synonymously, we also speak of M as a pre-ordered, partially ordered, and a strictly partially ordered set, respectively, or as of a set M with a pre-order, partial order and strict partial order, respectively.

The Strict Part of an Ordering

Let $\sqsubseteq$ be a pre-order (reflexive, transitive) on $P \neq \emptyset$.

Definition A.2.4 (Strict Part of $\sqsubseteq$)

The relation $\sqsubset$ on P defined by

$$\forall\, p, q \in P.\ p \sqsubset q \Longleftrightarrow_{df} p \sqsubseteq q \ \wedge\ p \neq q$$

is called the strict part of $\sqsubseteq$.

Corollary A.2.5 (Strict Partial Order)

Let $(P, \sqsubseteq)$ be a partial order, let $\sqsubset$ be the strict part of $\sqsubseteq$.

Then: $(P, \sqsubset)$ is a strict partial order.

Useful Results

Let $\sqsubset$ be a strict partial order (asymmetric, transitive) on $P \neq \emptyset$.

Lemma A.2.6

The relation $\sqsubset$ is irreflexive.

Lemma A.2.7

The pair $(P, \sqsubseteq)$, where $\sqsubseteq$ is defined by

$$\forall\, p, q \in P.\ p \sqsubseteq q \iff_{df} p \sqsubset q \ \vee\ p = q$$

is a partial order.

Bounds in Pre-Orders

Definition A.2.8 (Bounds in Pre-Orders)

Let $(Q, \sqsubseteq)$ be a pre-order, let $q \in Q$ and $Q' \subseteq Q$.

q is called a

- lower bound of Q', in symbols: $q \sqsubseteq Q'$, if $\forall\, q' \in Q'.\ q \sqsubseteq q'$
- upper bound of Q', in symbols: $Q' \sqsubseteq q$, if $\forall\, q' \in Q'.\ q' \sqsubseteq q$
- greatest lower bound (glb) of Q', if q is a lower bound of Q' and for every other lower bound $\hat{q}$ of Q' holds: $\hat{q} \sqsubseteq q$
- least upper bound (lub) of Q', if q is an upper bound of Q' and for every other upper bound $\hat{q}$ of Q' holds: $q \sqsubseteq \hat{q}$

Extremal Elements in Pre-Orders

Definition A.2.9 (Extremal Elements in Pre-Orders)

Let $(Q, \sqsubseteq)$ be a pre-order, let $\sqsubset$ be the strict part of $\sqsubseteq$, and let $q \in Q$ and $Q' \subseteq Q$.

q is called a

- minimal element of Q', if there is no element $q' \in Q'$ with $q' \sqsubset q$.
- maximal element of Q', if there is no element $q' \in Q'$ with $q \sqsubset q'$.
- least element of Q', if $q \sqsubseteq Q'$
- greatest element of Q', if $Q' \sqsubseteq q$

Existence and Uniqueness in Partial Orders

...of bounds and extremal elements in partially ordered sets.

Let $(P, \sqsubseteq)$ be a partial order.

Lemma A.2.10 (Unique if Existent)

Least upper bounds, greatest lower bounds, least elements, and greatest elements in P are unique, if they exist.

Lemma A.2.11 (Not Unique if Existent)

Minimal and maximal elements in a subset $Q \subseteq P$ are not necessarily unique if they exist.

Note: For pre-orders the uniqueness results of Lemma A.2.10 do no hold.

Suprema and Infima in Partial Orders

Given the existence (and thus their uniqueness) in partial orders, the

- lub and the glb of a set $P' \subseteq P$ are also called the supremum and the infimum of P', and are usually denoted by $\bigsqcup P'$ and $\bigsqcap P'$, respectively.
- least element and the greatest element of P are usually denoted by $\perp$ and $\top$, respectively.

Chains

Let $(P, \sqsubseteq)$ be a partial order.

Definition A.2.12 (Chain)

A subset $\emptyset \neq C \subseteq P$ is called a chain, if the elements of C are totally ordered.

Definition A.2.13 (Ascending, Descending Chain)

Let $C \subseteq P$ be a chain. Then: C given as

- $C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \ldots\}$
- $C = \{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \ldots\}$

is called an ascending chain and descending chain, respectively.

Definition A.2.14 (Finite, Infinite Chain)

Let $C \subseteq P$ be a chain. C is called finite, if the number of its elements is finite; otherwise, C is called infinite.

Examples of Chains

- The set $M =_{df} \{n \in \mathbb{N} \mid n \ \mathit{even}\}$ is a chain in $\mathbb{N}$.
- The set $M =_{df} \{z \in \mathbb{Z} \mid z \ \mathit{odd}\}$ is a chain in $\mathbb{Z}$.
- The set $M =_{df} \{\, \{k \in \mathbb{N} \mid k < n\} \mid n \in \mathbb{N}\}$ is a chain in the powerset $\mathcal{P}(\mathbb{N})$ of $\mathbb{N}$.

Note: A chain can always be given in the form of an ascending or descending chain.

- $\{0 \leq 2 \leq 4 \leq 6 \leq \ldots\}$: ascending chain in $\mathbb{N}$.
- $\{\ldots \geq 6 \geq 4 \geq 2 \geq 0\}$: descending chain in $\mathbb{N}$.
- $\{\ldots \leq -3 \leq -1 \leq 1 \leq 3 \leq \ldots\}$: ascending chain in $\mathbb{Z}$.
- $\{\ldots \geq 3 \geq 1 \geq -1 \geq -3 \geq \ldots\}$: descending chain in $\mathbb{Z}$.
- ...

Directed Sets

Let $(P, \sqsubseteq)$ be a partial order, and let $D \subseteq P$.

Definition A.2.15 (Directed Set)

D is called a directed set (in German: gerichtete Menge), if every finite subset $D' \subseteq D$ has a supremum in D, i.e.,
$\bigsqcup D'$ *exists* $= d \in D$.

Useful Results

Let $(P, \sqsubseteq)$ be a partial order, and let $C, D \subseteq P$.

Lemma A.2.16 (Non-Emptyness of Directed Sets)

Let D be a directed set. Then: $D \neq \emptyset$.

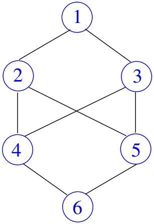
Proof. We have: $\emptyset \subseteq D$. Since D is a directed set, the supremum of $\emptyset$ exists in D, i.e., $\bigsqcup \emptyset$ *exists* $= \perp \in D$, by definition. Thus, we have: $\perp \in D$, and therefore $D \neq \emptyset$. □

Lemma A.2.17 (Chains are Directed Sets)

Let C be a non-empty chain. Then: C is a directed set.

Hasse Diagrams

...are an economic graphical representation of partial orders.



The links of a Hasse diagram

- are read from below to above (lower means smaller).
- represent the relation R of '$\cdot$ is an immediate predecessor of $\cdot$' defined by
  $p \, R \, q \iff_{df} \; p \sqsubset q \wedge \nexists \, r \in P. \; p \sqsubset r \sqsubset q$
  of a partial order $(P, \sqsubseteq)$, where $\sqsubset$ is the strict part of $\sqsubseteq$.

Reading Hasse Diagrams

The Hasse diagram representation of a partial order

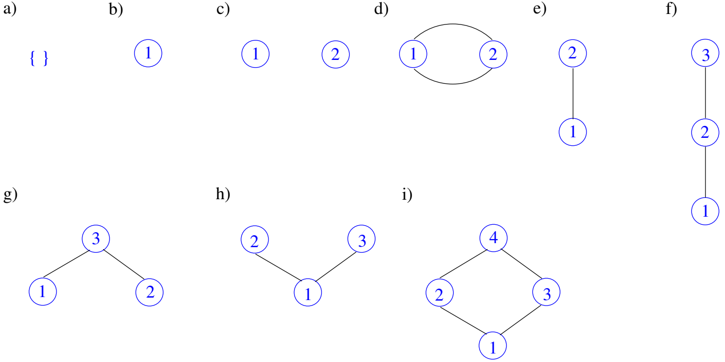
- omits to explicitly represent reflexive and transitive links
- focuses on the 'immediate predecessor' relation.

This focused representation of a Hasse diagram

- is economical (in the number of links)
- while preserving all relevant information of the represented partial order:
  - $p \sqsubseteq q \land p = q$: explicitly represented (though without an explicit link)
  - $p \sqsubseteq q \land p \neq q$: holds, if there is an ascending path (with at least one element) from p to q.

Exercise

Which of the below diagrams are Hasse diagrams of partial orders? Which ones are directed sets? Which of their subsets are directed sets?



Monotonic and Inflationary Functions on POs

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be partial orders (POs), let $f : C \to D$ be a function from C to D, let $g : C \to C$ be a function on C, and let $\hat{c} \in C$ be an element of C.

Definition A.2.18 (Monotonic Functions on POs)

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)

Definition A.2.19 (Inflationary Functions on POs)

g is called

- inflationary for $\hat{c}$ iff $\hat{c} \sqsubseteq g(\hat{c})$
- inflationary iff $\forall\, c \in C.\; c \sqsubseteq g(c)$

A.3
Complete Partially Ordered Sets

Complete Partially Ordered Sets

...or Complete Partial Orders:

- a slightly weaker notion than that of a lattice (cf. Appendix A.4), which is sufficient for the modelling of many problems in computer science, and more adequate if full lattice properties are not required.
- come in different variants as so-called
  - Chain Complete Partial Orders (CCPOs)
  - Directed Complete Partial Orders (DCPOs)

  based on the notions of chains and directed sets, respectively.

Complete Partial Orders: CCPOs and DCPOs

Let $(P, \sqsubseteq)$ be a partial order.

Definition A.3.1 (Chain Complete Partial Order)

$(P, \sqsubseteq)$ is a chain complete partial order (CCPO), if every (ascending) chain $C \subseteq P$ has a least upper bound $\bigsqcup C$ in P, i.e., $\bigsqcup C$ *exists* $\in P$.

Definition A.3.2 (Directed Complete Partial Order)

A partial order $(P, \sqsubseteq)$ is a directed complete partial order (DCPO), if every directed subset $D \subseteq P$ has a least upper bound $\bigsqcup D$ in P, i.e., $\bigsqcup D$ *exists* $\in P$.

Remarks about CCPOs and DCPOs

About CCPOs

- A CCPO is often called a domain.
- 'Ascending chain' and 'chain' can equivalently be used in Definition A.3.1, since a chain can always be given in ascending order. 'Ascending chain' is just more intuitive.

About DCPOs

- A directed set S, in which by definition every finite subset has a supremum in S, does not need to have a supremum itself in S, if S is infinite. Therefore, the DCPO property does not trivially follow from the directed set property.
- $(P, \sqsubseteq) =_{df} (\emptyset, \emptyset)$ is a DCPO. (Note that the DCPO property holds trivially, since $\emptyset$ as the only subset of $\emptyset$ is not a directed set (cf. Lemma A.2.16). Note also that $P = \emptyset$ implies $\sqsubseteq \; = \emptyset \subseteq P \times P$.)

Existence of Least Elements in CCPOs

Lemma A.3.3 (Existence of a Least Element)

Let $(C, \sqsubseteq)$ be a CCPO. Then there is a least element in C, denoted by $\perp$, which is given by the supremum of the empty chain: $\perp = \bigsqcup \emptyset$.

Corollary A.3.4 (Non-Emptyness)

Let $(C, \sqsubseteq)$ be a CCPO. Then: $C \neq \emptyset$.

Note: Lemma A.3.3 does not hold for DCPOs, i.e., if $(D, \sqsubseteq)$ is a DCPO, there does not need to be a least element in D.

Relating DCPOs and CCPOs

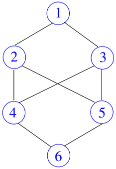
Lemma A.3.5 (Relating DCPOs and CCPOs)

Let $(D, \sqsubseteq)$ be a DCPO. Then:

$(D, \sqsubseteq)$ is a CCPO, if D contains a least element.

Examples of CCPOs

- $(\mathcal{P}(\mathbb{N}), \subseteq)$ is a CCPO (and a DCPO).
  - Least element: $\emptyset$
  - Least upper bound $\bigsqcup C$ of C *chain* $\subseteq \mathcal{P}(\mathbb{N})$: $\bigcup_{C' \in C} C'$
- The partial order P given by the below graph (Hasse diagram) is a CCPO (but not a DCPO).



- The set of finite and infinite strings S partially ordered by the prefix relation $\sqsubseteq_{pfx}$ defined by

  $$\forall\, s, s' \in S.\ s \sqsubseteq_{pfx} s' \iff_{df} \exists\, s'' \in S.\ s +\!\!+ s'' = s'$$

  is a CCPO, i.e., $(S, \sqsubseteq_{pfx})$ is a CCPO (and a DCPO).

Examples of DCPOs

- $(\emptyset, \emptyset)$ is a DCPO (but not a CCPO).
- $(\{-n \mid n \in \mathbb{N}\}, \leq)$ is a DCPO (but not a CCPO).
- The set of finite and infinite strings S partially ordered by the lexicographical order $\sqsubseteq_{lex}$ defined by

  $$\forall\, s, t \in S.\ s \sqsubseteq_{lex} t \Longleftrightarrow_{df}$$
  $$\exists\, p, s', t' \in S.\ s = p +\!\!+ s' \ \wedge\ t = p +\!\!+ t' \ \wedge$$
  $$(s' = \varepsilon \ \vee\ s'_1 < t'_1)$$

  where ε denotes the empty string, w_1 denotes the first character of a string w, and $<$ the lexicographical ordering on characters, is a DCPO, i.e., $(S, \sqsubseteq_{lex})$ is a DCPO (and a CCPO).

