Optimierende Compiler

LVA 185.A04, VU 2.0, ECTS 3.0
WS 2012/13

(Stand: 17.01.2013)

Jens Knoop

Technische Universität Wien
Institut für Computersprachen

Table of Contents

Table of Contents (1)

Table of Contents (2)

Table of Contents (3)

Part III: Interprocedural Data Flow Analysis

Table of Contents (4)

Part IV: Extensions, Other Settings

Table of Contents (5)

Part I

Introduction

Chapter 1

Motivation

See Separate Slide Package of Lecture 1.

Further Reading for Chapter 1

- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004. (Chapter 1, Overview of Compilation; Chapter 10, Scalar Optimizations)
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 1, Introduction to Advanced Topics)
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-Verlag, 2005. (Chapter 1, Introduction)

Chapter 2

Program Analysis

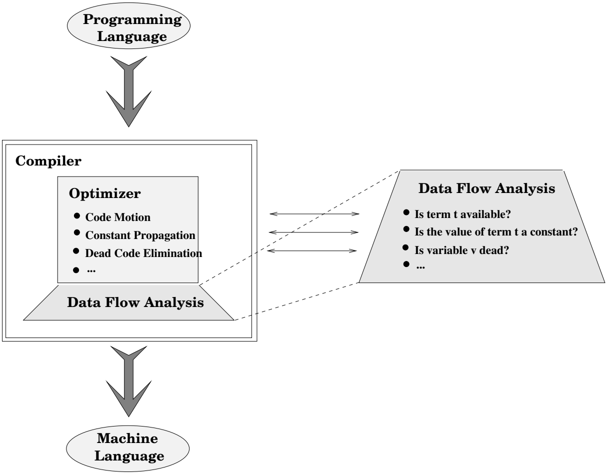
Typical Questions

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

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

Background

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



Essential Issues

comprise...

fundamental ones

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

as (apparently) minor ones:

- What is an appropriate and suitable program representation?

Outlook

In more detail we will distinguish:

- intraprocedural
- interprocedural
- parallel
- ...

data flow analysis (DFA).

Outlook (cont'd)

Ingredients of (intraprocedural) data flow analysis:

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

Theory of Intraprocedural DFA

Main Results:

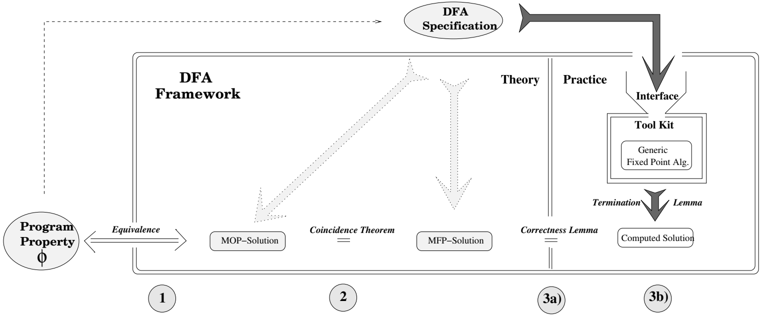
- Safety (Soundness) Theorem
- Coincidence (Completeness Theorem

Plus:

- Effectivity (Termination) Theorem

Practice of Intraprocedural DFA

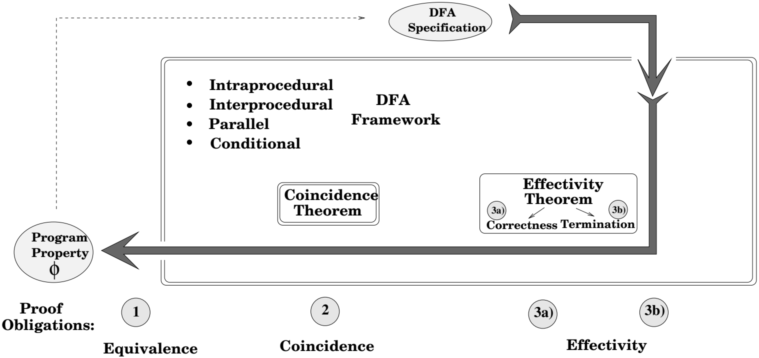
The Intraprocedural DFA Framework / DFA Toolkit View:



Practice of DFA

The constraint "intraprocedural" can be dropped.

The DFA Framework / DFA-Toolkit View holds generally:



Ultimate Goal

Optimal Program Optimization

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

There is no free Lunch!

In the diction of optimizing compilation:

...w/out analysis no optimization!

Chapter 3

First Examples

Chapter 3.1

Forward Analyses

See Separate Slide Package of Lecture 2.

Contents
Chap. 1
Chap. 2
Chap. 3
3.1
3.2
3.3
Chap. 4
Chap. 5
Chap. 6
Chap. 7
Chap. 8
Chap. 9
Chap. 10
Chap. 11
Chap. 12
Chap. 13
Bibliograp
Appendix
A

Chapter 3.2

Backward Analyses

See Separate Slide Package of Lecture 3.

Chapter 3.3

Framework

See Separate Slide Package of Lecture 4.

Further Reading for Chapter 3

Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-Verlag, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)

Part II

Intraprocedural Data Flow Analysis

Chapter 4

Program Representation

Programs as Flow Graphs

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

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

Flow Graphs

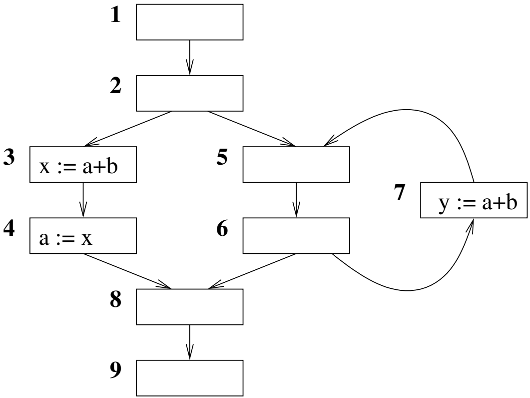
A (non-deterministic) flow graph is a 4-tuple $G = (N, E, s, e)$ with

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

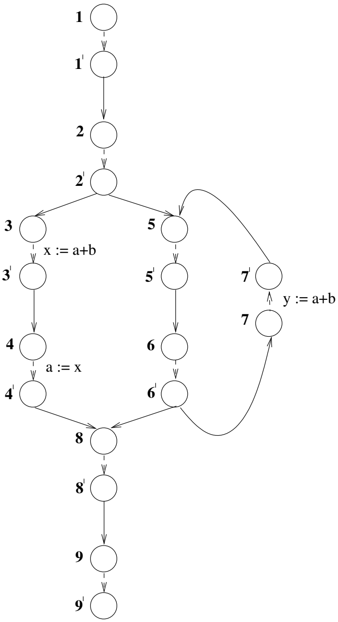
Nodes represent program points, edges represent the branching structure. Elementary program statements (assignments, tests) can be represented by

- either nodes ($\rightsquigarrow$ node labelled flow graph)
- or edges ($\rightsquigarrow$ edge labelled flow graph)

Example: A Node Labelled Flow Graph

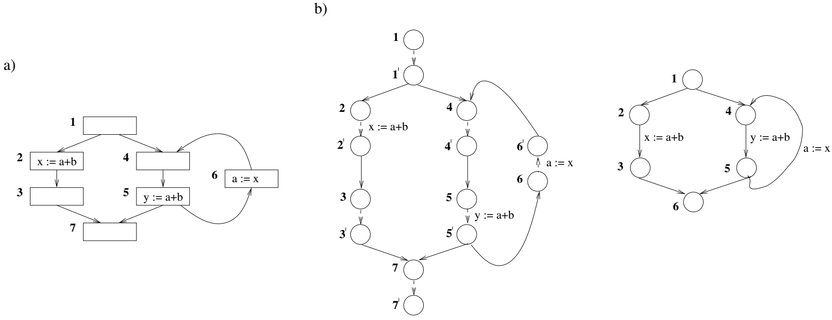


Example: An Edge Labelled Flow Graph



Flow Graphs: Single Instruction Variants

Node labelled vs. edge labelled single instruction flow graphs

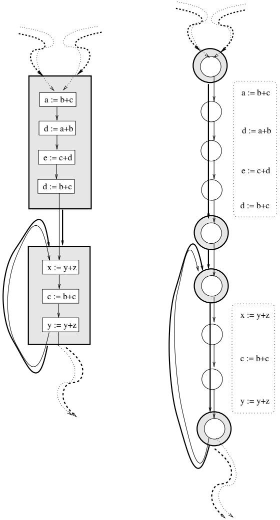


i) Schematisch

ii) "Optimiert"

Flow Graphs: Basic Block Variants

Node labelled vs. edge labelled basic block flow graphs



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

Summing up

We distinguish:

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

In the following we will preferably deal w/ edge labelled SI graphs.

Notations

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

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

Remark: If G is uniquely determined by the context, then we drop the index and simply write $\mathbf{P}$ instead of $\mathbf{P}_G$.

Chapter 5

The Intraprocedural DFA Framework

DFA Specification

- (Local) abstract semantics
  1. A data flow analysis lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$
  2. A data flow analysis functional $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$
- A start information (start assertion) $c_{\mathbf{s}} \in \mathcal{C}$

Globalizing a Local Abstract Semantics

Two Strategies:

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

Chapter 5.1

The *MOP* Approach

The *MOP* Approach

Essential:

Extending the local abstract semantics to paths:

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

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

The *MOP* Solution

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ \mathit{MOP}_{c_{\mathbf{s}}}(n) = \bigsqcap \{\, [\![\, p \,]\!](c_{\mathbf{s}}) \mid p \in \mathbf{P}[\mathbf{s}, n] \,\}$$

The *MOP* Solution: The specifying solution of the DFA problem given by $\mathcal{C}$, $[\![\ \ ]\!]$, and $c_{\mathbf{s}}$.

Unfortunately

The *MOP* solution is undecidable in general:

Theorem (5.1.1, Undecidabality)

(John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. Acta Informatica 7, 305-317, 1977)
There is no algorithm A satisfying:

1. *The input of A are*
   1.1 *algorithms for the computation of the meet, the equality test, and the application of functions on the lattice elements of a monotonic DFA framework*
   1.2 *an instance I of the framework given by $\mathcal{C}$, $[\![\ ]\!]$, and $c_{\mathbf{s}}$*
2. *The output of A is the MOP solution of I.*

Because of this negative result we introduce a second globalization strategy.

Chapter 5.2

The *MaxFP* Approach

The *MaxFP* Approach

Essential:

The *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 *MaxFP* Solution

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ \mathit{MaxFP}_{([\![\ ]\!], c_{\mathbf{s}})}(n) =_{df} \mathit{inf}\ {}^{*}_{c_{\mathbf{s}}}(n)$$

where $\mathit{inf}\ {}^{*}_{c_{\mathbf{s}}}$ denotes the greatest solution of the *MaxFP* equation system wrt $[\![\ ]\!]$ and $c_{\mathbf{s}}$.