(Anti-) Examples of CCPOs and DCPOs

- $(\mathbb{N}, \leq)$ is neither a CCPO nor a DCPO.
- The set of finite strings S_{fin} partially ordered by the
  - prefix relation $\sqsubseteq_{pfx}$ defined by

    $$\forall\, s, s' \in S_{fin}.\ s \sqsubseteq_{pfx} s' \iff_{df} \exists\, s'' \in S_{fin}.\ s +\!\!+ s'' = s'$$

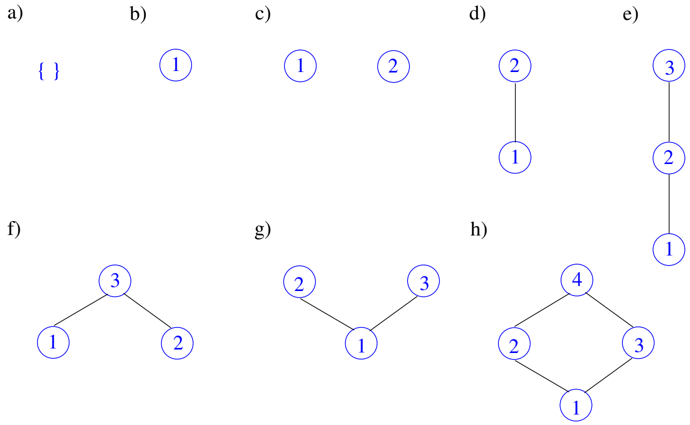
    neither a CCPO nor a DCPO.
  - lexicographical order $\sqsubseteq_{lex}$ defined by

    $$\forall\, s, t \in S_{fin}.\ s \sqsubseteq_{lex} t \iff_{df}$$
    $$\exists\, p, s', t' \in S_{fin}.\ s = p +\!\!+ s' \ \wedge\ t = p +\!\!+ t' \ \wedge$$
    $$(s' = \varepsilon \ \vee\ s'_1 < t'_1)$$

    where ε denotes the empty string, w_1 denotes the first character of a string w, and $<$ the lexicographical ordering on characters, is neither a CCPO nor a DCPO.
- $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is neither a CCPO nor a DCPO.

Exercise

Which of the partial orders given by the below Hasse diagrams are CCPOs? Which ones are DCPOs?



Continuous Functions on CCPOs

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be CCPOs, and let $f : C \to D$ be a function from C to D.

Definition A.3.6 (Continuous Functions on CCPOs)

f is called continuous iff

$$\forall\, C' \neq \emptyset \; chain \subseteq C.\; f(\bigsqcup_C C') \;=_D\; \bigsqcup_D f(C')$$

(Preservation of least upper bounds)

Note: $\forall\, C' \subseteq C.\; f(C') =_{df} \{\, f(c) \,|\, c \in C' \}$

Continuous Functions on DCPOs

Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be DCPOs, and let $f : D \to E$ be a function from D to E.

Definition A.3.7 (Continuous Functions on DCPOs)

f is called continuous iff

$$\forall\, D' \neq \emptyset \textit{ directed set} \subseteq D.\ f(\bigsqcup_D D') =_E \bigsqcup_E f(D')$$

(Preservation of least upper bounds)

Note: $\forall\, D' \subseteq D.\ f(D') =_{df} \{\, f(d) \mid d \in D' \}$

Useful Results

Let $(C, \sqsubseteq_C), (D, \sqsubseteq_D)$ be CCPOs, let $(E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be DCPOs.

Lemma A.3.8 (Characterizing Monotonicity)

1. $f : C \to D$ is monotonic
   iff $\forall\, C' \neq \emptyset$ *chain* $\subseteq C.\ f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$
2. $g : E \to F$ is monotonic
   iff $\forall\, E' \neq \emptyset$ *directed set* $\subseteq E.\ g(\bigsqcup_E E') \sqsupseteq_F \bigsqcup_F g(E')$

Corollary A.3.9

f and g are monotonic, if f and g are continuous, respectively (i.e., continuity implies monotonicity.).

Strict Functions on CCPOs and DCPOs

Let $(C, \sqsubseteq_C), (D, \sqsubseteq_D)$ be CCPOs with least elements $\perp_C$ and $\perp_D$, respectively, let $(E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be DCPOs with least elements $\perp_E$ and $\perp_F$, respectively, and let $f : C \to D$ and $g : E \to F$ be continuous functions.

Definition A.3.10 (Strict Functions)

The functions f and g are called strict, if the equalities

- $f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$
- $g(\bigsqcup_E E') =_F \bigsqcup_F g(E')$

also hold for $C' = \emptyset$ and $E' = \emptyset$, respectively, i.e., if

- $f(\bigsqcup_C \emptyset) =_C \; f(\perp_C) =_D \perp_D \; =_D \bigsqcup \emptyset$
- $f(\bigsqcup_E \emptyset) =_E \; g(\perp_E) =_F \perp_F \; =_F \bigsqcup \emptyset$

holds for f and g, respectively.

Common CCPO and DCPO Constructions (1)

Most of the following construction principles hold for

- CCPOs
- DCPOs

In these cases, we simply write CPO.

Common CPO Constructions: Flat CPOs (2)

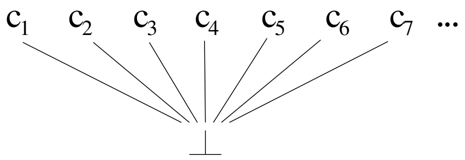
Lemma A.3.11 (Flat CPO Construction)

Let C be a set.

Then $(C \,\dot\cup\, \{\perp\}, \sqsubseteq_{flat})$, where $\sqsubseteq_{flat}$ is defined by

$$\forall\, c, d \in C \,\dot\cup\, \{\perp\}.\ c \sqsubseteq_{flat} d \Leftrightarrow c = \perp\ \vee\ c = d$$

is a CPO, a so-called flat CPO.



Common CPO Constructions: Flat DCPOs (3)

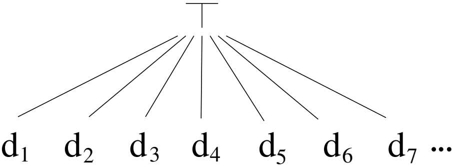
Lemma A.3.12 (Flat DCPO Construction)

Let D be a set.

Then $(D \,\dot{\cup}\, \{\top\}, \sqsubseteq_{flat})$, where $\sqsubseteq_{flat}$ is defined by

$$\forall\, d, e \in D \,\dot{\cup}\, \{\top\}.\; d \sqsubseteq_{flat} e \Leftrightarrow e = \top \;\vee\; d = e$$

is a DCPO, a so-called flat DCPO.



Common CPO Constructions: Products (3)

Lemma A.3.13 (Product Construction(s))

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs.

Then the

- non-strict product
  - $(\bigtimes P_i, \sqsubseteq_\times) = (P_1 \times P_2 \times \ldots \times P_n, \sqsubseteq_\times)$, where $\sqsubseteq_\times$ is defined by: $\forall\, (p_1, p_2, \ldots, p_n), (q_1, q_2, \ldots, q_n) \in \bigtimes P_i.$
  $$(p_1, p_2, \ldots, p_n) \sqsubseteq_\times (q_1, q_2, \ldots, q_n) \;\Leftrightarrow\; \forall\, i \in \{1, \ldots, n\}.\; p_i \sqsubseteq_i q_i$$
- strict product or smash product
  - $(\bigotimes P_i, \sqsubseteq_\otimes) = (P_1 \otimes P_2 \otimes \ldots \otimes P_n, \sqsubseteq_\otimes)$, where $\sqsubseteq_\otimes$ is defined as $\sqsubseteq_\times$ with the additional constraint:
  $$(p_1, p_2, \ldots, p_n) = \bot \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\; p_i = \bot_i$$

are CPOs.

Common CPO Constructions: Sums (4)

Lemma A.3.14 (Sum Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs.

Then the direct sum

- $(\bigoplus P_i, \sqsubseteq_\oplus) = (P_1 \dot\cup P_2 \dot\cup \ldots \dot\cup P_n, \sqsubseteq_\oplus)$, where $\bigoplus P_i$ is defined as the disjoint union of all P_i, $i \in \{1, \ldots, n\}$, and $\sqsubseteq_\oplus$ is defined by

  $$\forall\, p, q \in \bigoplus P_i.\ p \sqsubseteq_\oplus q \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\ p, q \in P_i \ \wedge\ p \sqsubseteq_i q$$

is a CPO.

Note: The least elements of $(P_i, \sqsubseteq_i)$, $i \in \{1, \ldots, n\}$, are usually identified, i.e., $\perp =_{df} \perp_i$, $i \in \{1, \ldots, n\}$.

Common CPO Constructions: Functions (5)

Lemma A.3.15 (Function-space Construction)

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be CPOs, and let $[C \to D] =_{df} \{f : C \to D \mid f \text{ continuous}\}$ be the set of continuous functions from C to D.

Then the continuous function space

- $([C \to D], \sqsubseteq_{cfs})$, where $\sqsubseteq_{cfs}$ is defined by: $\forall f, g \in [C \to D].\ f \sqsubseteq_{cfs} g \Longleftrightarrow \forall c \in C.\ f(c) \sqsubseteq_D g(c)$

is a CPO.

Note: The definition of $\sqsubseteq_{cfs}$ does not require C to be a CPO.

A.4
Lattices

Lattices and Complete Lattices

Let $(P, \sqsubseteq)$ be a partial order.

Definition A.4.1 (Lattice)

$(P, \sqsubseteq)$ is a lattice, if every nonempty finite subset P' of P has a least upper bound and a greatest lower bound in P.

Definition A.4.2 (Complete Lattice)

$(P, \sqsubseteq)$ is a complete lattice, if every subset P' of P has a least upper bound and a greatest lower bound in P.

Note: (Complete) lattices are special partial orders.

Properties of Complete Lattices

Lemma A.4.3 (Existence of Extremal Elements)

Let $(P, \sqsubseteq)$ be a complete lattice. Then there is

1. a least element $\bot$ in P satisfying: $\bot = \bigsqcup \emptyset = \bigsqcap P$.
2. a greatest element $\top$ in P satisfying: $\top = \bigsqcap \emptyset = \bigsqcup P$.

Lemma A.4.4 (Characterization Lemma)

Let $(P, \sqsubseteq)$ be a partial order. Then the following statements are equivalent:

1. $(P, \sqsubseteq)$ is a complete lattice.
2. Every subset of P has a least upper bound.
3. Every subset of P has a greatest lower bound.

Relating Lattices and Complete Partial Orders

Lemma A.4.5 (Lattices and CCPOs, DCPOs)

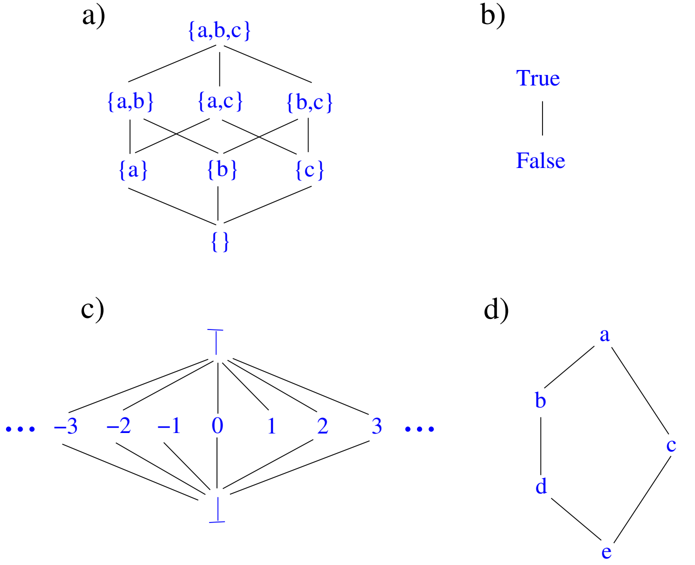
Let $(P, \sqsubseteq)$ be a complete lattice.

Then: $(P, \sqsubseteq)$ is a

- CCPO
- DCPO

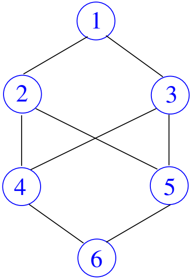
Note: Lemma A.4.5 does not hold for lattices.

Examples of Complete Lattices



(Anti-) Examples

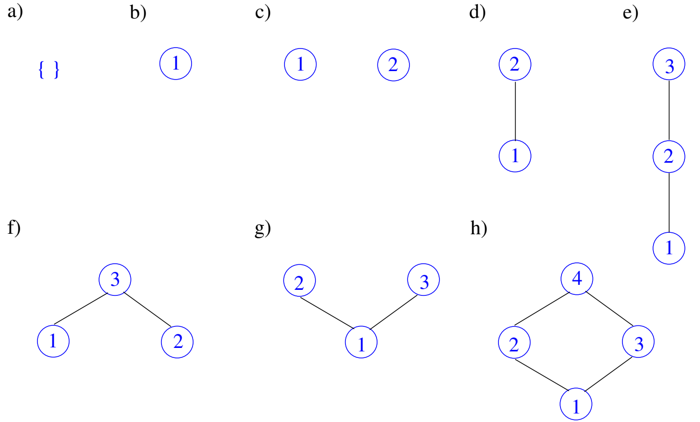
- The partial order $(P, \sqsubseteq)$ given by the below Hasse diagram is not a lattice (though it is a CCPO but not a DCPO).



- $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is not a complete lattice (and not a CCPO and not a DCPO).

Exercise

Which of the partial orders given by the below Hasse diagrams are lattices? Which ones are complete lattices?



Descending, Ascending Chain Condition

Let $(P, \sqsubseteq)$ be a lattice.

Definition A.4.6 (Chain Condition)

P satisfies the

1. descending chain condition, if every descending chain gets stationary, i.e., for every chain $p_1 \sqsupseteq p_2 \sqsupseteq \ldots \sqsupseteq p_n \sqsupseteq \ldots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.
2. ascending chain condition, if every ascending chain gets stationary, i.e., for every chain $p_1 \sqsubseteq p_2 \sqsubseteq \ldots \sqsubseteq p_n \sqsubseteq \ldots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.

Distributive and Additive Functions on Latties

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f : P \to P$ be a function on P.

Definition A.4.7 (Distributive, Additive Function)

f is called

- distributive iff $\forall\, P' \subseteq P.\;\; f(\bigsqcap P') = \bigsqcap \{f(p) \mid p \in P'\}$
  (Preservation of greatest lower bounds)
- additive iff $\forall\, P' \subseteq P.\;\; f(\bigsqcup P') = \bigsqcup \{f(p) \mid p \in P'\}$
  (Preservation of least upper bounds)

Characterizing Monotonicity

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

Lemma A.4.8

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f : P \to P$ be a function on P. Then:

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

Useful Results

Let $(P, \sqsubseteq)$ be a complete lattice, and let $f : P \to P$ be a function on P.

Lemma A.4.9

f is distributive iff f is additive.

Lemma A.4.10

f is monotonic if f is distributive (additive)
(i.e., f is distributive (additive) implies f is monotonic)

Lattice Constructions: Flat Lattices (1)

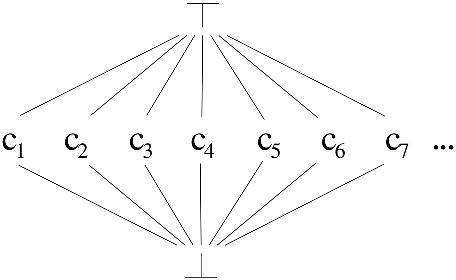
Lemma A.4.11 (Flat Construction)

Let C be a set.

Then $(C \,\dot{\cup}\, \{\bot, \top\}, \sqsubseteq_{flat})$, where $\sqsubseteq_{flat}$ is defined by

$$\forall\, c, d \in C \,\dot{\cup}\, \{\bot, \top\}.\ c \sqsubseteq_{flat} d \Leftrightarrow c = \bot \ \vee\ c = d \ \vee\ d = \top$$

is a complete lattice, a so-called flat lattice.



Lattice Constructions: Products, Sums,... (2)

Analogously to CPOs, the construction principles for

- non-strict products
- strict products
- direct sums
- continuous (here: additive, distributive) function spaces

carry over from CPOs to lattices and complete lattices
(cf. Appendix A.3).

A.5
Fixed Point Theorems

Fixed Points, Least and Greatest Fixed Points

Let $(C, \sqsubseteq)$ be a CCPO, let $f : C \to C$ be a function on C, and let $c \in C$ be an element of C.

Definition A.5.1 (Fixed Point)

c is called a fixed point of f iff $f(c) = c$.

Definition A.5.2 (Least, Greatest Fixed Point)

Let c be a fixed point of f. Then c is the

- least fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow c \sqsubseteq d$
- greatest fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow d \sqsubseteq c$

The least fixed point of f and the greatest fixed point of f are usually denoted by μf and νf, respectively.

Fixed Points Theorem for Monotonic Functions

Theorem A.5.3 (Fixed Point Theorem)

Let $(C, \sqsubseteq)$ be a CCPO, and let $f : C \to C$ be a monotonic function on C.

Then f has a unique least fixed point μf.

Note

- Theorem A.5.3 ensures the existence of a unique least fixed point for a monotonic function but it does not provide a constructive procedure for computing it.
- This is in contrast to continuous functions on CCPOs (cf. Theorem A.5.4). In practice, thus, continuous functions instead of monotonic ones are considered.

Fixed Point Theorem (Knaster/Tarski, Kleene)

Theorem A.5.4 (Knaster/Tarski, Kleene)

Let $(C, \sqsubseteq)$ be a CCPO, 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 $\{\perp, f(\perp), f^2(\perp), \ldots\}$, the so-called Kleene chain, i.e.

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{\perp, f(\perp), f^2(\perp), \ldots\}$$

Note: $f^0 =_{df} Id_C$; $f^i =_{df} f \circ f^{i-1}$, $i > 0$.

Proof of Fixed Point Theorem A.5.4 (1)

We have to prove:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) = \bigsqcup \{ f^i(\bot) \mid i \geq 0 \}$$

1. exists,
2. is a fixed point of f,
3. is the least fixed point of f.

Proof of Fixed Point Theorem A.5.4 (2)

1. Existence

- By definition of $\perp$ as the least element of C and of f^0 as the identity on C we have: $\perp = f^0(\perp) \sqsubseteq f^1(\perp) = f(\perp)$.
- Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
  $\forall\, i, j \in \mathbb{N}_0.\ i < j \Rightarrow f^i(\perp) \sqsubseteq f^{i+1}(\perp) \sqsubseteq f^j(\perp)$.
- Hence, the set $\{f^i(\perp) \mid i \geq 0\}$ is a (possibly infinite) chain in C.
- Since $(C, \sqsubseteq)$ is a CCPO and $\{f^i(\perp) \mid i \geq 0\}$ a chain in C, this implies by definition of a CPO that the least upper bound of the chain $\{f^i(\perp) \mid i \geq 0\}$

$$\bigsqcup \{f^i(\perp) \mid i \geq 0\} = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \text{ exists.}$$

Proof of Fixed Point Theorem A.5.4 (3)

2. Fixed point property

$$
\begin{array}{rcl}
 & & f(\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)) \\
(f \text{ continuous}) & = & \bigsqcup_{i \in \mathbb{N}_0} f(f^i(\bot)) \\
 & = & \bigsqcup_{i \in \mathbb{N}_1} f^i(\bot) \\
(C' =_{df} \{f^i \bot \mid i \geq 1\} \text{ is a chain } \Rightarrow & & \\
\bigsqcup C' \text{ exists } = \bot \sqcup \bigsqcup C') & = & \bot \sqcup \bigsqcup_{i \in \mathbb{N}_1} f^i(\bot) \\
(f^0(\bot) =_{df} \bot) & = & \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)
\end{array}
$$

Proof of Fixed Point Theorem A.5.4 (4)

3. Least fixed point property

- Let c be an arbitrary fixed point of f. Then: $\perp \sqsubseteq c$.
- Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
  $\forall\, i \in \mathbb{N}_0.\ f^i(\perp) \sqsubseteq f^i(c)\ (= c)$.
- Since c is a fixed point of f, this implies:
  $\forall\, i \in \mathbb{N}_0.\ f^i(\perp) \sqsubseteq c\ (= f^i(c))$.
- Thus, c is an upper bound of the set $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$.
- Since $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$ is a chain, and $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is by definition the least upper bound of this chain, we obtain the desired inclusion
  $$\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c.$$

□

Least Conditional Fixed Points

Let $(C, \sqsubseteq)$ be a CCPO, let $f : C \to C$ be a function on C, and let $d, c_d \in C$ be elements of C.

Definition A.5.5 (Least Conditional Fixed Point)

c_d is called the

- least conditional fixed point of f wrt d (in German: kleinster bedingter Fixpunkt) iff c_d is the least fixed point of C with $d \sqsubseteq c_d$, i.e.,
  $\forall x \in C.\ d \sqsubseteq x \land f(x) = x \Rightarrow c_d \sqsubseteq x$.

Conditional Fixed Point Theorem

Theorem A.5.6 (Conditional Fixed Point Theorem)

Let $(C, \sqsubseteq)$ be a CCPO, let $d \in C$, and let $f : C \to C$ be a continuous function on C which is inflationary for d, i.e., $d \sqsubseteq f(d)$.

Then f has a least conditional fixed point μf_d, which equals the least upper bound of the (possibly infinite) (generalized) Kleene chain $\{d, f(d), f^2(d), \ldots\}$, i.e.

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \ldots\}$$

Finite Fixed Point Theorems

Let $(C, \sqsubseteq)$ be a CCPO, let $d \in C$, and let $f : C \to C$ be a monotonic function on C.

Theorem A.5.7 (Finite Fixed Point Theorem)

If two succeeding elements in the Kleene chain of f are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(\perp) = f^{i+1}(\perp)$, then we have: $\mu f = f^i(\perp)$.

Theorem A.5.8 (Finite Conditional FP Theorem)

If f is inflationary for d, i.e., $d \sqsubseteq f(d)$, and two succeeding elements in the (generalized) Kleene chain of f wrt d are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(d) = f^{i+1}(d)$, then we have: $\mu f_d = f^i(d)$.

Note: Continuity of f is not required for Theorems A.5.7 and A.5.8. Monotonicity (and inflationarity) of f suffice(s).

Towards the Existence of Finite Fixed Points

Let $(P, \sqsubseteq)$ be a partial order, and let $p, r \in P$.

Definition A.5.9 (Chain-finite Partial Order)

$(P, \sqsubseteq)$ is called

- chain-finite (in German: kettenendlich) iff P does not contain an infinite chain.

Definition A.5.10 (Finite Element)

p is called

- finite iff the set $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ does not contain an infinite chain.
- finite relative to r iff the set $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ does not contain an infinite chain.

Existence of Finite Fixed Points

There are numerous conditions, which are sufficient to ensure the existence of the least finite fixed point of a function f and often hold in practice (cf. Nielson/Nielson 1992), e.g.

- the domain or the range of f are finite or chain-finite,
- the least fixed point of f is finite,
- f is of the form $f(c) = c \sqcup g(c)$, where g is a monotonic function on a chain-finite (data) domain.

Fixed Point Theorems, DCPOs, and Lattices

Last but not least:

DCPOs with a least element (cf. Lemma A.3.5) and complete lattices (cf. Lemma A.4.5) are CCPOs, too. Thus, we have:

Corollary A.5.11 (Fixed Points, DCPOs, Lattices)

The fixed point theorem results of Chapter A.5 hold for functions on DCPOs with a least element and on complete lattices, too.

A.6
References, Further Reading

Further Reading for Appendix A (1)

- André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
- Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012. (Kapitel 1, Ordnungen und Verbände; Kapitel 2.4, Vollständige Verbände; Kapitel 3, Fixpunkttheorie mit Anwendungen)
- Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013. (Kapitel 2, Verbände und Ordnungen; Kapitel 3.4, Vollständige Verbände; Kapitel 4, Fixpunkttheorie mit Anwendungen)
- Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.

Further Reading for Appendix A (2)

- Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002. (Chapter 1, Ordered Sets; Chapter 2, Lattices and Complete Lattices; Chapter 8, CPOs and Fixpoint Theorems)
- Marcel Erné. *Einführung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
- Helmuth Gericke. *Theorie der Verbände*. Bibliographisches Institut, 2. Auflage, 1967.
- George Grätzer. *General Lattice Theory*. Birkhäuser, 2nd edition, 2003. (Chapter 1, First Concepts; Chapter 2, Distributive Lattices; Chapter 5, Varieties of Lattices)

Further Reading for Appendix A (3)

- Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001. (Chapter 6, Ordered Pairs; Chapter 7, Relations; Chapter 8, Functions)
- Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.
- Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009. (Chapter 3, Functions, Sequences, and Relations)
- Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998. (Chapter 4, Functions; Chapter 6, Relations)

Further Reading for Appendix A (4)

- David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008. (Chapter 1, Collecting Things Together: Sets; Chapter 2, Comparing Things: Relations)
- Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92), 96-108, 1992.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 4, Denotational Semantics)
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 5, Denotational Semantics)

Further Reading for Appendix A (5)

- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Appendix A, Partially Ordered Sets)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmie- rung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 10, Beispiel: Berechnung von Fixpunkten; Kapitel 10.2, Ein bisschen Mathematik: CPOs und Fixpunkte)

Appendix B
Pragmatics of Flow Graph Representations

B.1
Background and Motivation

B.1.1
Flow Graph Variants

Representing Instructions in Flow Graphs

...given a flow graph, program instructions (assignments, tests) can be represented by

- nodes
- edges

where nodes and edges can be labelled by

- single instructions
- basic blocks.