The *MaxFP* Solution: The effectively computable solution of the DFA problem given by $\mathcal{C}$, $[\![\ ]\!]$, and $c_{\mathbf{s}}$, if these satisfy certain constraints.

The Generic Fixed Point Algorithm (1)

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

Output: The *MaxFP* solution, if the preconditions of the Effectivity Theorem hold (cf. Chap. 5.3). Depending on the properties of the DFA functional we have:
(1) $[\![\ ]\!]$ is distributive: The variable *inf* stores for each node the strongest post-condition wrt the start information $c_\mathbf{s}$.
(2) $[\![\ ]\!]$ is monotonic: The variable *inf* stores for each node a safe (i.e. lower) approximation of the strongest post-condition wrt the start information $c_\mathbf{s}$.

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

The Generic Fixed Point Algorithm (2)

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

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

$inf[\mathbf{s}] := c_{\mathbf{s}}$;

$workset := \{ \mathbf{s} \}$;

(Main loop: The iterative fixed point computation)

```
WHILE workset ≠ ∅ DO
    CHOOSE m ∈ workset;
        workset := workset \ { m };
        ( Update the annotations of all successors of node m )
        FORALL n ∈ succ(m) DO
            meet := [[ (m, n) ]](inf[m]) ⊓ inf[n];
            IF inf[n] ⊐ meet
                THEN
                    inf[n] := meet;
                    workset := workset ∪ { n }
            FI
        OD ESOOHC OD.
```

Some yet to be defined Notions

...related to the generic fixed point algorithm:

- Descending (ascending) chain condition
- Monotonicity and distributivity of a
  - local abstract semantic functions
  - DFA functional

Ascending and Descending Chain Condition

Definition (5.2.1, Ascending, Descending Chain Condition)

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

1. the ascending chain condition, if each ascending chain eventually gets stationary, i.e. for each chain $p_1 \sqsubseteq p_2 \sqsubseteq \ldots \sqsubseteq p_n \sqsubseteq \ldots$ there is an index $m \geq 1$ such that $x_m = x_{m+j}$ for all $j \in \mathbb{N}$
2. the descending chain condition, if every descending chain eventually gets stationary, i.e. for each chain $p_1 \sqsupseteq p_2 \sqsupseteq \ldots \sqsupseteq p_n \sqsupseteq \ldots$ there is an index $m \geq 1$ such that $x_m = x_{m+j}$ for all $j \in \mathbb{N}$

Monotonicity, Distributivity, Additivity

...of functions on (DFA) lattices.

Definition (5.2.2, Monotonicity, Distributivity, Additivity)

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

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

Often useful

...the following equivalent characterization of monotonicity:

Lemma (5.2.3)

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

$$f \text{ is monotonic} \iff \forall\, C' \subseteq \mathcal{C}.\ \ f(\bigsqcap C') \sqsubseteq \bigsqcap \{f(c) \mid c \in C'\}$$

Monotonicity, Distributivity, and Additivity

...of DFA functionals.

Definition (5.2.4)

A DFA functional $[\![\ ]\!] : E \to (\mathcal{C} \to \mathcal{C})$ is monotonic (distributive, additive) iff $\forall\, e \in E.\ [\![\ e\ ]\!]$ is monotonic (distributive, additive).

Chapter 5.3

Coincidence and Safety Theorem

Main Results: Soundness, Completeness, Effectivity (Termination)

The relationship of

- *MOP* and *MaxFP* Solution
  - Soundness
  - Completeness
- *MaxFP* solution and generic algorithm
  - Termination with the *MaxFP* solution

Soundness

Theorem (5.3.1, Safety)

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

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ \mathit{MaxFP}_{c_{\mathbf{s}}}(n) \sqsubseteq \mathit{MOP}_{c_{\mathbf{s}}}(n)$$

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

Completeness (and Soundness)

Theorem (5.3.2, Coincidence)

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

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ \mathit{MaxFP}_{c_{\mathbf{s}}}(n) = \mathit{MOP}_{c_{\mathbf{s}}}(n)$$

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

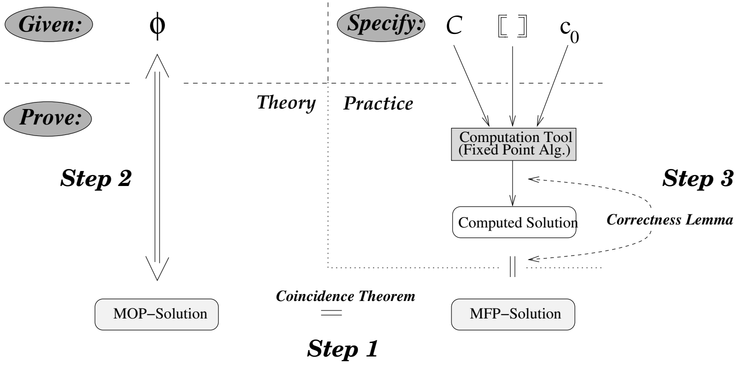
Effectivity (Termination)

Theorem (5.3.3, Effectivity (Termination))

The generic fixed point algorithm terminates with the MaxFP solution, if the DFA functional is monotonic and the DFA lattice satisfies the descending chain condition.

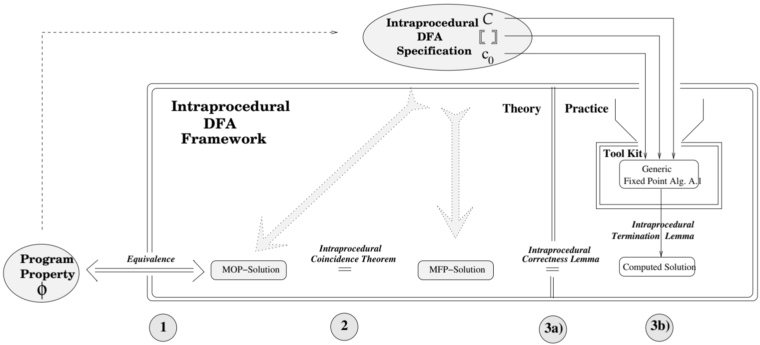
Overview on Intraprocedural DFA (1)

The schematic view:



Overview on Intraprocedural DFA (2)

Focused on the framework/toolkit view:



Chapter 5.4

Two Examples: Available Expressions and Simple Constants

Two Prototypical DFA Problems

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

Chapter 5.4.1

Available Expressions

Available Expressions

...a typical distributive DFA problem.

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

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

Notations

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

...and for a fixed candidate expression t:

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

Main Results

Lemma (5.4.1.1)

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

Corollary (5.4.1.2)

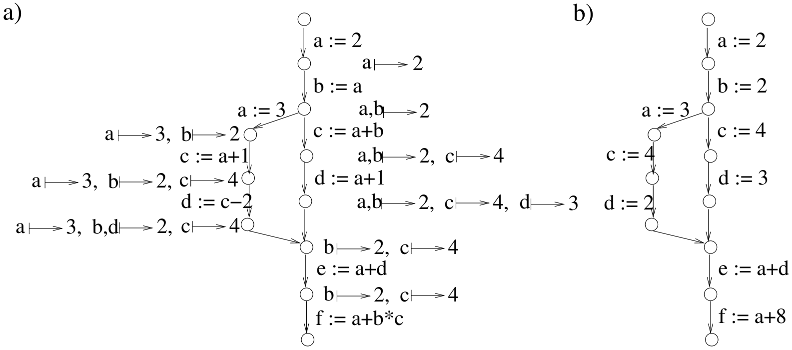
The MOP solution and the MaxFP solution coincide for available expressions.

Chapter 5.4.2

Simple Constants

Simple Constants

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



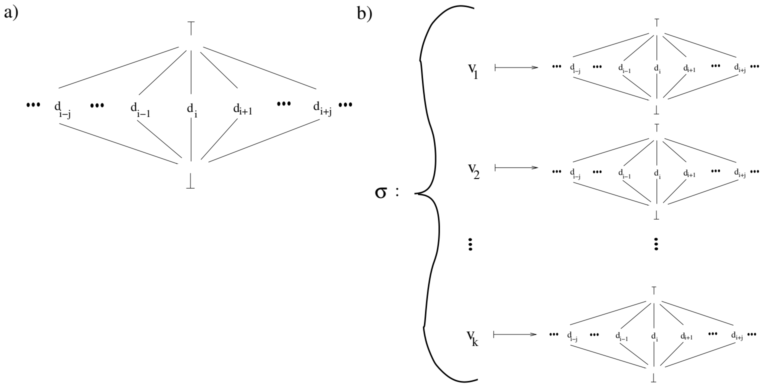
Abstract Semantics for Simple Constants

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

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

DFA Lattice for Simple Constants

The “canonical” lattice for constant propagation and folding:



The Semantics of Terms

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

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

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

Some Yet to be defined Notions

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

- Term syntax
- Interpretation
- State

The Syntax of Terms (1)

Let

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

The Syntax of Terms (2)

We legen fest:

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

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

Interpretation

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

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

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

Set of States

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

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

In particular

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

The State Transformation Function

The state transformation function

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

is defined by:

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

Main Results

Lemma (5.4.2.1)

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

Note: Distributivity does not hold! (Excercise)

Corollary (5.4.2.2)

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

Further Reading for Chapter 5 (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007. (Chapter 1, Introduction; Chapter 9.2, Introduction to Data-Flow Analysis; Chapter 9.3, Foundations of Data-Flow 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 1, Overview of Compilation; Chapter 8, Introduction to Code Optimization; Chapter 9, Data Flow Analysis)

Further Reading for Chapter 5 (2)

- Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
- John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. Acta Informatica 7:305-317, 1977.
- Gary A. Kildall. *A Unified Approach to Global Program Optimization*. In Conference Record of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73), 194-206, 1973.
- 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-Verlag, LNCS 1579, 360-374, 1999.

Further Reading for Chapter 5 (3)

- Janusz Laski, William Stanley. *Software Verification and Analysis*. Springer-Verlag, 2009. (Chapter 7, What can one tell about a Program without its Execution: Static Analysis)
- Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997. (Chapter 1, Introduction to Advanced Topics; Chapter 4, Intermediate Representations; Chapter 7, Control-Flow Analysis; Chapter 8, Data Flow Analysis; Chapter 11, Introduction to Optimization; Chapter 12, Early Optimizations)

Further Reading for Chapter 5 (4)

- 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-Verlag, 2007. (Chapter 7, Program Analysis; Chapter 8, More on Program Analysis; Appendix B, Implementation of Program Analysis)
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-Verlag, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)

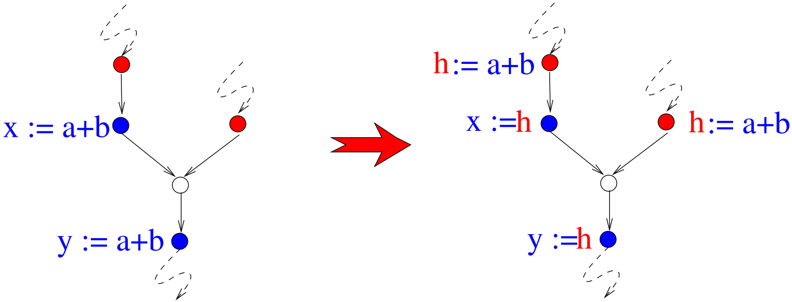
Chapter 6

Partial Redundancy Elimination

Partial Redundancy Elimination (PRE)

What's it all about?