Flow Graph Variants

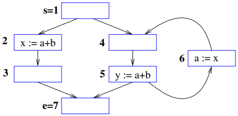
In total, this leads to four flow graph variants:

- Node-labelled flow graphs
  (in the style of Kripke structures)
  - Single instruction graphs (SI graphs)
  - Basic block graphs (BB graphs)
- Edge-labelled flow graphs
  (in the style of transition systems)
  - Single instruction graphs (SI graphs)
  - Basic block graphs (BB graphs)

Illustration: Node-labelled Flow Graphs

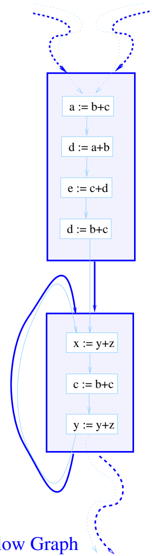
Single instruction vs. basic block flow graphs:

a)



Node–labelled SI Flow Graph

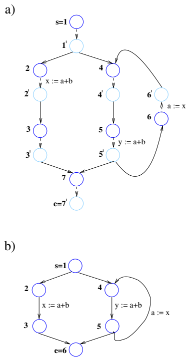
b)



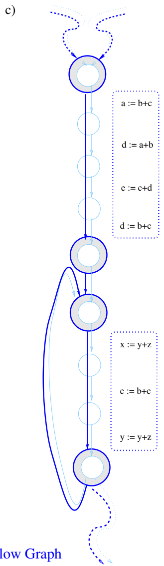
Node–labelled BB Flow Graph

Illustration: Edge-labelled Flow Graphs

Single instruction vs. basic block flow graphs:



Edge−labelled SI Flow Graphs



Edge−labelled BB Flow Graph

Which Flow Graph Representation to Choose?

Conceptually, there is

- no difference between the various flow graph variants making the choice of a particular one essentially a matter of taste.

Pragmatically, however,

- the flow graph variants differ in the ease and hence adequacy of use for specifying and implementing program analyses and optimizations.

This will be considered in detail next.

B.1.2

Flow Graph Variants: Which one to Choose?

Basic Block or Single Instruction Graphs

...this is the question.

In the following we will investigate and compare the adequacy of different flow graph representations.

To this end we will consider node and edge-labelled flow graphs with basic blocks and single instructions and investigate their pragmatic

- advantages and disadvantages for program analysis

...addressing thereby especially the question:

- Basic block or single instruction graphs: Just a matter of taste?

On the fly we will learn

- Faint variable analysis as an example of a non-separable real world data flow analysis problem.

Basic Block Graphs: Supposed Advantages

Advantages of basic block graphs commonly attributed to them in the perception of "folk knowledge":

Basic block graphs are smaller than

- single instruction graphs containing less nodes

...which leads to better scalability of program analyses because

- less nodes (edges) are involved in the (potentially) computationally costly fixed point iteration
- larger programs fit into the main memory.

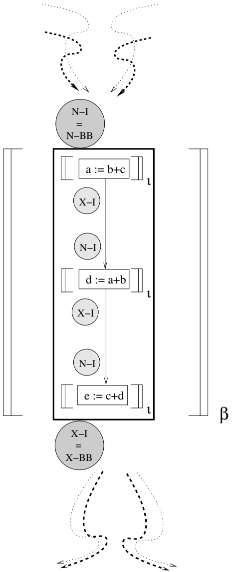
Basic Block Graphs: Definite Disadvantages

Definite disadvantages of basic blocks in applications:

- Higher conceptual complexity: Basic blocks introduce an undesired hierarchy into flow graphs making both theoretical reasoning and practical implementations more difficult.
- Need for pre- and post-processes: These are usually required in order to cope with the additional problems introduced by the hierarchical structure of basic block flow graphs (e.g., in dead code elimination, constant propagation,...); or which necessitate "tricky" formulations to avoid them (e.g., in partial redundancy elimination).
- Limited generality: Some practically relevant program analyses and optimizations are difficult or not at all expressible on the level of basic block flow graphs (e.g., faint variable elimination).

Illustrating the Hierarchy Due to Basic Blocks

...for node-labelled and edge-labelled flow graphs:



In the following

...we investigate and compare

- advantages and disadvantages of basic block (BB) flow graphs and single instructions (SI) flow graphs

by means of DFA problems

- we already considered
  - Available expressions
  - Simple constants
- or consider afresh
  - Faint variables

B.2

MOP and *MaxFP* Approach for Selected Flow Graph Variants

B.2.1
MOP and *MaxFP* Approach for Edge-labelled SI Flow Graphs

Fixing the Setting

Let

- $G = (N, E, \mathbf{s}, \mathbf{e})$ be an edge-labelled SI flow graph
- $\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]_{E,\iota}, c_{\mathbf{s}}, fw)$ be a DFA specification.

The *MOP* Approach and *MOP* Solution

...for an edge-labelled single instruction flow graph.

Definition B.2.1.1 (The *MOP* Solution)

The $MOP_{E,\iota}$ solution of $\mathcal{S}_G$ is defined by:

$$MOP_{E,\iota}^{\mathcal{S}_G} : N \to \mathcal{C}$$

$$\forall\, n \in N.\ MOP_{E,\iota}^{\mathcal{S}_G}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!]_{E,\iota}(c_{\mathbf{s}}) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\}$$

The *MaxFP* Approach and *MaxFP* Solution

...for an edge-labelled single instruction flow graph.

Definition B.2.1.2 (The *MaxFP* Solution)

The $MaxFP_{E,\iota}$ solution of $\mathcal{S}_G$ is defined by:

$$MaxFP^{\mathcal{S}_G}_{E,\iota} : N \to \mathcal{C}$$

$$\forall\, n \in N.\ MaxFP^{\mathcal{S}_G}_{E,\iota}(n) =_{df} inf^{*}_{c_{\mathbf{s}}}(n)$$

where $inf_{c_{\mathbf{s}}}$ denotes the greatest solution of the *MaxFP* Equation System:

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

B.2.2

MOP and *MaxFP* Approach for Node-labelled BB Flow Graphs

Fixing the Setting (1)

In the following we denote

- basic block nodes by boldface letters (**m**, **n**,...)
- single instruction nodes by normalface letters (m, n,...)

We start from

- $G = (N, E, \mathbf{s}, \mathbf{e})$, a node-labelled SI flow graph
- $\mathcal{S}_G =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]_{N,\iota}, c_{\mathbf{s}}, fw)$, a DFA specification

which induce a BB flow graph $\mathbf{G}$ and a corresponding DFA specification $\mathcal{S}_{\mathbf{G}}$.

Fixing the Setting (2)

Given G and $\mathcal{S}_G$, let

- $\mathbf{G} = (\mathbf{N}, \mathbf{E}, \mathbf{s_G}, \mathbf{e_G})$
- $\mathcal{S}_\mathbf{G} =_{df} (\widehat{\mathcal{C}}, [\![\ ]\!]_{\mathbf{N},\beta}, c_\mathbf{s}, fw)$

denote the node-labelled BB flow graph and the DFA specification induced by G and $\mathcal{S}_G$, respectively, where

- $[\![\ ]\!]_{\mathbf{N},\beta} : \mathbf{N} \to \mathcal{C} \to \mathcal{C}$

denotes the extension of the SI DFA functional $[\![\ ]\!]_{N,\iota}$ from nodes to basic blocks defined by

- $\forall\, \mathbf{n} = \langle n_{\iota_1}, \ldots, n_{\iota_k} \rangle \in \mathbf{N}.\ [\![\, \mathbf{n} \,]\!]_{N,\beta} =_{df} [\![\, \langle n_1, \ldots, n_k \rangle \,]\!]_{N,\iota}$

Fixing the Setting (3)

Auxiliary Mappings

- *bb*: maps a node n to the basic block $\mathbf{n}$ it is included in.
- *start*: maps a basic block node $\mathbf{n}$ to its entry node n.
- *end*: maps a basic block node $\mathbf{n}$ to its exit node n.

The *MOP* Approach and *MOP* Solution (1)

...for a node-labelled basic block flow graph.

Definition B.2.2.1 (The *MOP* Solution, Part 1)

The $MOP_{\mathbf{N},\beta}$ solution of $\mathcal{S}_{\mathbf{G}}$ is defined by

$$MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}} : \mathbf{N} \to (\mathcal{C}, \mathcal{C})$$

$$\forall\, \mathbf{n} \in \mathbf{N}.\; MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}) =_{df} (\, N\text{-}MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}),\; X\text{-}MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n})\,)$$

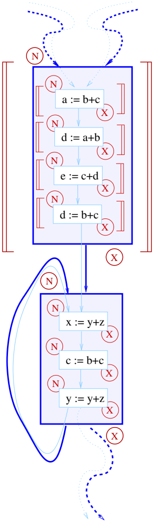
where

$$N\text{-}MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}) =_{df} \bigsqcap \{\, [\![\, p \,]\!]_{\mathbf{N},\beta}(c_{\mathbf{s}}) \mid p \in \mathbf{P_G}[\mathbf{s}, \mathbf{n}[\,\}$$

$$X\text{-}MOP_{\mathbf{N},\beta}^{\mathcal{S}_{\mathbf{G}}}(\mathbf{n}) =_{df} \bigsqcap \{\, [\![\, p \,]\!]_{\mathbf{N},\beta}(c_{\mathbf{s}}) \mid p \in \mathbf{P_G}[\mathbf{s}, \mathbf{n}]\,\}$$

The *MOP* Approach and *MOP* Solution (2)

Entry (N) and exit (X) information for basic block nodes must be pushed inside of the basic blocks:



The *MOP* Approach and *MOP* Solution (3)

...and its refinement to the SI Level:

Definition B.2.2.1 (The *MOP* Solution, Part 2)

The $MOP_{N,\iota}$ solution of $\mathcal{S}_G$ is defined by

$$MOP_{N,\iota}^{\mathcal{S}_G} : N \to (\mathcal{C}, \mathcal{C})$$

$$\forall\, n \in N.\; MOP_{N,\iota}^{\mathcal{S}_G}(n) =_{df} (\, N\text{-}MOP_{N,\iota}^{\mathcal{S}_G}(n),\; X\text{-}MOP_{N,\iota}^{\mathcal{S}_G}(n)\,)$$

The *MOP* Approach and *MOP* Solution (4)

where

$$N\text{-}MOP^{\mathcal{S}_G}_{N,\iota}(n) \;=_{df}\; \begin{cases} N\text{-}MOP^{\mathcal{S}_\mathbf{G}}_{\mathbf{N},\beta}(bb(n)) \\ \quad \text{if } n = start(bb(n)) \\ \\ [\![\, p \,]\!]_{N,\iota}(N\text{-}MOP^{\mathcal{S}_\mathbf{G}}_{\mathbf{N},\beta}(bb(n))) \\ \quad \text{otherwise (note: } p \text{ is the prefix path from} \\ \quad start(bb(n)) \text{ up to but exclusive of } n) \end{cases}$$

$$X\text{-}MOP^{\mathcal{S}_G}_{N,\iota}(n) \;=_{df}\; [\![\, p \,]\!]_{N,\iota}(N\text{-}MOP^{\mathcal{S}_\mathbf{G}}_{\mathbf{N},\beta}(bb(n)))$$

(note: p is the prefix path from $start(bb(n))$ up to and inclusive of n)

The *MaxFP* Approach and *MaxFP* Solution (1)

...for a node-labelled basic block flow graph.

Definition B.2.2.2 (The *MaxFP* Solution, Part 1)

The $MaxFP_{\mathbf{N},\beta}$ solution of $\mathcal{S}_{\mathbf{G}}$ is defined by

$$\forall\, \mathbf{n} \in \mathbf{N}.\ MaxFP^{\mathcal{S}_{\mathbf{G}}}_{\mathbf{N},\beta}(\mathbf{n}) =_{df} (\, N\text{-}MaxFP^{\mathcal{S}_{\mathbf{G}}}_{\mathbf{N},\beta}(\mathbf{n}),\ X\text{-}MaxFP^{\mathcal{S}_{\mathbf{G}}}_{\mathbf{N},\beta}(\mathbf{n})\,)$$

where

$$N\text{-}MaxFP^{\mathcal{S}_{\mathbf{G}}}_{\mathbf{N},\beta}(\mathbf{n}) =_{df} \mathrm{pre}^{\star}_{c_s,\beta}(\mathbf{n})$$

$$X\text{-}MaxFP^{\mathcal{S}_{\mathbf{G}}}_{\mathbf{N},\beta}(\mathbf{n}) =_{df} \mathrm{post}^{\star}_{c_s,\beta}(\mathbf{n})$$

The *MaxFP* Approach and *MaxFP* Solution (2)

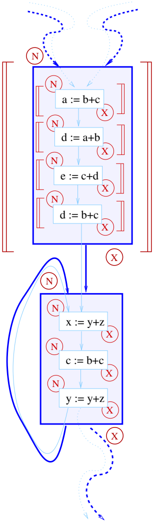
...and where $\text{pre}^{\star}_{c_{\mathbf{s}},\beta}$ and $\text{post}^{\star}_{c_{\mathbf{s}},\beta}$ denote the greatest solution of the *MaxFP* Equation System:

$$\text{pre}(\mathbf{n}) \;=\; \begin{cases} c_{\mathbf{s}} & \text{if } \mathbf{n} = \mathbf{s} \\ \bigsqcap \{\, \text{post}(\mathbf{m}) \mid \mathbf{m} \in \mathit{pred}_{\mathbf{G}}(\mathbf{n}) \,\} & \text{otherwise} \end{cases}$$

$$\text{post}(\mathbf{n}) \;=\; [\![\, \mathbf{n} \,]\!]_{\mathbf{N},\beta}(\text{pre}(\mathbf{n}))$$

The *MaxFP* Approach and *MaxFP* Solution (3)

Entry (N) and exit (X) information for basic block nodes must be pushed inside of the basic blocks:



The *MaxFP* Approach and *MaxFP* Solution (4)

...and its refinement to the SI Level:

Definition B.2.2.2 (The *MaxFP* Solution, Part 2)

The $MaxFP_{N,\iota}$ solution of $\mathcal{S}_G$ is defined by

$$\forall\, n \in N.\ MaxFP^{\mathcal{S}_G}_{N,\iota}(n) =_{df} (\, N\text{-}MaxFP^{\mathcal{S}_G}_{N,\iota}(n),\ X\text{-}MaxFP^{\mathcal{S}_G}_{N,\iota}(n)\,)$$

where

$$N\text{-}MaxFP^{\mathcal{S}_G}_{N,\iota}(n) =_{df} \mathrm{pre}^{\star}_{c_s,\iota}(n) \qquad \text{and}$$
$$X\text{-}MaxFP^{\mathcal{S}_G}_{N,\iota}(n) =_{df} \mathrm{post}^{\star}_{c_s,\iota}(n)$$

The *MaxFP* Approach and *MaxFP* Solution (5)

...and where $\mathrm{pre}^{\star}_{c_s,\iota}$ and $\mathrm{post}^{\star}_{c_s,\iota}$ denote the greatest solution of the *MaxFP* Equation System:

$$\mathrm{pre}(n) = \begin{cases} \mathrm{pre}^{\star}_{c_s,\beta}(bb(n)) & \text{if } n = start(bb(n)) \\ \mathrm{post}(m) & \text{otherwise, where } m \text{ is the unique predecessor of } n \text{ in } bb(n) \end{cases}$$

$$\mathrm{post}(n) = \begin{cases} \mathrm{post}^{\star}_{c_s,\beta}(bb(n)) & \text{if } n = end(bb(n)) \\ [\![\, n \,]\!]_{N,\iota}(\mathrm{pre}(n)) & \end{cases}$$

B.3
Available Expressions

B.3.1

Available Expressions for Node-labelled BB Flow Graphs

Available Expressions (1)

...for a single term t and a node-labelled BB flow graph.

Stage I: The Basic Block Level

Local Predicates (associated with BB nodes):

- BB-X-Comp$^t_\beta$: β contains a statement ι computing t, and neither ι nor a statement following ι in β modifies an operand of t.
- BB-Transp$^t_\beta$: β does not contain a statement which modifies an operand of t.

Available Expressions (2)

The Basic Block *MaxFP* Equation System of Stage I:

$$\text{BB-N-Avail}_\beta \;=\; \begin{cases} c_{\mathbf{s}} & \text{if } \beta = \mathbf{s_G} \\ \displaystyle\bigwedge_{\hat\beta \in pred(\beta)} \text{BB-X-Avail}_{\hat\beta} & \text{otherwise} \end{cases}$$

$$\text{BB-X-Avail}_\beta \;=\; (\text{BB-N-Avail}_\beta \wedge \text{BB-Transp}^t_\beta) \vee \text{BB-X-Comp}^t_\beta$$

Available Expressions (3)

Stage II: The Instruction Level

Local Predicates (associated with SI nodes):

- Comp_ι^t: ι computes t.
- Transp_ι^t: ι does not modify an operand of t.
- BB-N-Avail*, BB-X-Avail*: the greatest solution of the *MaxFP* Equation System of Stage I.

Auxiliary Mappings

- *bb*: maps an instruction ι to the basic block β it is included in.
- *start*: maps a basic block β to its entry instruction ι.
- *end*: maps a basic block β to its exit instruction ι.

Available Expressions (4)

The Single Instruction *MaxFP* Equation System of Stage II:

$$\text{N-Avail}_\iota \;=\; \begin{cases} \text{BB-N-Avail}^\star_{bb(\iota)} & \text{if } \iota = \mathit{start}(bb(\iota)) \\ \text{X-Avail}_{pred(\iota)} & \text{otherwise (note: } \mid \mathit{pred}(\iota) \mid = 1) \end{cases}$$

$$\text{X-Avail}_\iota \;=\; \begin{cases} \text{BB-X-Avail}^\star_{bb(\iota)} & \text{if } \iota = \mathit{end}(bb(\iota)) \\ \\ (\text{N-Avail}_\iota \vee \text{Comp}^t_\iota) \wedge \text{Transp}^t_\iota & \text{otherwise} \end{cases}$$

B.3.2
Available Expressions for Node-labelled SI Flow Graphs

Available Expressions

...for a single term t and a node-labelled SI flow graph.

Local Predicates (associated with SI nodes):

- Comp_ι^t: ι computes t.
- Transp_ι^t: ι does not modify an operand of t.

The Single Instruction *MaxFP* Equation System:

$$\text{N-Avail}_\iota = \begin{cases} c_{\mathbf{s}} & \text{if } \iota = \mathbf{s} \\ \bigwedge_{\hat{\iota} \in pred(\iota)} \text{X-Avail}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

$$\text{X-Avail}_\iota = (\text{N-Avail}_\iota \vee \text{Comp}_\iota^t) \wedge \text{Transp}_\iota^t$$

B.3.3
Available Expressions for Edge-labelled SI Flow Graphs

Available Expressions

...for a single term t and an edge-labelled SI flow graph.

Locale Predicates (associated with SI edges):

- $\text{Comp}_\varepsilon^t$: Instruction ι of edge ε computes t.
- $\text{Transp}_\varepsilon^t$: Instruction ι of edge ε does not modify an operand of t.

The Single Instruction *MaxFP* Equation System:

$$\text{Avail}_n \;=\; \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigwedge\limits_{m \in pred(n)} (\text{Avail}_m \vee \text{Comp}^t_{(m,n)}) \wedge \text{Transp}^t_{(m,n)} & \text{otherwise} \end{cases}$$

Looking ahead

Next we consider two more examples to illustrate the impact of a chosen flow graph variant on the conceptual and practical complexity of data flow analysis:

- Simple constants analysis (cf. Chapter B.4)
- Faint variables analysis (cf. Chapter B.5)

To this end we will oppose and investigate *MaxFP* formulations of these problems for

- node-labelled BB flow graphs
- edge-labelled SI flow graphs

B.4
Simple Constants

Simple Constants Analysis

...for the formal problem formulation we require two auxiliary functions:

- Backward substitution δ
- State transformation θ

Backward Substitution, State Transformation

Let $\iota \equiv (x := t)$ be an instruction. We define:

- Backward substitution δ_ι
  $\delta_\iota : \mathbf{T} \rightarrow \mathbf{T}$ defined by

  $$\forall\, s \in \mathbf{T}.\ \delta_\iota(s) =_{df} s[t/x]$$

  where $s[t/x]$ denotes the simultaneous replacement of all occurrences of x by t in s.

- State transformation θ_ι
  $\theta_\iota : \Sigma \rightarrow \Sigma$ defined by

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

The Relationship of δ and θ

Let $\mathcal{I}$ denote the set of all instructions.

Lemma B.4.1 (Substitution Lemma for Instructions)

$$\forall\, \iota \in \mathcal{I}\ \forall\, t \in \mathbf{T}\ \forall\, \sigma \in \Sigma.\ \mathcal{E}(\delta_\iota(t))(\sigma) = \mathcal{E}(t)(\theta_\iota(\sigma))$$

Proof by induction on the structure of t.

B.4.1

Simple Constants for Edge-labelled SI Flow Graphs

Simple Constants Analysis

...for an edge-labelled SI flow graph.

The SI_E *MaxFP* Equation System:

$\forall\, v \in \mathbf{V}.\ \mathrm{SC}_n(v) =$

$$\begin{cases} \sigma_{\mathbf{s}}(v) & \text{if } n = \mathbf{s} \\ \prod\{\, \mathcal{E}(\delta_{(m,n)}(v))(\mathrm{SC}_m) \mid m \in \mathit{pred}(n)\,\} & \text{otherwise} \end{cases}$$

where $\sigma_{\mathbf{s}} \in \Sigma$ start information.

The Solution of the SI_E SC Analysis is given by

- $\mathrm{SC}^\star : N \to \Sigma$, the greatest solution of the above EQS.

B.4.2

Simple Constants for Node-labelled BB Flow Graphs

Backward Substitution and State Transformation for Paths

...adapting and extending δ and θ from edge-labelled flow graphs to path (and hence basic blocks) on node-labelled flow graphs:

- $\Delta_p : \mathbf{T} \to \mathbf{T}$ 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 $q > 1$.

The Relationship of Δ and Θ

Let $\mathcal{B}$ denote the set of all basic blocks.

Lemma B.4.2.1 (Substitution Lemma for BBs)

$$\forall\, \beta \in \mathcal{B}\ \forall\, t \in \mathbf{T}\ \forall\, \sigma \in \Sigma.\ \mathcal{E}(\Delta_\beta(t))(\sigma) = \mathcal{E}(t)(\Theta_\beta(\sigma))$$

Proof by induction on the length of β.

Simple Constants Analysis (1)

...for a node-labelled BB flow graph.

Stage I: The Basic Block Level

The BB_N *MaxFP* Equation System of Stage I:

$$\text{BB-N-SC}_\beta = \begin{cases} \sigma_{\mathbf{s}} & \text{if } \beta = \mathbf{s} \\ \prod\{\text{BB-X-SC}_{\hat{\beta}} \mid \hat{\beta} \in \mathit{pred}(\beta)\} & \text{otherwise} \end{cases}$$

$$\forall\, v \in \mathbf{V}.\ \text{BB-X-SC}_\beta(v) = \mathcal{E}(\Delta_\beta(v))(\text{BB-N-SC}_\beta)$$

where $\sigma_{\mathbf{s}} \in \Sigma$ start information.

The Solution of the BB_N SC Analysis is given by

- $\text{BB-N-SC}^\star_\beta, \text{BB-X-SC}^\star_\beta : \mathbf{N} \to \Sigma$, the greatest solutions of the above equation systems.

Simple Constants Analysis (2)

Stage II: The Instruction Level

Auxiliary Mappings

- *bb*: maps an instruction ι to the basic block β it is included in.
- *start*: maps a basic block β to its entry instruction ι.
- *end*: maps a basic block β to its exit instruction ι.

The SI_N *MaxFP* Equation System of Stage II:

$$\text{N-SC}_\iota \;=\; \begin{cases} \text{BB-N-SC}^\star_{bb(\iota)} & \text{if } \iota = start(bb(\iota)) \\ \text{X-SC}_{pred(\iota)} & \text{otherwise (note: } \\ & \;\; |\, pred(\iota)\,| = 1) \end{cases}$$

$$\forall\, v \in \mathbf{V}.\ \text{X-SC}_\iota(v) \;=\; \begin{cases} \text{BB-X-SC}^\star_{bb(\iota)}(v) & \text{if } \iota = end(bb(\iota)) \\ \mathcal{E}(\delta_\iota(v))(\text{N-CP}_\iota) & \text{otherwise} \end{cases}$$

Simple Constants Analysis (3)

The Solution of the SI_N SC Analysis is given by

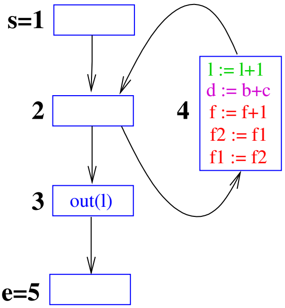
- N-SC*, X-SC* : $N \to \Sigma$, the greatest solution of the preceding equation systems.

B.5
Faint Variables

Faint Variables: Between Life and Death

The instruction

- $l \;:=\; l + 1$ is live,
- $d \;:=\; b + c$ is dead,
- $f \;:=\; f + 1$, $f1 \;:=\; f2$, and $f2 \;:=\; f1$ are live but faint (in German: kraftlos, ohnmächtig, schattenhaft).



Faint Variables Analysis (1)

...for an edge-labelled SI flow graph.

Local Predicates (associated with SI edges):

- Rel-Used$_\varepsilon^v$: a relevant use of variable v, i.e., v is used in the instruction ι associated with edge ε and "is forced to live" by it (i.e., ι is an output or test operation).
- Ass-Used$_\varepsilon^v$: every other use of variable v, i.e., v occurs in the right-hand side expression of the instruction ι associated with edge ε.
- Mod$_\varepsilon^v$: the instruction ι at edge ε modifies variable v.

Auxiliary Mapping

- *LhsVar*: maps an edge ε to the left-hand side variable of the instruction ι associated with it.