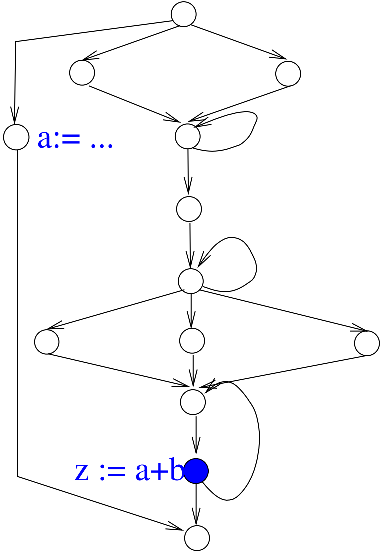
...avoiding multiple (re-) computations of the same value!



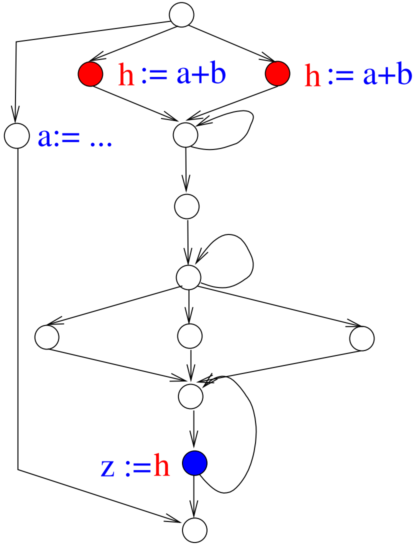
Chapter 6.1

Motivation

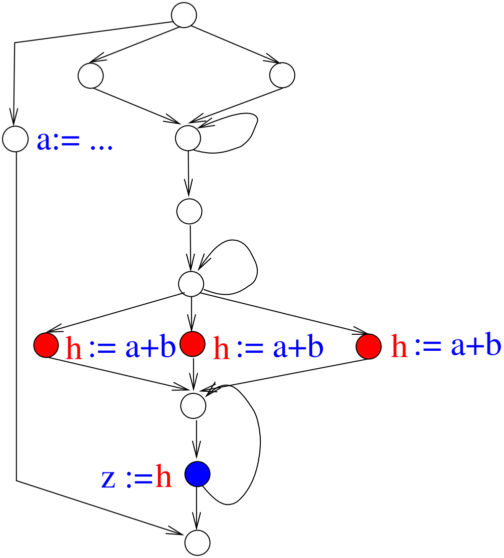
PRE – Particularly Striking for Loops



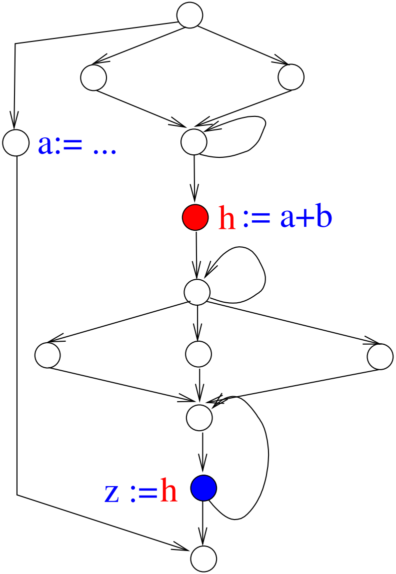
A Program w/out Redundancies at all



Often there is more than one!



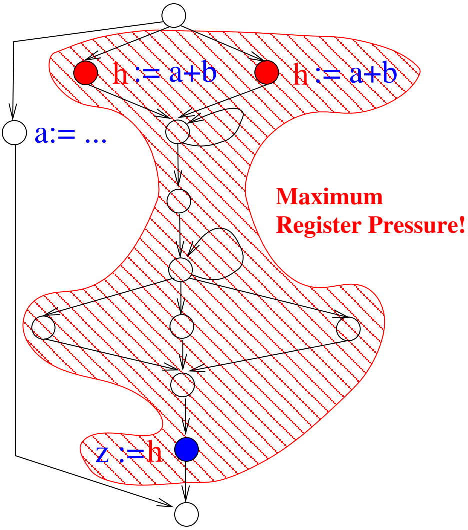
Which one shall PRE deliver?



The (Optimization) Goals make the Difference!

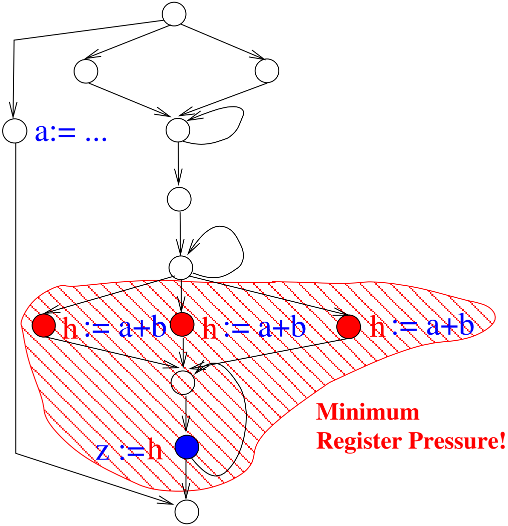
The first Transformation

...no redundancies but maximum register pressure!



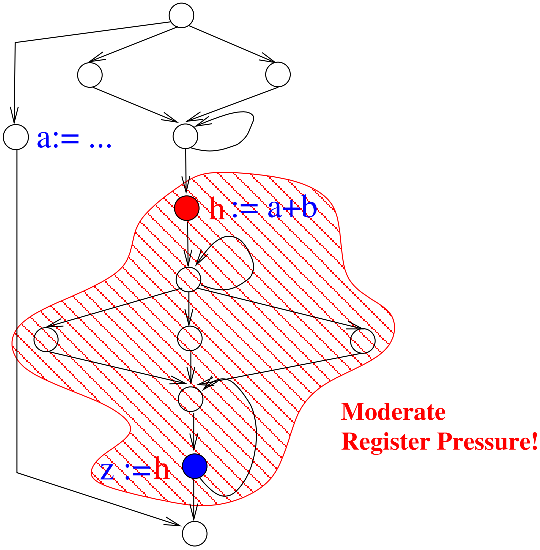
The second Transformation

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



The third Transformation

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



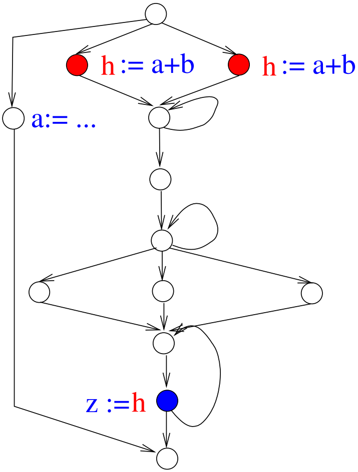
The (Optimization) Goals make the Difference!

In our running example:

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

The Result of Busy Code Motion

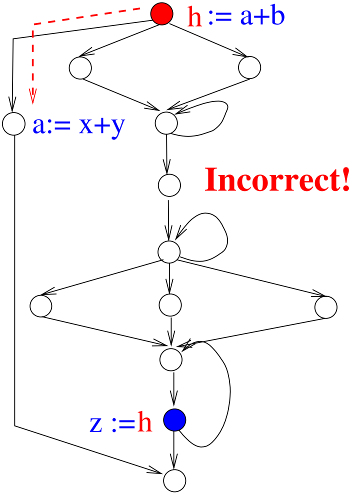
...placing computations as early as possible!



...yields computationally optimal programs.

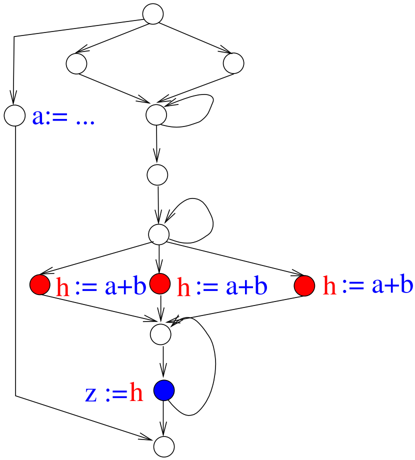
Note: As Early as Possible

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



The Result of Lazy Code Motion

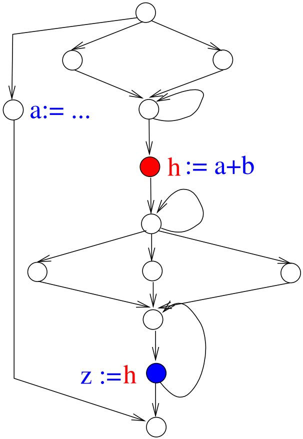
...placing computations as late as possible!



...yields computationally and lifetime optimal programs.

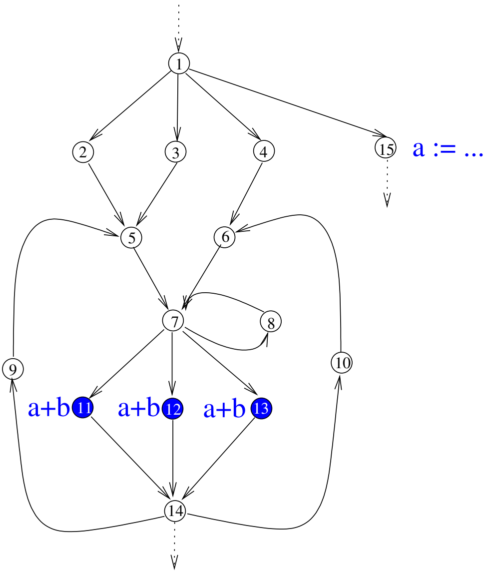
The Result of Sparse Code Motion

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

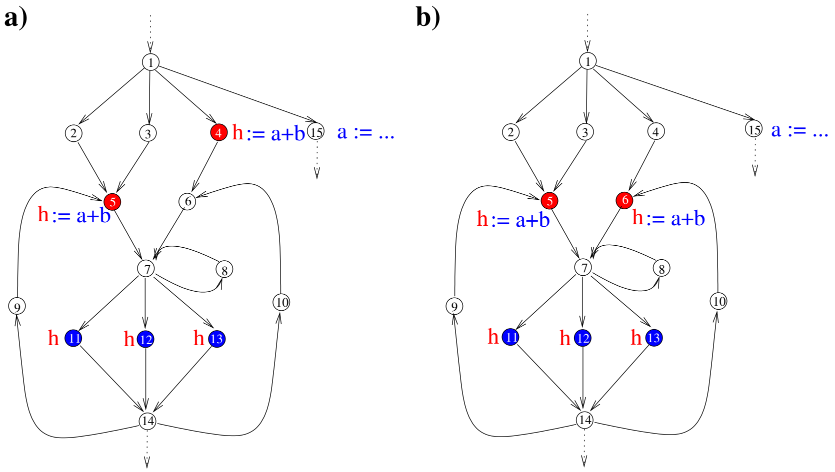


...yields comp. and lifetime best code-size optimal programs.