Faint Variables Analysis (2)

The SI_E *MaxFP* Equation System:

$\forall\, v \in \mathbf{V}.\ \mathrm{Faint}_n^v\ =$

$$\begin{cases} fv_{\mathbf{e}} & \text{if } n = \mathbf{e} \\ \displaystyle\bigwedge_{m \in succ(n)} \neg\text{Rel-Used}_{(n,m)}^v \wedge \\ \qquad\qquad (\mathrm{Mod}_{(n,m)}^v \ \vee\ \mathrm{Faint}_m^v) \wedge \\ \qquad\qquad (\neg\text{Ass-Used}_{(n,m)}^v \ \vee\ \mathrm{Faint}_m^{LhsVar_{(n,m)}}) & \text{otherwise} \end{cases}$$

where $fv_{\mathbf{e}} \in \mathrm{IB}^{|\mathbf{V}|}$ start information.

The Solution of the SI_E Faint Variables Analysis is given by

- $\mathrm{Faint}^\star : N \to \mathrm{IB}^{|\mathbf{V}|}$, the greatest solution of the above equation system.

Summing up

The faint variables problem is an example of a non-separable DFA problem, where a formulation leading to an efficient implementation is

- obvious for (node and edge-labelled) SI flow graphs,
- not at all obvious, if not impossible at all, for (node and edge-labelled) BB flow graphs.

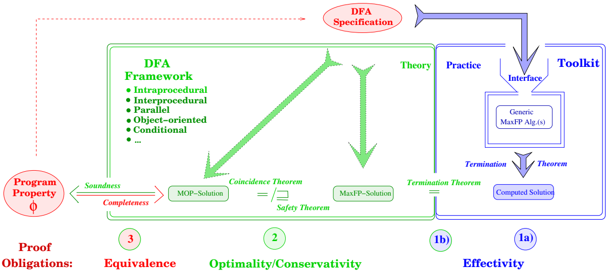
  (Note that the naive straightforward extension to BB graphs would require for every basic block $\mathbf{n}$ to compute the full semantic function $[\![\mathbf{n}]\!]_{faint} : \mathbb{B}^{k_\mathbf{n}} \to \mathbb{B}^{k_\mathbf{n}}$, where $k_\mathbf{n}$ is the number of variables occuring in $\mathbf{n}$, a function with $2^{k_\mathbf{n}}$ arguments. In the worst case, $k_\mathbf{n}$ coincides even with the number of all variables in the program under consideration.)

B.6
Conclusions

In closing, all

...4 flow graph variants are essentially equivalent with in most cases only minor pragmatic advantages and disadvantages.

Thus the general holistic framework and tool kit view of DFA



is conceptually adequate and sufficient while being aware of these differences for the adequacy and ease of their use in specification, implementation, and proof obligation accomplishment.

B.7
References, Further Reading

Further Reading for Appendix B

- 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.

Appendix C
Implementing Busy and Lazy Code Motion on SI and BB Flow Graphs

C.1

Implementing *BCM* and *LCM* on SI Graphs

C.1.1
Preliminaries

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).

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.

C.1.2
Implementing BCM_ι

Implementing BCM_ι (1)

1. Analyses for Up-Safety and Down-Safety

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

$$\text{N-USAFE}_\iota \;=\; \begin{cases} \mathit{false} & \text{if } \iota = \mathbf{s} \\ \prod\limits_{\hat\iota \in \mathit{pred}(\iota)} \text{X-USAFE}_{\hat\iota} & \text{otherwise} \end{cases}$$

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

Implementing BCM_ι (2)

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

$$
\begin{aligned}
\text{N-DSAFE}_\iota &= \text{COMP}_\iota + \text{X-DSAFE}_\iota \cdot \text{TRANSP}_\iota \\
\text{X-DSAFE}_\iota &= \begin{cases} \mathit{false} & \text{if } \iota = \mathbf{e} \\ \prod\limits_{\hat\iota \in \mathit{succ}(\iota)} \text{N-DSAFE}_{\hat\iota} & \text{otherwise} \end{cases}
\end{aligned}
$$

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.

Implementing BCM_ι (4)

Computing Earliestness (no data flow analysis!):

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

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

Implementing BCM_ι (5)

The BCM_ι Transformation:

$$\begin{array}{rcl} \text{N-INSERT}_\iota^{\text{BCM}} & =_{df} & \text{N-EARLIEST}_\iota \\ \text{X-INSERT}_\iota^{\text{BCM}} & =_{df} & \text{X-EARLIEST}_\iota \\ \text{REPLACE}_\iota^{\text{BCM}} & =_{df} & \text{COMP}_\iota \end{array}$$

C.1.3
Implementing LCM_ι

Implementing LCM_ι (1)

3. Analyses for Delayability and Isolation

The *MaxFP*-Equation System for Delayability:

$$\text{N-DELAYED}_\iota = \text{N-EARLIEST}_\iota + \begin{cases} \mathit{false} & \text{if } \iota = \mathbf{s} \\ \prod_{\iota' \in \mathit{pred}(\iota)} \text{X-DELAYED}_{\iota'} & \mathit{otherwise} \end{cases}$$

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

Implementing LCM_ι (2)

Computing Latestness (no data flow analysis!):

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

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

where

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

Implementing LCM_ι (3)

The $ALCM_\iota$ Transformation:

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

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

Implementing LCM_ι (4)

The *MaxFP*-Equation System for Isolation:

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

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

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.

Implementing LCM_ι (6)

The LCM_ι Transformation:

$$\begin{array}{lcl} \text{N-INSERT}_\iota^{\text{LCM}} & =_{df} & \text{N-LATEST}_\iota \cdot \overline{\text{N-ISOLATED}_\iota^\star} \\ \text{X-INSERT}_\iota^{\text{LCM}} & =_{df} & \text{X-LATEST}_\iota \\ & & \\ \text{REPLACE}_\iota^{\text{LCM}} & =_{df} & \text{COMP}_\iota \cdot \overline{\text{N-LATEST}_\iota \cdot \text{N-ISOLATED}_\iota^\star} \end{array}$$

C.2
Implementing *BCM* and *LCM* on BB Graphs

C.2.1
Preliminaries

Implementing Busy and Lazy Code Motion

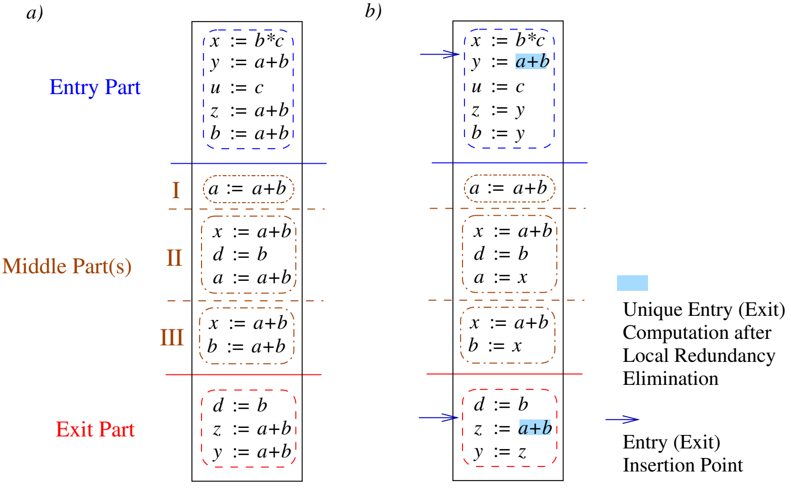
...for node-labelled BB graphs:

- BCM_β Transformation
- LCM_β Transformation

Convention: For the following we assume that (1) only critical edges are split. Therefore, BCM_β and LCM_β require insertions at both node entries and node exits (N-insertions and X-insertions), and that (2) all redundancies within a basic block have been removed by a preprocess.

Conceptual Splitting of a Basic Block

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



Original Basic Block

Basic Block after Local Redundancy Elimination

Entry and Exit Parts of a Basic Block

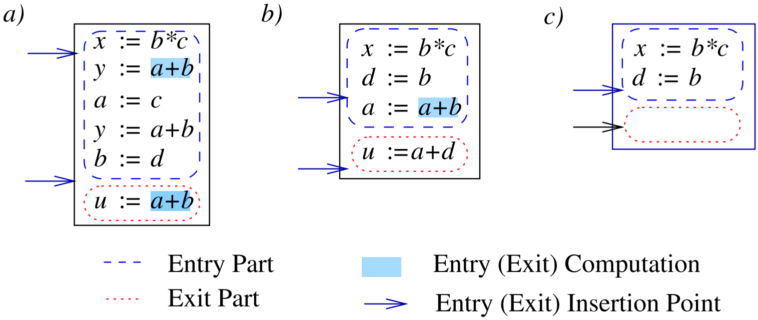
For PRE, we do not need to distinguish between entry and middle part(s), and can consider them a unit. This gives rise to the following definition:

Given a computation t, a basic block $\mathbf{n}$ 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 $\mathbf{n}$.

Note: The entry part of a non-empty basic block is always non-empty; in distinction, the exit part of a non-empty basic block can be empty (as illustrated in the following figure).

Illustrating Entry & Exit Part of a Basic Block



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}

Local Predicates for BCM_β and LCM_β

Local Predicates:

- BB-N-Comp$_\beta(t)$: β contains a statement ι that computes t, and that is not preceded by a statement that modifies an operand of t.
- BB-X-Comp$_\beta(t)$: 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 modi- fies an operand of t.

C.2.2
Implementing BCM_β

Implementing BCM_β (1)

1. Analyses for Up-Safety and Down-Safety

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

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

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

Implementing BCM_β (2)

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

$$\text{BB-N-DSAFE}_\beta \quad = \quad \text{BB-N-Comp}_\beta + \text{BB-X-DSAFE}_\beta \cdot \text{BB-Transp}_\beta$$

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

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.

Implementing BCM_β (4)

Computing Earliestness (no data flow analysis!):

$$\begin{aligned}
\text{N-EARLIEST}_\beta \quad &=_{df} \quad \text{BB-N-DSAFE}^\star_\beta \cdot \\
&\qquad \prod_{\hat\beta \in \mathit{pred}(\beta)} (\, \overline{\text{BB-X-USAFE}^\star_{\hat\beta} + \text{BB-X-DSAFE}^\star_{\hat\beta}}\,) \\
\text{X-EARLIEST}_\beta \quad &=_{df} \quad \text{BB-X-DSAFE}^\star_\beta \cdot \overline{\text{BB-Transp}_\beta}
\end{aligned}$$

Implementing BCM_β (5)

The BCM_β Transformation:

$$\begin{array}{rcl}
\text{N-INSERT}_\beta^{\text{BCM}} & =_{df} & \text{N-EARLIEST}_\beta \\
\text{X-INSERT}_\beta^{\text{BCM}} & =_{df} & \text{X-EARLIEST}_\beta \\[1ex]
\text{N-REPLACE}_\beta^{\text{BCM}} & =_{df} & \text{BB-N-Comp}_\beta \\
\text{X-REPLACE}_\beta^{\text{BCM}} & =_{df} & \text{BB-X-Comp}_\beta
\end{array}$$

C.2.3
Implementing LCM_β

Implementing LCM_β (1)

3. Analyses for Delayability and Isolation

The *MaxFP*-Equation System for Delayability:

$$\text{N-DELAYED}_\beta = \text{N-EARLIEST}_\beta + \begin{cases} \mathit{false} & \mathit{if} \quad \beta = \mathbf{s} \\ \prod\limits_{\hat{\beta} \in \mathit{pred}(\beta)} \overline{\text{BB-X-Comp}_{\hat{\beta}}} \cdot \text{X-DELAYED}_{\hat{\beta}} & \mathit{otherwise} \end{cases}$$

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

Implementing LCM_β (2)

Computing Latestness (no data flow analysis!):

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

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

where

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

Implementing LCM_β (3)

The $ALCM_\beta$ Transformation:

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

$$\begin{aligned}
\text{N-REPLACE}_\beta^{\text{ALCM}} &=_{df} \text{BB-N-Comp}_\beta \\
\text{X-REPLACE}_\beta^{\text{ALCM}} &=_{df} \text{BB-X-Comp}_\beta
\end{aligned}$$

Implementing LCM_β (4)

The *MaxFP*-Equation System for Isolation:

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

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

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.