A More Complex Example (1)



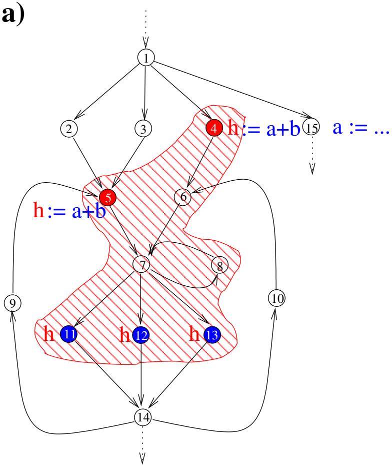
A More Complex Example (2)



Two Code–size Optimal Programs

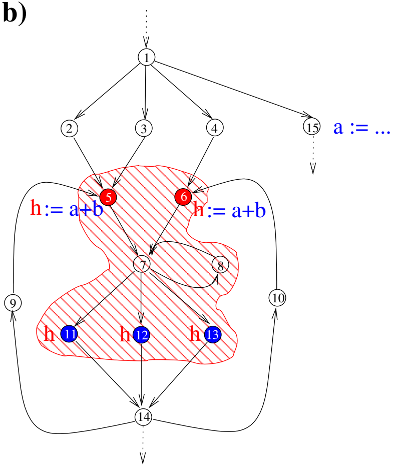
A More Complex Example (3)

a)



SQ > CQ > LQ

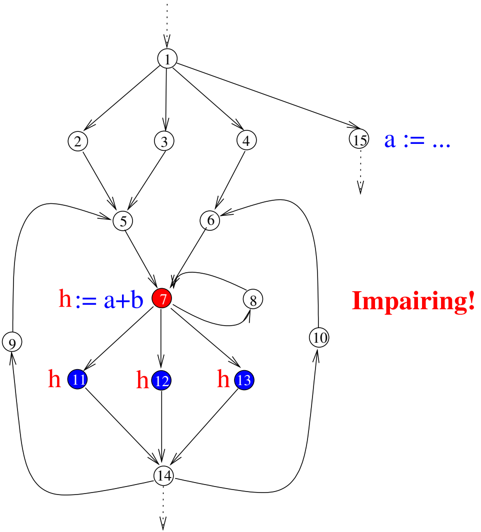
b)



SQ > LQ > CQ

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, lifetime, and space optimality

at the same time.

But think about the following (homework):

- Let P be a program containing partially redundant computations.

  Is it always possible to transform P into a program P' such that P and P' have the same semantics and that P' is free of any partially redundant computation?

Chapter 6.2

The PRE Algorithm of Morel&Renvoise

The Groundbreaking PRE Algorithm of Morel and Renvoise

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

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

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

The PRE Algorithm of Morel&Renvoise (1)

- Availability:

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

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

The PRE Algorithm of Morel&Renvoise (2)

- Very Busyness (Anticipability):

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

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

The PRE Algorithm of Morel&Renvoise (3)

- Partial Availability:

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

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

The PRE Algorithm of Morel&Renvoise (4)

- Placement Possible:

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

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

where

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

The PRE Algorithm of Morel&Renvoise (5)

- Initializing temporaries where:

$$\begin{aligned}\textbf{INSIN}(n) &=_{df} \textbf{false}\\[1em] \textbf{INSOUT}(n) &=_{df} \textbf{PPOUT}(n) * \neg\textbf{AVOUT}(n) * \\ &\quad\; (\neg\textbf{PPIN}(n) + \neg\mathrm{TRANSP}(n))\end{aligned}$$

- Replacing original computations where:

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

Summing up (1)

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

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

Summing up (2)

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

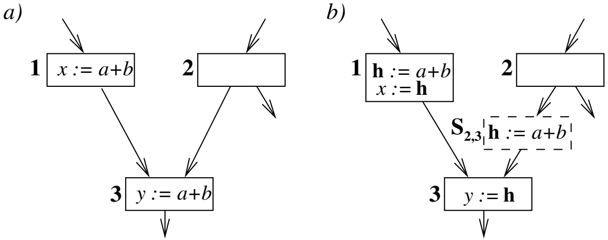
- Conceptually
  - Fails computational optimality
    $\rightsquigarrow$ only, however, because of not splitting critical edges
  - Fails lifetime optimality
    $\rightsquigarrow$ Register pressure is heuristically dealt with
  - Fails code-size optimality
    $\rightsquigarrow$ Not considered at all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

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

Critical Edges

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

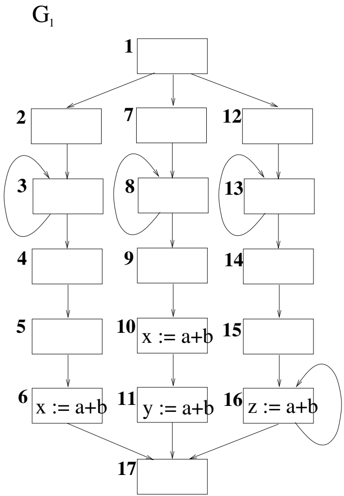
Illustration:

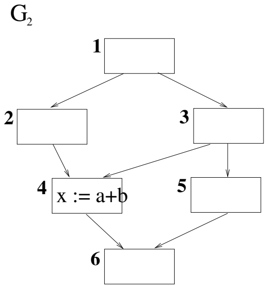


...by introducing the synthetic node $\mathbf{S}_{2,3}$, the critical edge from node **2** to node **3** is split.

Instructive

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





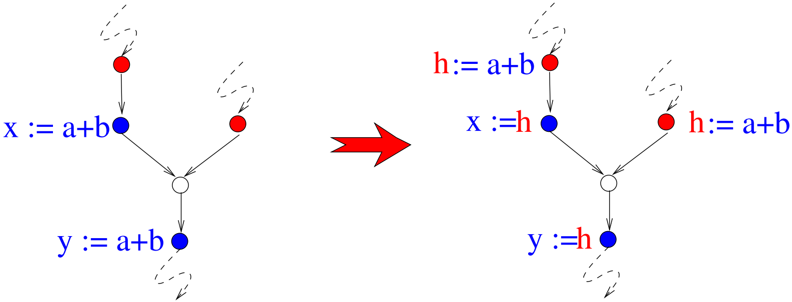
Chapter 6.3

Formalizing Code Motion

Partial Redundancy Elimination (PRE)

The very idea:

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



Notations (1)

Let $G = (N, E, \mathbf{s}, \mathbf{e})$ be a flow graph. Then:

- $pred(n) =_{df} \{m \mid (m, n) \in E\}$: The set of all predecessors
- $succ(n) =_{df} \{m \mid (n, m) \in E\}$: The set of all successors
- $source(e)$, $dest(e)$: Start node and end node of an edge
- Finite Path: A sequence of edges $(e_1, \ldots, e_k)$ such that $dest(e_i) = source(e_{i+1})$ for all $1 \leq i < k$
- Instead of edge sequences we also consider node sequences as paths, where reasonable.

Notations (2)

- $p = \langle e_1, \ldots, e_k \rangle$ path from m to n, if $source(e_1) = m$ and $dest(e_k) = n$
- $\mathbf{P}[m, n]$: The set of all paths from m to n
- λ_p: The length of p, i.e., the number of edges of p
- ε: The path of length 0
- $N_J \subseteq N$: The set of join nodes, i.e., the set of nodes w/ more than one predecessor
- $N_B \subseteq N$: The set of branch nodes, i.e. the set of nodes w/ more than one successor

Convention

W/out losing generality we assume:

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

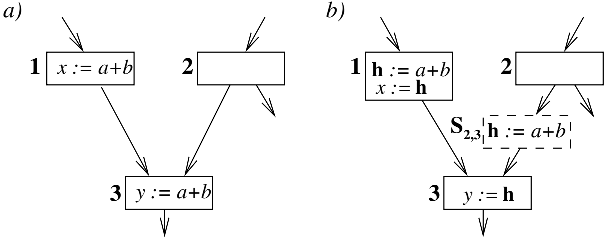
  Intuition: There are no unreachable parts within a flow graph.

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

Reminder: Critical Edges

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

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



A PRE specific Convention

W/out losing generality we consider in the following flow graphs that are given

- as node labelled SI graphs,
- where all edges ending in a join node are split by inserting a so-called synthetic node,

...this is a PRE specific assumption.

Background

...of this convention:

- The PRE process becomes simpler.

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

Remark

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

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

Note that this is not a problem at all. Agreeing, however, on the above assumption simplifies the presentation of the PRE algorithm even more.

Work Plan

In the following we will define:

- The set of PRE transformations
- The set of admissible PRE transformations
- The set of computationally optimal PRE transformations
- The *BCM* transformation as a specific computationally optimal PRE transformation
- The *LCM* transformation as the one and only computationally and lifetime optimal PRE transformation

The Set of PRE Transformations

The generic (transformation) pattern for a term t:

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

Remark: t is often called a candidate expression.

Observation

Two predicates (defined on nodes)

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

suffice to specify a PRE (resp. CM) transformation completely (note: the step of declaring the temporary h is the same for each CM transformation and thus does not need to be considered explicitly).

CM Transformations

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

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

Observation

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

This leads us to the notion of admissible CM transformations.

Admissible CM Transformations

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

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

Intuitively:

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

Formalising this

...requires two (local) predicates:

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

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

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

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

Extending Predicates to Paths

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

Then we define:

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

Safety and Correctness

Definition (6.3.1, Safety and Correctness)

Let $n \in N$. We define:

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

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

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

Intuition

A computation of t at program point n is

- up-safe, if t is computed on all paths p from **s** to n and the last computation of t on p is not followed by a modification of (an operand of) t.
- down-safe, if t is computed on all paths p from n to **e** and the first computation of t on p is not preceded by a modification of (an operand of) t.

Up-Safety and Down-Safety

Definition (6.3.2, Up-Safety and Down-Safety)

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

Admissible CM-Transformations

This allows us to define:

Definition (6.3.3, Admissible CM-Transformation)

A CM-transformation $CM \in \mathcal{CM}$ is admissible iff for every node $n \in N$ holds:

1. $Insert_{CM}(n) \Rightarrow Safe(n)$
2. $Repl_{CM}(n) \Rightarrow Correct_{CM}(n)$

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

First Results (1)

Lemma (6.3.4, Correctness)

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

First Results (2)

Lemma (6.3.5, Safety)

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

Computationally Better

Definition (6.3.6, Computationally Better)

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

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

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

Computational Optimality

Definition (6.3.7, Computationally Optimal CM-Transformation)

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

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

Conceptually

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

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

Chapter 6.4

Busy Code Motion

The Earliestness Principle

...induces an extreme placing strategy:

Placing computations as early as possible...