Implementing LCM_β (6)

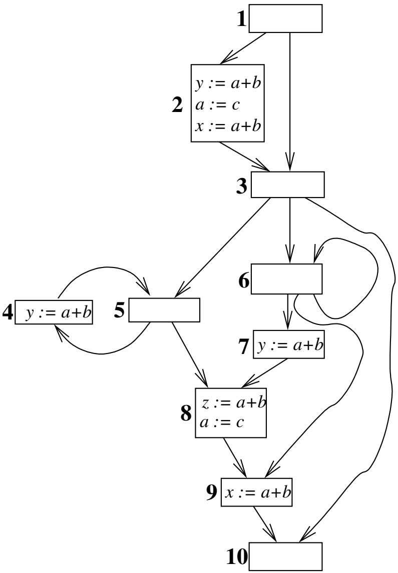
The LCM_β Transformation:

$$\begin{array}{rcl}
\text{N-INSERT}_\beta^{\text{LCM}} & =_{df} & \text{N-LATEST}_\beta \cdot \overline{\text{N-ISOLATED}_\beta^\star} \\
\text{X-INSERT}_\beta^{\text{LCM}} & =_{df} & \text{X-LATEST}_\beta \cdot \overline{\text{X-ISOLATED}_\beta^\star}
\end{array}$$

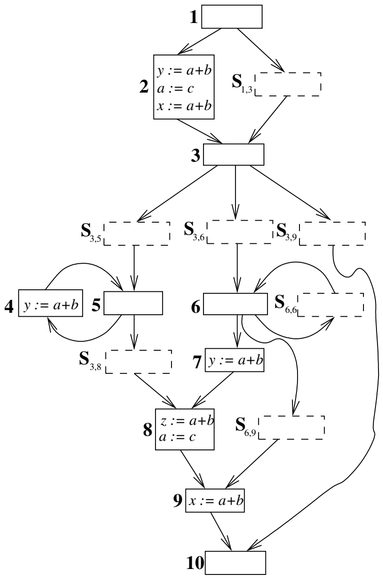
$$\begin{array}{rcl}
\text{N-REPLACE}_\beta^{\text{LCM}} & =_{df} & \text{BB-N-Comp}_\beta \cdot \overline{\text{N-LATEST}_\beta \cdot \text{N-ISOLATED}_\beta^\star} \\
\text{X-REPLACE}_\beta^{\text{LCM}} & =_{df} & \text{BB-X-Comp}_\beta \cdot \overline{\text{X-LATEST}_\beta \cdot \text{X-ISOLATED}_\beta^\star}
\end{array}$$

C.3
Illustrating Example

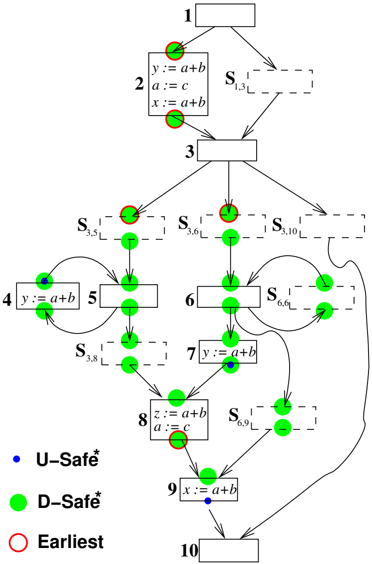
The Original Program



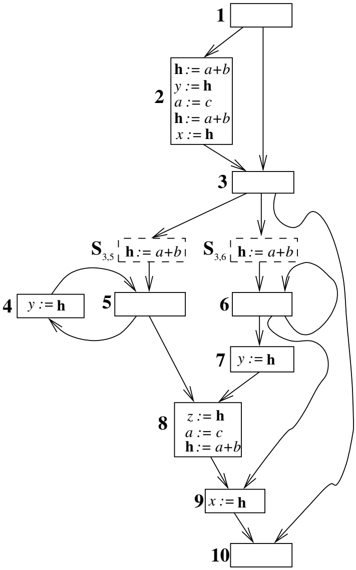
After the Splitting of Critical Edges



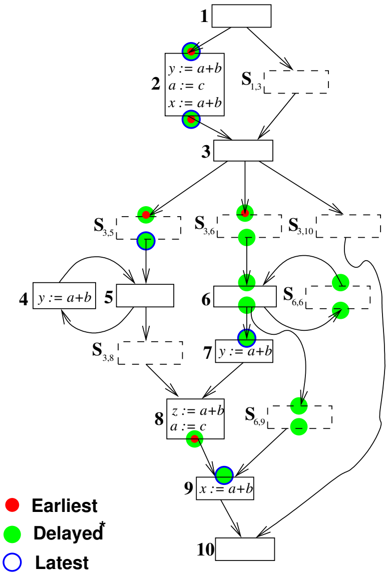
Up/Down-Safe, Earliest Computation Points



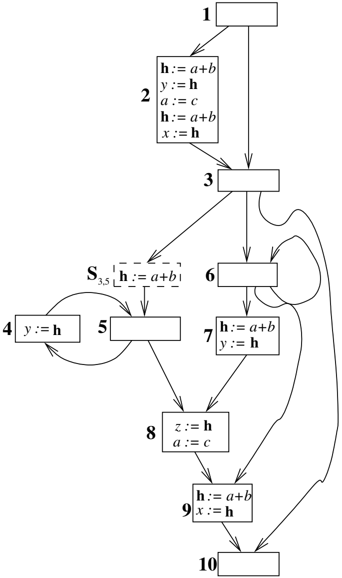
The Result of the BCM_β Transformation



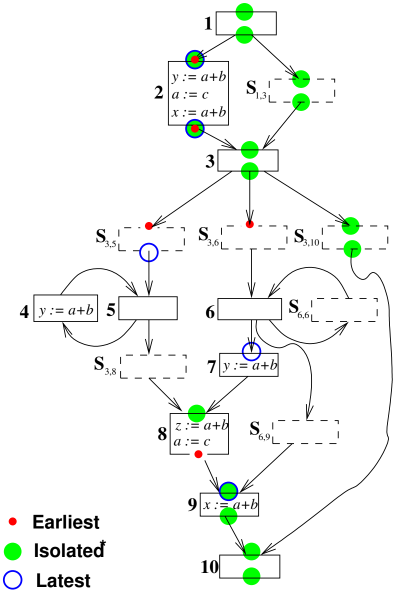
Delayable and Latest Computation Points



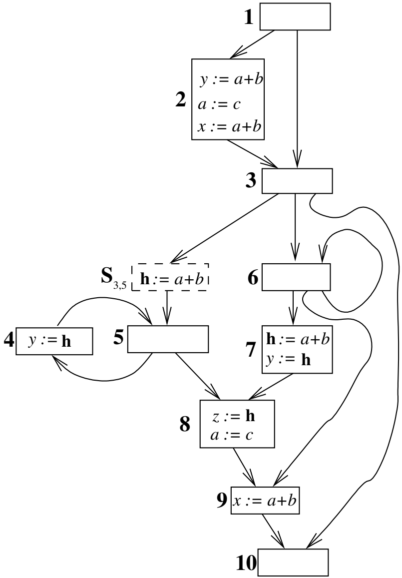
The Result of the $ALCM_\beta$-Transformation



Latest and Isolated Program Points



The Result of the LCM_β Transformation



C.4
References, Further Reading

Further Reading for Appendix C

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Optimal Code Motion: Theory and Practice*. ACM Transactions on Programming Languages and Systems 16(4):1117-1155, 1994.

Appendix D
Lazy Strength Reduction

D.1
Motivation

From *BCM* to *LSR*

...from busy code motion to lazy strength reduction.

Objective: Developing a program optimization which

- uniformly covers
  - Partial Redundancy Elimination (PRE)
  - Strength Reduction (SR)
- avoids superfluous register pressure caused by unnecessary code motion
- requires only uni-directional data flow analyses.

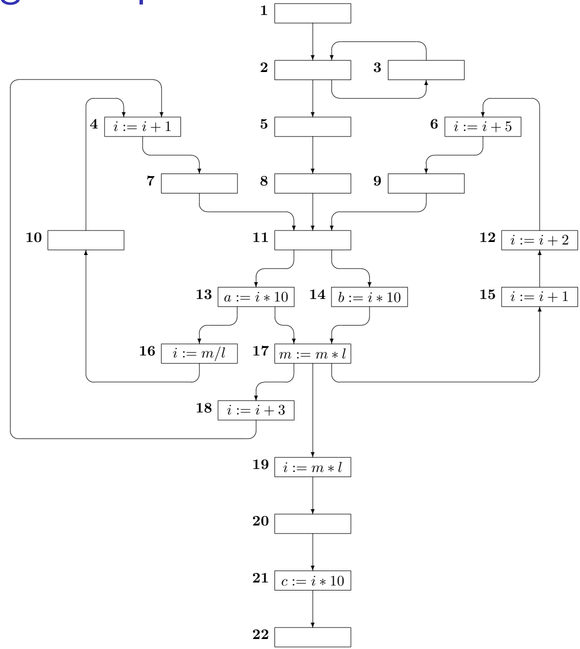
The Approach

We will stepwise and modularly extend

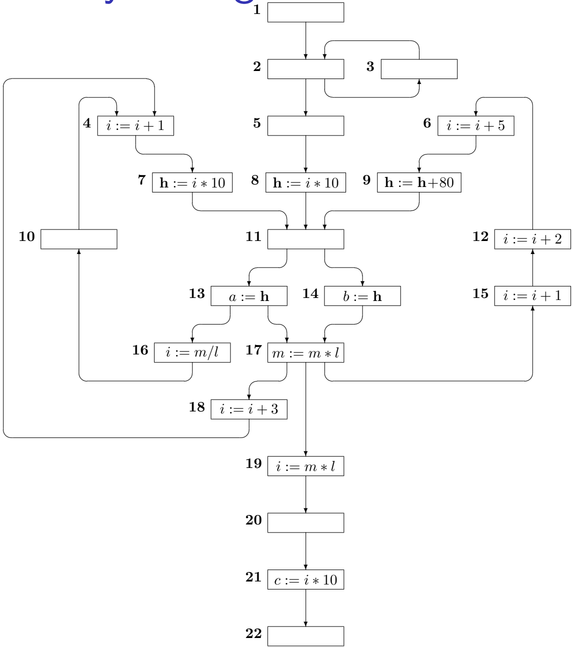
- the *BCM* and the *LCM* to arrive at the
  - Busy Strength Reduction (*BSR*)
  - Lazy Strength Reduction (*LSR*)

D.2
Running Example

The Running Example



The Result of Lazy Strength Reduction



D.3
Preliminaries

Candidate Expressions

...for lazy strength reduction.

Candidate expressions for

- Code motion: No restrictions, every term $t \in \mathbf{T}$
- Lazy strength reduction: Every term of the form $v * c \in \mathbf{T}$, where
  - v is a variable
  - c is a source code constant

Local Predicates for Lazy Strength Reduction

Local predicates for lazy strength reduction

- $Used(n) =_{df} v * c \in SubTerms(t)$
- $Transp(n) =_{df} x \not\equiv v$
- $SR\text{-}Transp(n) =_{df} Transp(n) \vee t \equiv v + d$ with $d \in \mathbf{C}$

Note: The value of a candidate expression $v * c$ is

- killed at a node n, if $\neg(Transp(n) \vee SR\text{-}Transp(n))$

while it is

- injured at a node n, if $\neg Transp(n) \wedge SR\text{-}Transp(n)$

The Essence of Strength Reduction

Values which are injured but not killed can be

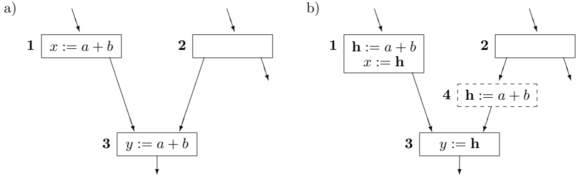
- cured by inserting an update instruction of the form
  $\mathbf{h} := \mathbf{h} + Cure(n)$, where $Cure(n) =_{df} c * d$.

Note that the (correction) value of $Cure(n)$ can be computed at compile time since both c and d are source code constants.

Splitting of Critical Edges

As common for code motion transformations like *BCM* and *LCM*, critical edges need to be split in order to get the full power of

- Lazy Strength Reduction (LSR)



Splitting of Join Edges

Moreover, in order to allow insertions of statements

- uniformly at node entries

we assume that even

- all join edges (and not just critical edges)

are split as for the *BCM* and *LCM* transformations (cf. Chapter 7 and 8).

D.4

Extending *BCM* to Strength Reduction: The *BSR* Transformation

Extending *BCM* to Strength Reduction

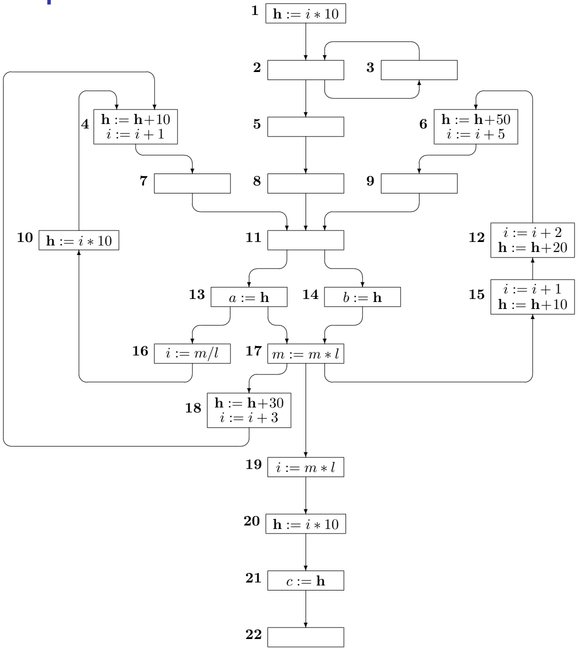
...straightforward leads to Busy Strength Reduction (*BSR*).

The *BSR* Transformation

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node n satisfying
   2.1 $\mathtt{Ins}_{\mathtt{BSR}}$ the assignment $\mathbf{h} := v * c$
   2.2 $\mathtt{InsUpd}_{\mathtt{BSR}}$ the assignment $\mathbf{h} := \mathbf{h} + Cure(n)$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

Note: If $\mathtt{Ins}_{\mathtt{BSR}}$ and $\mathtt{InsUpd}_{\mathtt{BSR}}$ hold both at some node n, the initialization statement $\mathbf{h} := v * c$ must precede the update instruction $\mathbf{h} := \mathbf{h} + Cure(n)$.