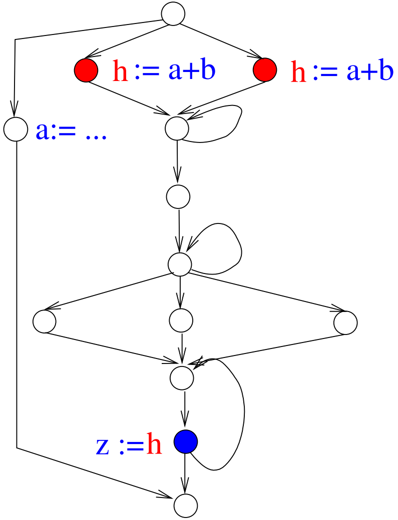
- Theorem (Computational Optimality)
  ...hoisting computations to their earliest safe computation points yields computationally optimal programs.

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

Earliestness Principle

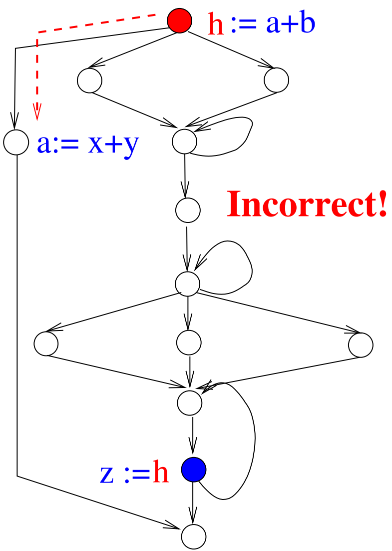
Placing computations as early as possible...

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 w/out violating safety and correctness!

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

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

Earliestness

Definition (6.4.1, Earliestness)

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

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

The *BCM* Transformation

The *BCM* Transformation:

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

The *BCM*-Theorem

Theorem (6.4.2, *BCM*-Theorem)

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

The proof of the *BCM*-Theorem 6.4.2 relies on the Earliestness Lemma 6.4.3 and the *BCM*-Lemma 6.4.4.

The Earliestness Lemma

Lemma (6.4.3, Earliestness Lemma)

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

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

The *BCM*-Lemma

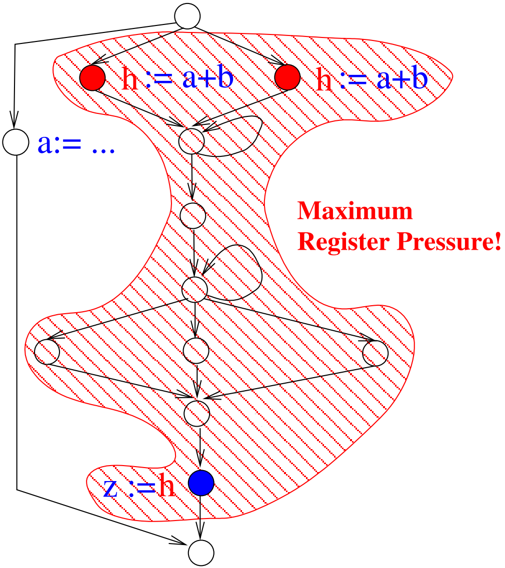
Lemma (6.4.4, *BCM*-Lemma)

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

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

The *BCM*-Transformation

...computationally optimal, but maximum register pressure.



Chapter 6.5

Lazy Code Motion

The Latestness Principle

...induces a dual extreme placing strategy:

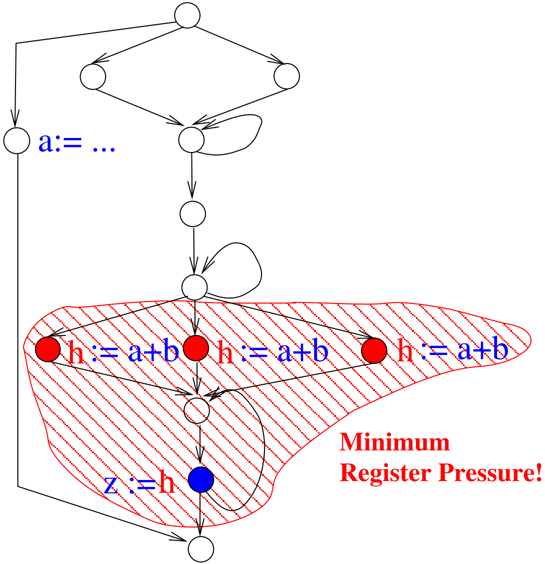
Placing computations as late as possible...

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

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

The *LCM*-Transformation

...computationally optimal w/ minimum register pressure!



Lazy Code Motion

Intuitively:

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

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

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

Work Plan

Next we will define:

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

Central for the Formalization

...is the notion of lifetime ranges.

Definition (6.5.1, 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 Results

Lemma (6.5.2, First-Use Lifetime-Range Lemma)

Let $CM \in \mathcal{CM}$, $p \in \mathbf{P}[s,e]$, and let i_1, i_2, j_1, j_2 indexes such that $p[i_1,j_1] \in$ FU-LtRg(CM) and $p[i_2,j_2] \in$ FU-LtRg(CM). Then we have:

- *either $p[i_1,j_1]$ and $p[i_2,j_2]$ coincide, i.e., $i_1 = i_2$ and $j_1 = j_2$, or*
- *$p[i_1,j_1]$ and $p[i_2,j_2]$ are disjoint, i.e., $j_1 < i_2$ or $j_2 < i_1$.*

Lifetime Better

Definition (6.5.3, 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.

Lifetime Optimality

Definition (6.5.4, Lifetime Optimal CM-Transformation)

A computationally optimal CM-transformation $CM \in \mathcal{CM}_{CmpOpt}$ is lifetime optimal iff CM is lifetime better than every other computationally optimal CM-transformation.

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

Reminder: Sets and 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

Uniqueness of Lifetime Optimal PRE

Obviously we have:

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

Even more we have:

Theorem (6.5.5, Uniqueness of Lifetime Optimal CM-Transformations)

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

Towards the *LCM* Transformation

We have:

Lemma (6.5.6)

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

Intuitively:

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

Delayability

Definition (6.5.7, Delayability)

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

The Delayability Lemma

Lemma (6.5.8, Delayability Lemma)

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

Latestness

Definition (6.5.9, Latestness)

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

The Latestness Lemma

Lemma (6.5.10, Latestness Lemma)

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

The *ALCM* Transformation

The "Almost Lazy Code Motion" Transformation:

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

Almost Lifetime Optimal

Definition (6.5.11, Almost Lifetime Optimal CM-Transformation)

A computationally optimal CM-transformation $CM \in \mathcal{CM}_{CmpOpt}$ is almost lifetime optimal iff

$$\forall\, p \in LtRg(CM).\ \lambda_p \geq 2 \Rightarrow$$
$$\forall\, CM' \in \mathcal{CM}_{CmpOpt}\ \exists\, q \in LtRg(CM').\ p \sqsubseteq q$$

We denote the set of all almost lifetime optimal CM-transformations by $\mathcal{CM}_{ALtOpt}$.

The *ALCM*-Theorem

Theorem (6.5.12, *ALCM*-Theorem)

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

Isolated Computations

Definition (6.5.13, *CM*-Isolation)

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

The Isolation Lemma

Lemma (6.5.14, 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:

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

The *LCM*-Theorem

Theorem (6.5.15, *LCM*-Theorem)

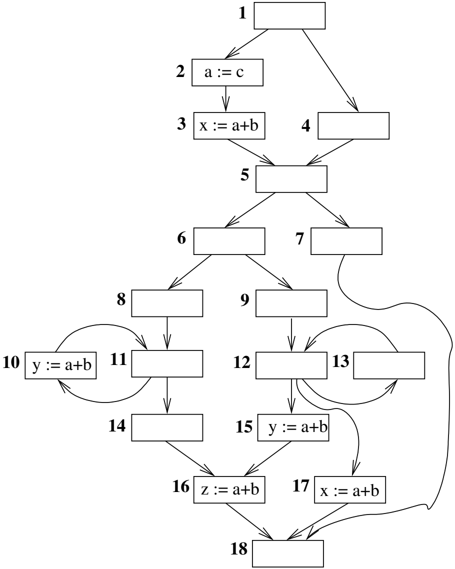
The LCM transformation is lifetime optimal, i.e., $LCM \in \mathcal{CM}_{LtOpt}$.

Chapter 6.6

An Extended Example

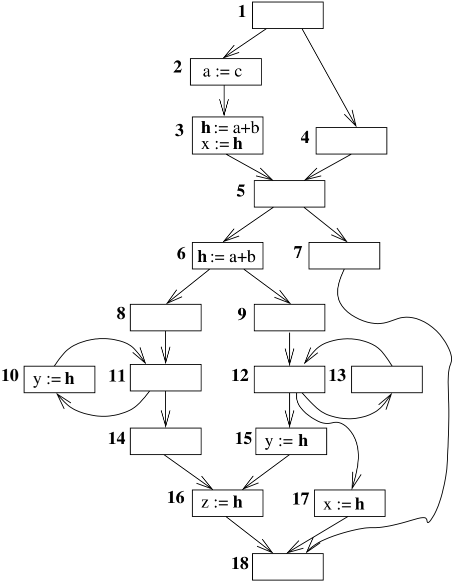
An Extended Example for Illustration (1)

The original program:



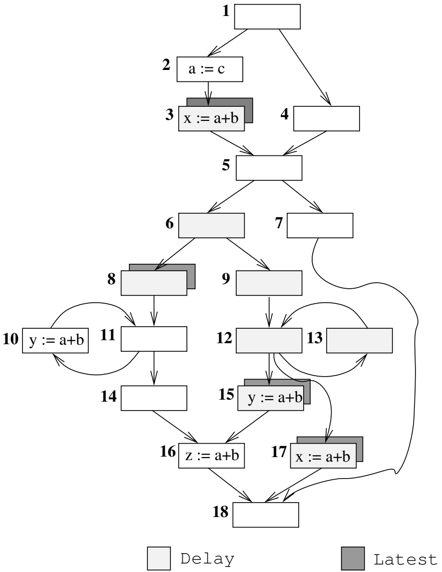
An Extended Example for Illustration (2)

The result of the *BCM* transformation:



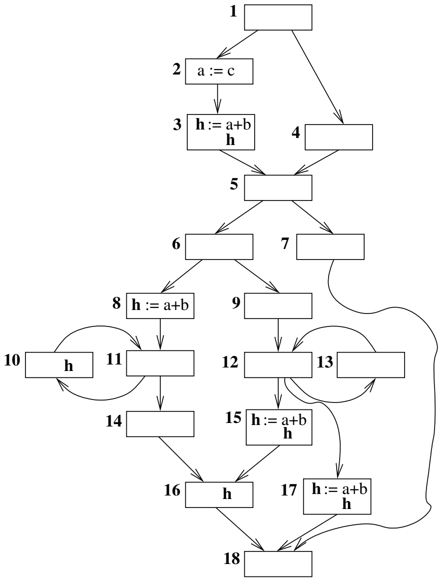
An Extended Example for Illustration (3)

Delayed and latest computation points:



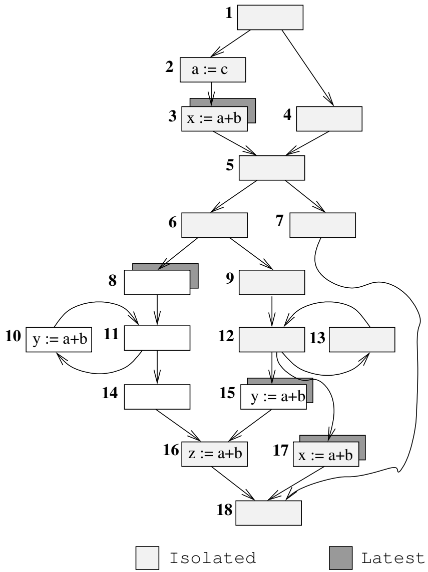
An Extended Example for Illustration (4)

The result of the *ALCM* transformation:



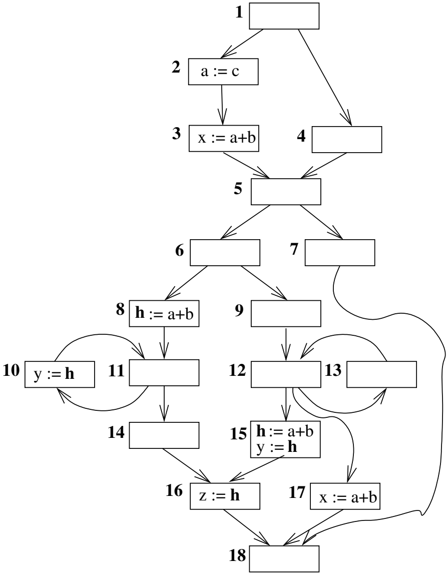
An Extended Example for Illustration (5)

Latest and isolated computation points...



An Extended Example for Illustration (6)

The result of the *LCM* transformation:



Chapter 6.7

Implementing Busy and Lazy Code Motion

Chapter 6.7.1

Implementing *BCM* on SI-Graphs

Implementing the BCM_ι Transformation

...on the level of single-instructions, here for node-labelled SI-graphs.

Note: For the following we assume that only critical edges are split. Therefore, the algorithm requires insertions at both node entries and node exits (N-insertions and X-insertions).

Busy Code Motion: BCM_ι (1)

1. Analyses for Up-Safety and Down-Safety

Local Predicates:

- $\text{COMP}_\iota(t)$: ι computes t.
- $\text{TRANSP}_\iota(t)$: ι does not modify an operand of t.

Busy Code Motion: BCM_ι (2)

The Equation System for Up-Safety:

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

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

Busy Code Motion: BCM_ι (3)

The Equation System for Down-Safety:

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

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

Busy Code Motion: BCM_ι (4)

2. The Transformation: Insertion and 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.

Busy Code Motion: BCM_ι (5)

The BCM_ι Transformation:

$$\text{N-INSERT}_\iota^{\text{BCM}} =_{df} \text{N-DSAFE}_\iota^\star \cdot \prod_{\hat{\iota} \in pred(\iota)} (\overline{\text{X-USAFE}_{\hat{\iota}}^\star + \text{X-DSAFE}_{\hat{\iota}}^\star})$$

$$\text{X-INSERT}_\iota^{\text{BCM}} =_{df} \text{X-DSAFE}_\iota^\star \cdot \overline{\text{TRANSP}_\iota}$$

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

Chapter 6.7.2

Implementing *BCM* on BB-Graphs

Implementing the BCM_β Transformation

...on the level of basic blocks, here for node-labelled BB-graphs.

Note: For the following we assume that (1) only critical edges are split. Therefore, the algorithm requires 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.

t-Refined Flow Graphs

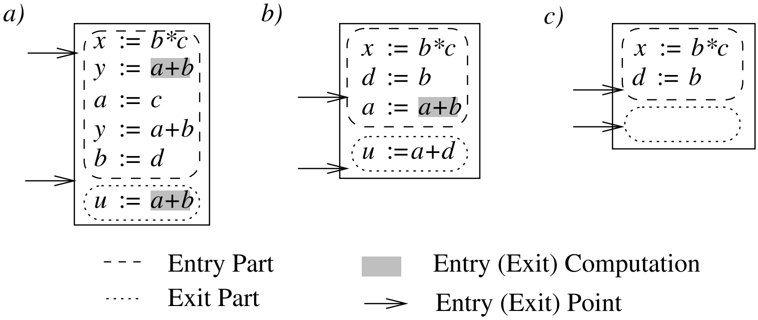
Given a computation t, a basic block $\mathbf{n}$ can be divided into two parts:

- 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: a non-empty basic block has also a non-empty entry part; in distinction to that the exit part can be empty (for illustration consider the following figure).

Entry and Exit Parts of a Basic Block

Illustrating the entry and exit part of a basic block:



Busy Code Motion: BCM_β (1)

1. Analyses for Up-Safety and Down-Safety

Local Predicates:

- BB-NCOMP$_\beta(t)$: β contains a statement ι that computes t, and that is not preceded by a statement that modifies an operand of t.
- BB-XCOMP$_\beta(t)$: β contains a statement ι that computes t and neither ι nor any other statement of β after ι modifies an operand of t.
- BB-TRANSP$_\beta(t)$: β contains no statement that modifies an operand of t.

Busy Code Motion: BCM_β (2)

The Equation System for Up-Safety:

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

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

Busy Code Motion: BCM_β (3)

The Equation System for Down-Safety:

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

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

Busy Code Motion: BCM_β (4)

2. The Transformation: Insertion and Replacement Points

Local Predicates:

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

Busy Code Motion: BCM_β (5)

The BCM_β Transformation:

$$\text{N-INSERT}_\beta^{\text{BCM}} \;=_{df}\; \text{BB-N-DSAFE}_\beta^\star \cdot \prod_{\hat\beta \in pred(\beta)} (\,\overline{\text{BB-X-USAFE}_{\hat\beta}^\star + \text{BB-X-DSAFE}_{\hat\beta}^\star}\,)$$

$$\text{X-INSERT}_\beta^{\text{BCM}} \;=_{df}\; \text{BB-X-DSAFE}_\beta^\star \cdot \overline{\text{BB-TRANSP}_\beta}$$

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

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

Chapter 6.7.3
Implementing *LCM*

The Equation Systems for *LCM*

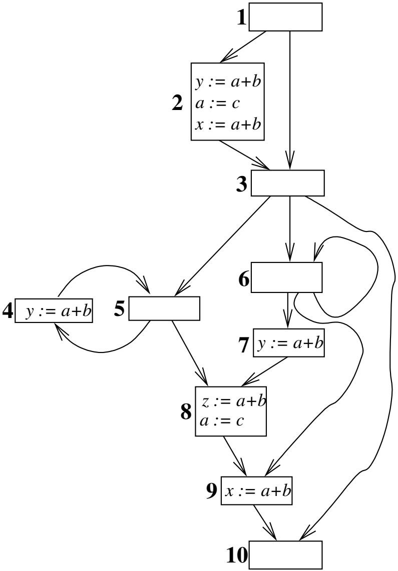
Quite similar! Homework!

Chapter 6.7.4

An Extended Example

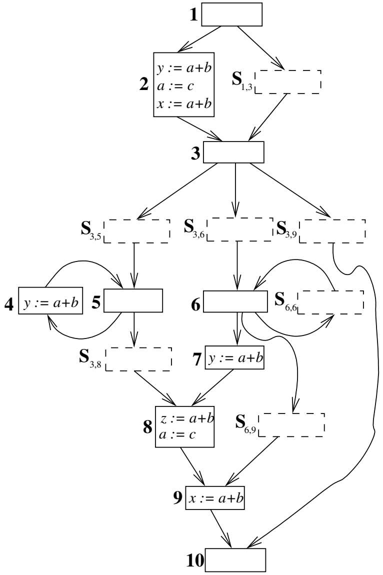
An Extended BB-Example for Illustration (1)

The original program:



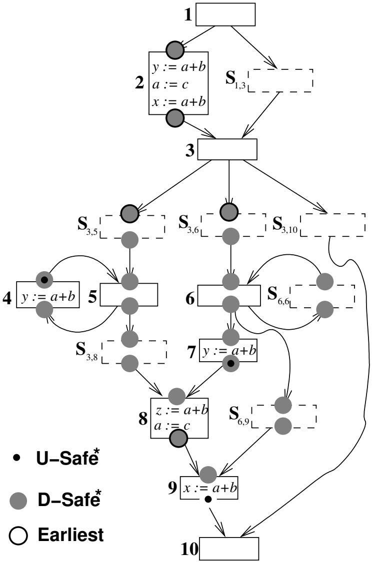
An Extended BB-Example for Illustration (2)

The original program after splitting of critical edges:



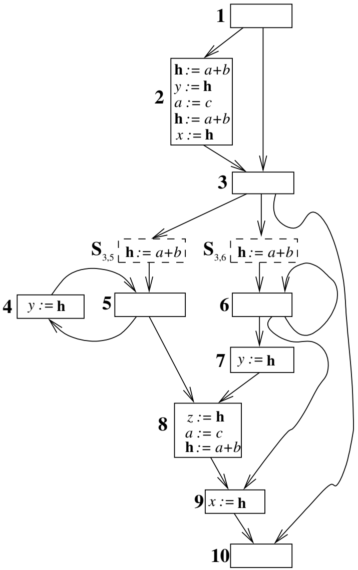
An Extended BB-Example for Illustration (3)

Earliest computation points:



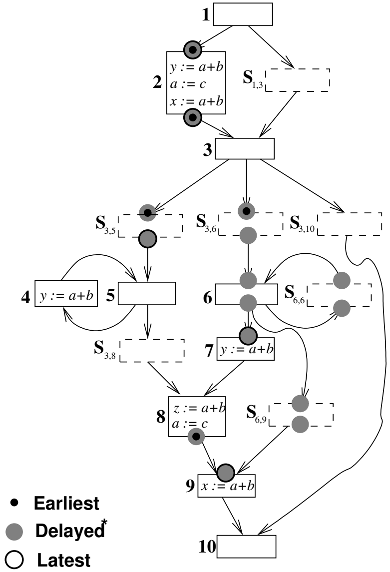
An Extended BB-Example for Illustration (4)

The result of the BCM_β transformation:



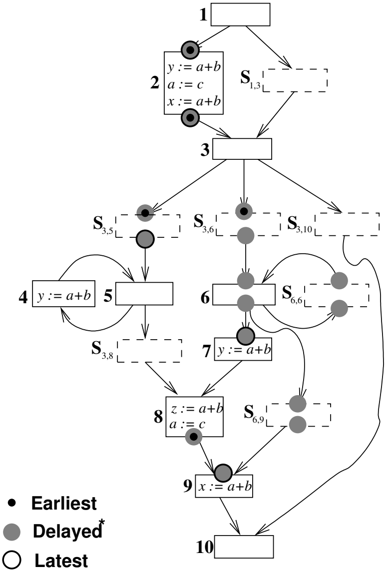
An Extended BB-Example for Illustration (5)