Running Example: The Result of *BSR*



Shortcomings of *BSR*

Shortcoming

- *BSR* can suffer from multiplication-addition-sequences.

Remedy

- Moving insertions at $(*, +)$-critical insertion points in the direction of the control flow to the "earliest" non-critical ones.

Note

- An insertion point is $(*, +)$-critical if there is a $v * c$-free program path from this point to a modification site of v.

D.5

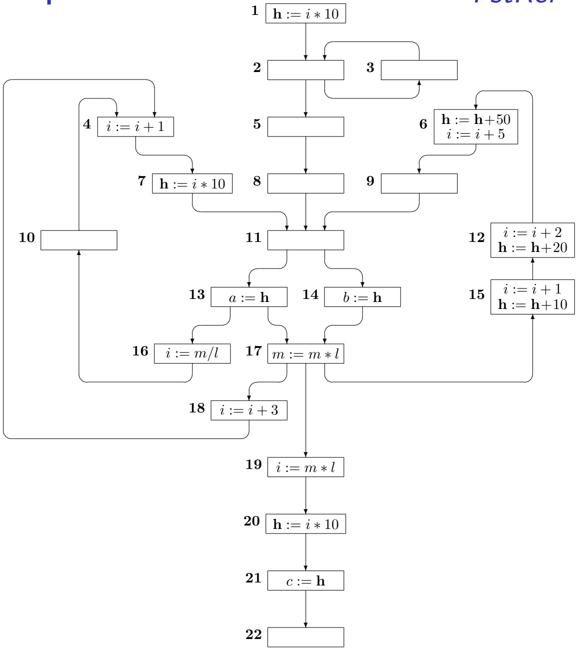
The 1st Refinement: Avoiding Multiplication-Addition Sequences – The BSR_{FstRef} Transformation

BSR_{FstRef}: Avoiding $(*, +)$ Sequences

The BSR_{FstRef} Transformation

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node n satisfying
   2.1 $\texttt{Ins}_{FstRef}$ the assignment $\mathbf{h} := v * c$
   2.2 $\texttt{InsUpd}_{FstRef}$ the assignment $\mathbf{h} := \mathbf{h} + Cure(n)$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

Running Example: The Result of BSR_{FstRef}



Shortcomings of BSR_{FstRef}

Shortcoming

- BSR_{FstRef} can suffer from unnecessary register pressure due to unnecessary code motion.

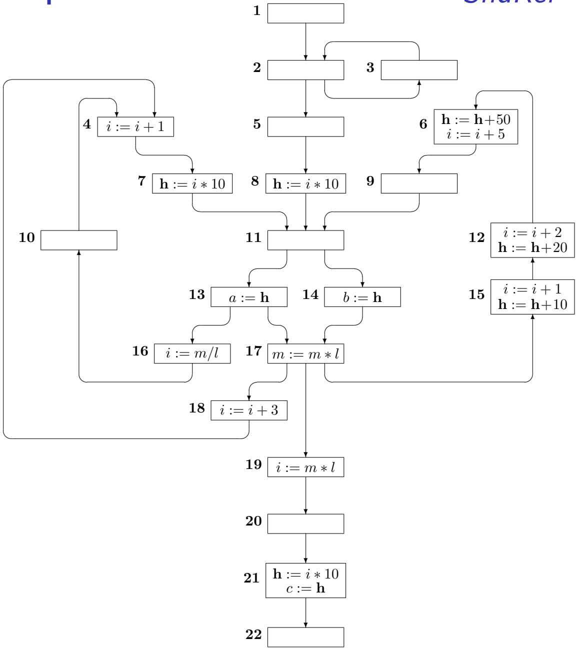
Remedy

- Adding lazyness to BSR_{FstRef}.

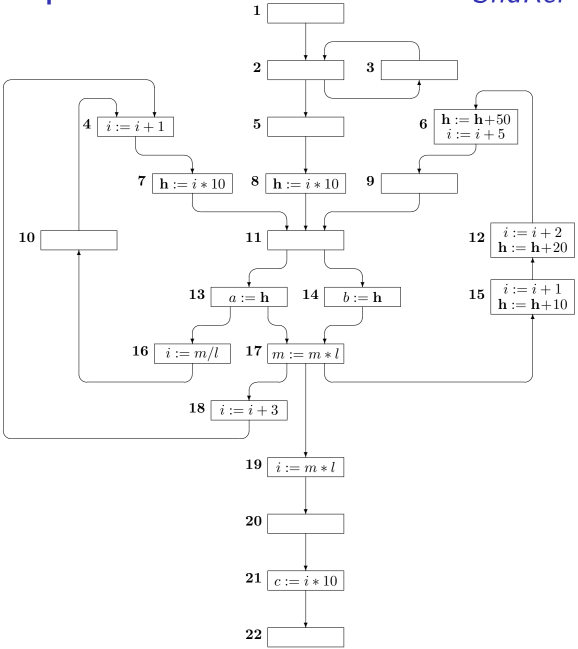
D.6

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

Running Example: Pre-Result of BSR_{SndRef}



Running Example: The Result of BSR_{SndRef}



Shortcomings of BSR_{SndRef}

Shortcoming

- BSR_{SndRef} can suffer from multiple-addition sequences.

Remedy

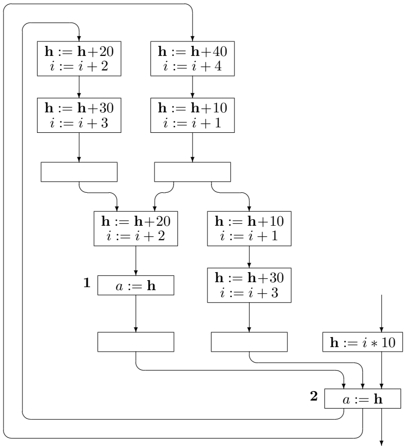
- Replacing of multiple-addition sequences by a single cumulated addition instruction.

Note

- The resulting third refinement of the BSR transforma- tion, BSR_{ThdRef}, defines the

  Lazy Strength Reduction (LSR), i.e., $LSR =_{df} BSR_{ThdRef}$.

Illustrating Multiple-Addition Sequences

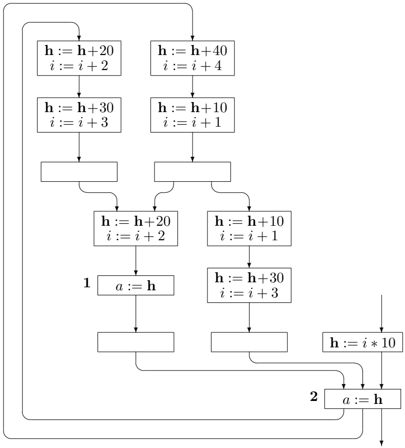


D.7

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

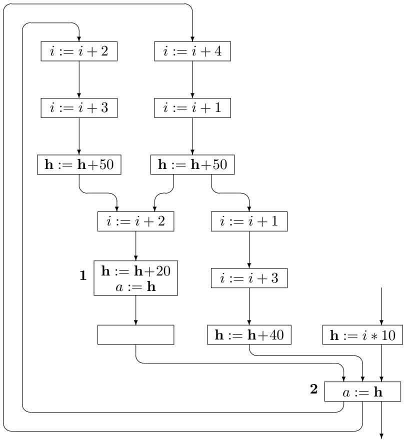
Illustrating Example

...suffering from multiple-addition sequences:



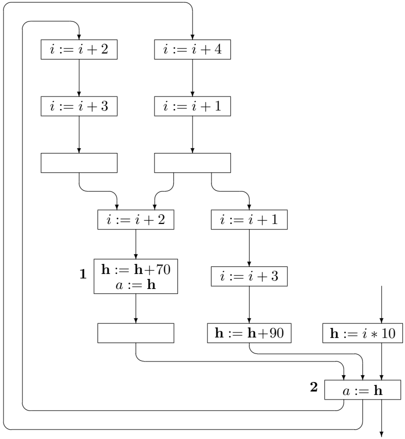
Avoiding Multiple-Addition Sequences (1)

...basic accumulation of the effect of cure instructions across basic blocks:

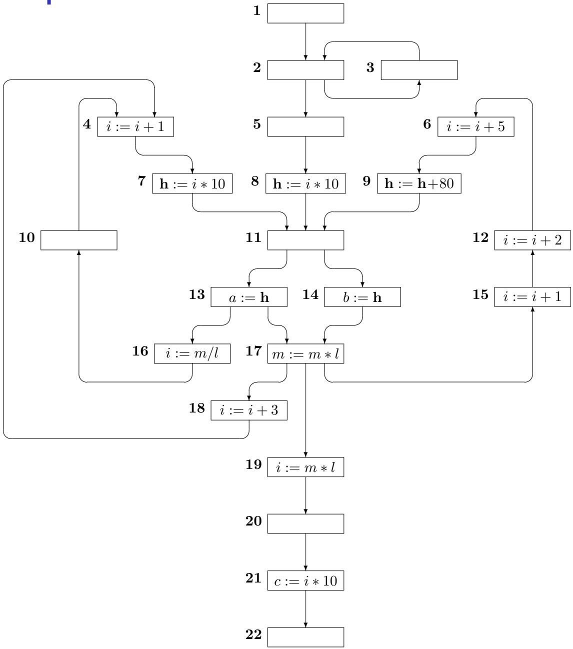


Avoiding Multiple-Addition Sequences (2)

...refined accumulation of the effect of cure instructions across extended basic blocks:



Running Example: The Result of *LSR*



Exercise

1. Specify the DFAs and transformations required for
   - BSR
   - BSR_{FstRef} (avoiding multiplication-addition sequences)
   - BSR_{SndRef} (avoiding unnecessary register pressure)
   - $LSR =_{df} BSR_{ThdRef}$ (avoiding multiple-addition sequ.)
2. Implement the DFAs in PAG.
3. Validate the DFAs on the running example of Appendix D (or an example coming close to it).

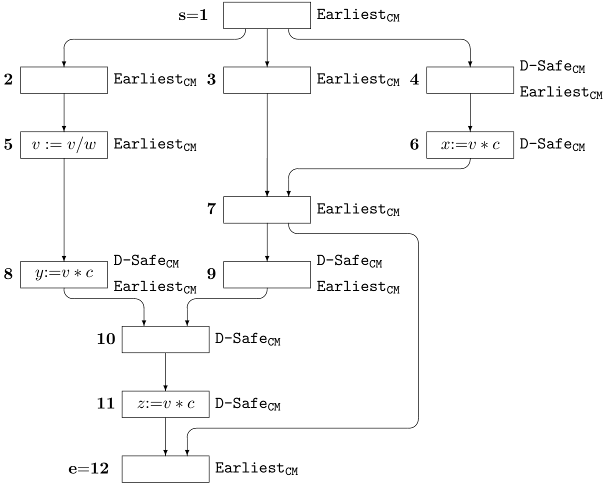
Running Example: Summary

...of the predicate values of the properties required for *LSR*:

Predicate	Node Number 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$Safe_{CM}$	1	1	1	0	1	0	1	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0
$Earliest_{CM}$	1	0	0	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0
$\mathtt{Insert}_{CM}$	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
$Safe_{SR}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	0
$Earliest_{SR}$	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
$\mathtt{Insert}_{SSR}$	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
$\mathtt{Insert}_{FstRef}$	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
$\mathtt{Update}_{SndRef}$	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	1	0
$\mathtt{Insert}_{SndRef}$	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
$\mathtt{Insert}_{LSR}$	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\mathtt{InsUpd}_{LSR}$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$\mathtt{Delete}_{LSR}$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Illustrating Down-Safety and Earliestness

...using a new, “not related”, example:



D.8
References, Further Reading

Further Reading for Appendix D (1)

- Frances E. Allen, John Cocke, Ken Kennedy. *Reduction of Operator Strength*. In Stephen S. Muchnick, Neil D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, Chapter 3, 79-101.
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 10.4.2, Strength Reduction)
- Dhananjay M. Dhamdhere. *A New Algorithm for Composite Hoisting and Strength Reduction Optimisation (+ Corrigendum)*. International Journal of Computer Mathematics 27:1-14&31-32, 1989.

Further Reading for Appendix D (2)

- Dhananjay M. Dhamdhere, J. R. Isaac. *A Composite Algorithm for Strength Reduction and Code Movement Optimization*. International Journal of Computer and Information Sciences 9(3):243-273, 1980.
- S. M. Joshi, Dhananjay M. Dhamdhere. *A Composite Hoisting-strength Reduction Transformation for Global Program Optimization – Part I and Part II*. International Journal of Computer Mathematics 11:21-41&111-126, 1982.
- Robert Kennedy, Fred C. Chow, Peter Dahl, Shing-Ming Liu, Raymond Lo, Mark Streich. *Strength Reduction via SSAPRE*. In Proceedings of the 7th International Conference on Compiler Construction (CC'98), Springer-V., LNCS 1383, 144-158, 1998.

Further Reading for Appendix D (3)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. Journal of Programming Languages 1(1):71-91, 1993.
- Bernhard Steffen, Jens Knoop, Oliver Rüthing. *Efficient Code Motion and an Adaption to Strength Reduction*. In Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), Springer-V., LNCS 494, 394-415, 1991.