Latest computation points:



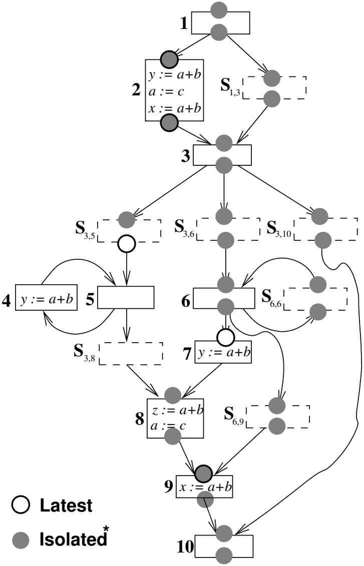
An Extended BB-Example for Illustration (6)

The result of the $ALCM_\beta$-transformation:



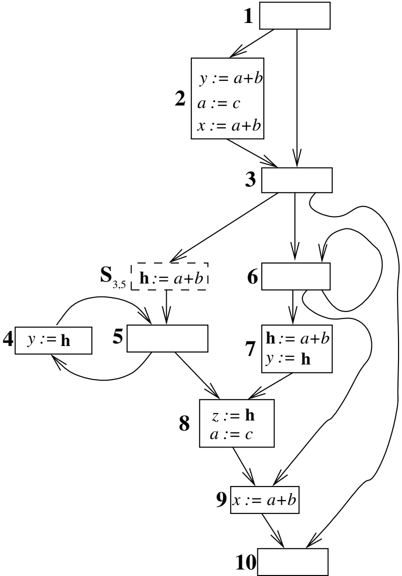
An Extended BB-Example for Illustration (7)

Isolated program points:



An Extended BB-Example for Illustration (8)

The result of the LCM_β transformation:



Chapter 6.8

Sparse Code Motion

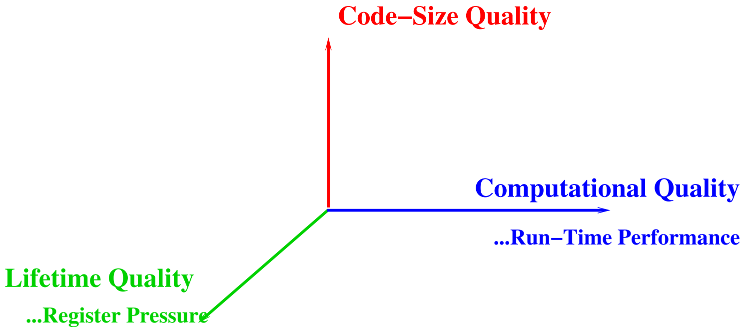
These days

...Lazy Code Motion is the

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

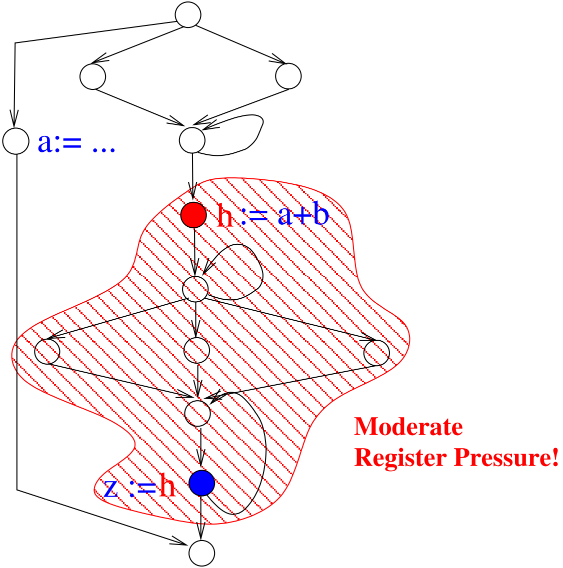
In the following

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



In the following (cont'd)

...to also render the below transformation possible:



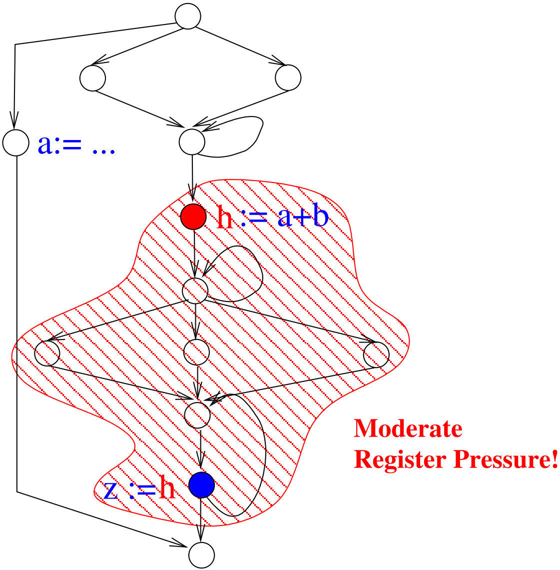
There is more than speed!

There is more than speed!

...for instance space!

There is more than speed!

...for instance space!



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,...
- Automative field
  - GPS navigation, airbags,...
- ...

Code for Embedded Systems

Demands:

- Performance (often real-time demands)
- Code size (system-on-chip, on-chip RAM/ROM)
- ...

For embedded systems:

- Code size is often more critical than speed!

Code for Embedded Systems (Cont'd)

Demands (and how they are often addressed):

- Assembler programming
- Manual post-optimization

Shortcomings:

- Error prone
- Delayed time-to-market

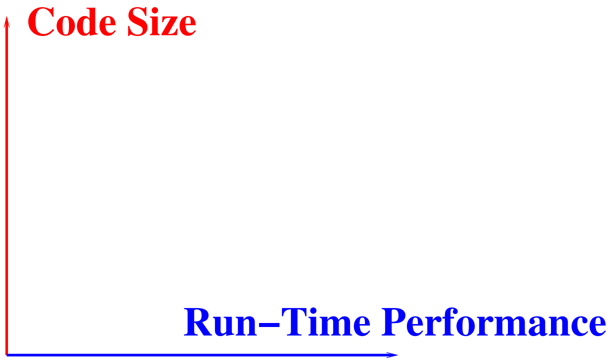
...problems which become greater with increasing complexity.

Generally, we observe:

- a trend towards high-level languages programming (C/C++)

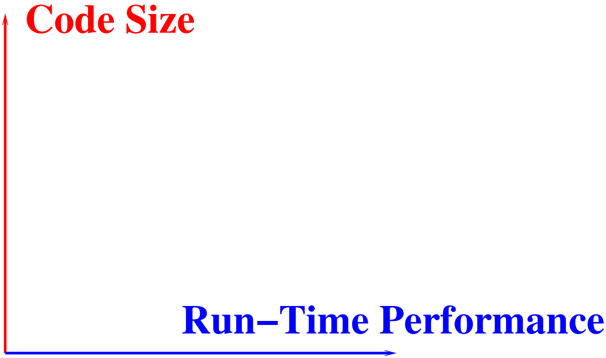
In Face of this Trend

...how do traditional compiler and optimizer technologies support the specific demands of code for embedded systems?



In Face of this Trend

...how do traditional compiler and optimizer technologies support the specific demands of code for embedded systems?



Unfortunately, only little.

W/out Doubt

Traditional Optimizations

- are almost exclusively tuned towards performance optimization
- are not code-size sensitive and in general do not provide any control on their impact on the code size

This holds especially

...for code motion based optimizations.

In particular, this includes:

- Partial redundancy elimination
- Partial dead-code elimination (cf. Lecture Course 185.A05 Analysis and Verification)
- Partial redundant-assignment elimination (cf. Lecture Course 185.A05 Analysis and Verification)
- Strength reduction
- ...

Reminder using PRE as an Example

PRE can conceptually be considered a two-stage process:

1. Expression Hoisting
   ...hoisting computations to "earlier" safe computation points
2. Totally Redundant Expression Elimination
   ...eliminating computations, which become totally redundant by expression hoisting

Reminder using *LCM* as an Example

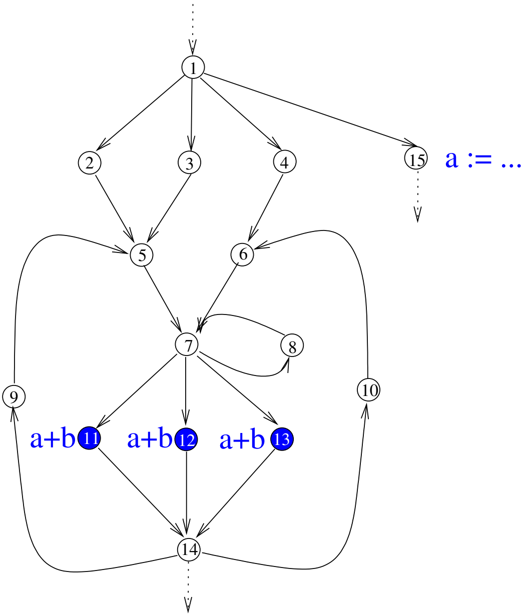
LCM can conceptually be considered the result of a two-stage process:

1. Hoisting Expressions
   ...to their "earliest" safe computation points
2. Sinking Expressions
   ...to their "latest" safe and still computationally optimal computation points

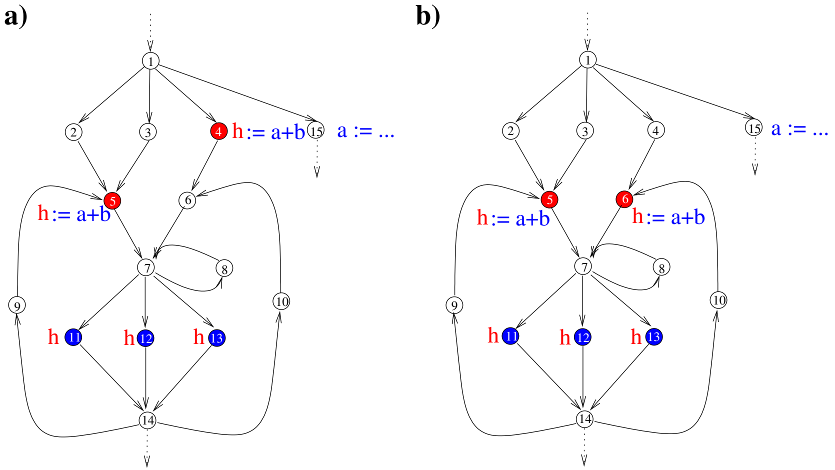
Towards Code-size Sensitive PRE

- Background: Classical PRE
  - ⇝ Busy Code Motion (BCM) / Lazy Code Motion (LCM) (Knoop, Rüthing, Steffen, PLDI'92)
    - Distinguished w/ the ACM SIGPLAN Most Influential PLDI Paper Award 2002 (for 1992)
    - Selected for the "20 Years of the ACM SIGPLAN PLDI: A Selection" (60 articles out of about 600 articles)
- Code-size Sensitive PRE
  - ⇝ Sparse Code Motion (SpCM) (Knoop, Rüthing, Steffen, POPL'00)
    - ...modular extension of BCM/LCM
      - ⇝ Modelling and solving the problem: ...based on graph-theoretic means
      - ⇝ Main Results: ...Correctness, Optimality

The Running Example (1)

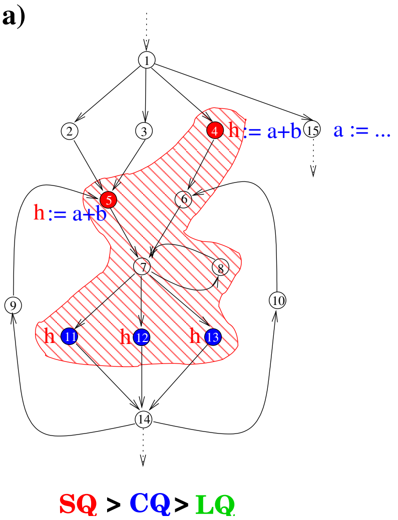


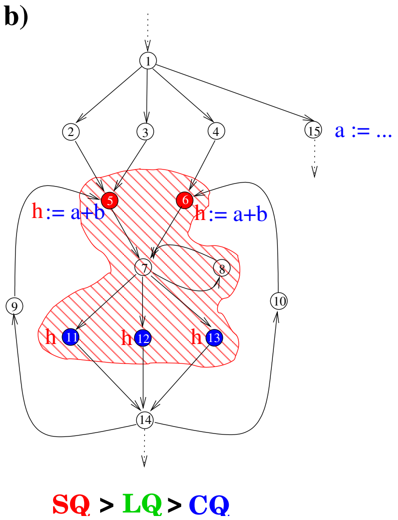
The Running Example (2)



Two Code–size Optimal Programs

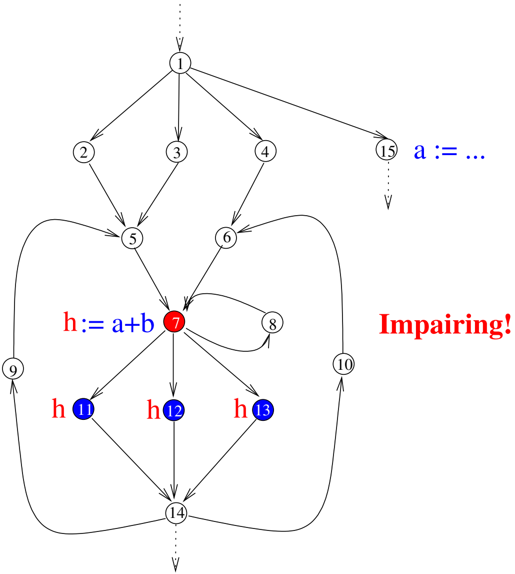
The Running Example (3)





The Running Example (4)

Note: The below transformation is not desired!



Code-size Sensitive PRE

⇝ The Problem

...how do we get code-size minimal placement of the computations, i.e., a placement that is

- admissible (semantics & performance preserving)
- code-size minimal?

⇝ The Solution: A new View to PRE

...consider PRE as a trade-off problem: Exchange original computations for newly inserted ones!

⇝ The Clou: Use Graph Theory!

...reduce the trade-off problem to the computation of tight sets in bipartite graphs based on maximum matchings!

We postpone but keep in mind that

...we have to answer:

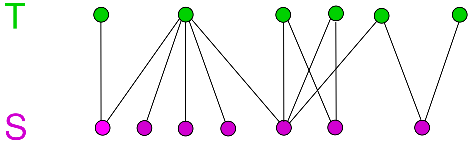
- Where are computations to be inserted and where original computations to be replaced?

...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!

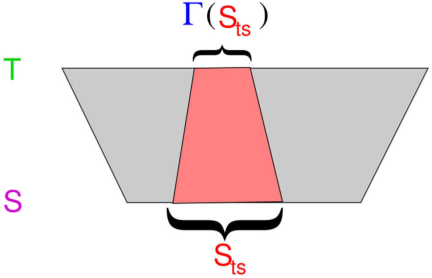
Bipartite Graphs



Tight Set

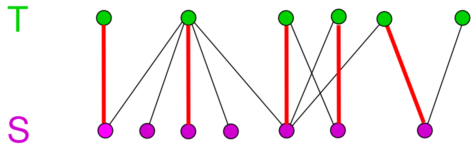
...of a bipartite graph $(S \cup T, E)$: Subset $S_{ts} \subseteq S$ w/

$$\forall\, S' \subseteq S.\ |S_{ts}| - |\Gamma(S_{ts})| \ \geq\ |S'| - |\Gamma(S')|$$



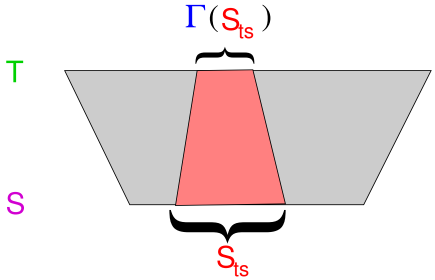
Two Variants: (1) Largest Tight Sets (2) Smallest Tight Sets

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 Sets

Obviously

...we can make use of off-the-shelve algorithms from graph theory in order to compute

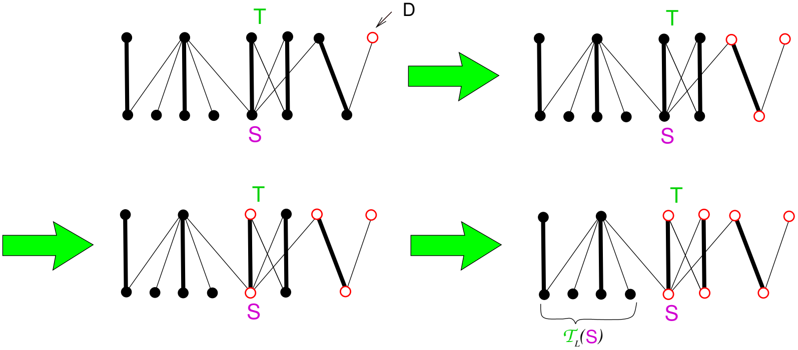
- Maximum matchings and
- Tight sets

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:



Algorithm LTS (Largest Tight Sets)

Input: A bipartite graph $(S \dot{\cup} T, E)$, a maximum matching M.

Output: The largest tight set $\mathcal{T}_{LaTS}(S) \subseteq S$.

$S_M := S$; $D := \{t \in T \mid t \text{ is unmatched}\}$;
`WHILE` $D \neq \emptyset$ `DO`
  choose some $x \in D$; $D := D \setminus \{x\}$;
  `IF` $x \in S$
    `THEN` $S_M := S_M \setminus \{x\}$;
      $D := D \cup \{y \mid \{x, y\} \in M\}$
    `ELSE` $D := D \cup (\Gamma(x) \cap S_M)$
  `FI`
`OD`;
$\mathcal{T}_{LaTS}(S) := S_M$

Algorithmus STS (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;
$\mathcal{T}_{SmTS}(S) := S_M$

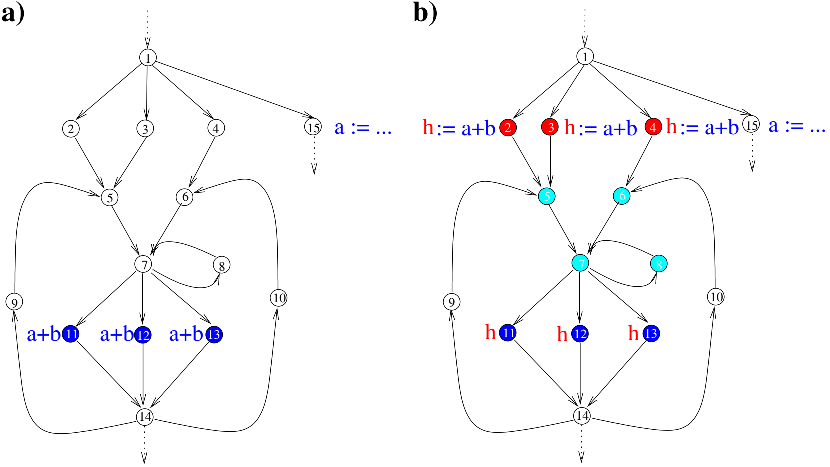
Modelling the Trade-off Problem

The Set of Nodes

$$T_{DS} \cup S_{DS} \quad \underbrace{2 \;\; 3 \;\; 4}_{Insert_{BCM}} \;\; \underbrace{5 \;\; 6 \;\; 7 \;\; 8}_{DownSafe/(Comp \cup UpSafe)} \;\; \underbrace{11 \;\; 12 \;\; 13}_{Comp/UpSafe}$$

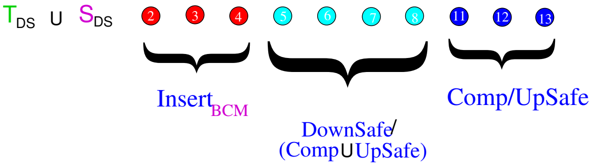
The Set of Edges...

The Set of Nodes

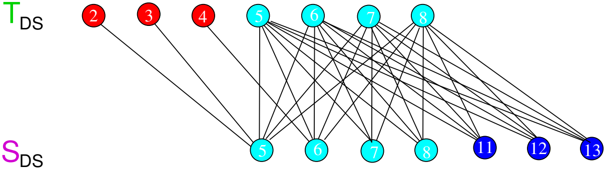


Modelling the Trade-off Problem

The Set of Nodes



The Bipartite Graph



The Set of Edges

...$\forall\, n \in S_{DS}\ \forall\, m \in T_{DS}$.

$$\{n, m\} \in E_{DS} \iff_{df} m \in \textbf{Closure}(pred(n))$$

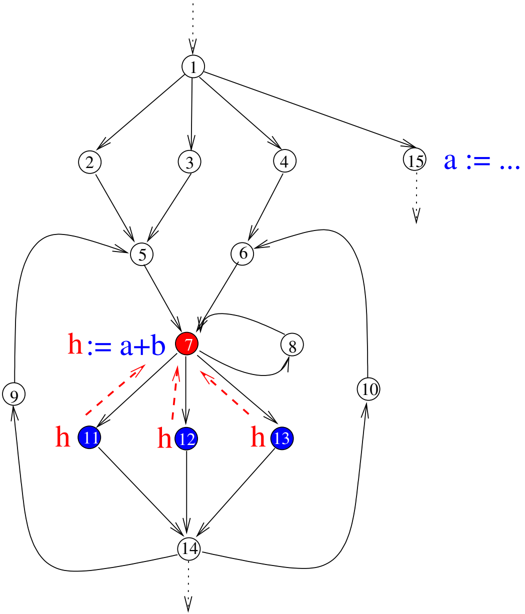
Down-Safety Closures

Definition (6.8.1, Down-Safety Closure)

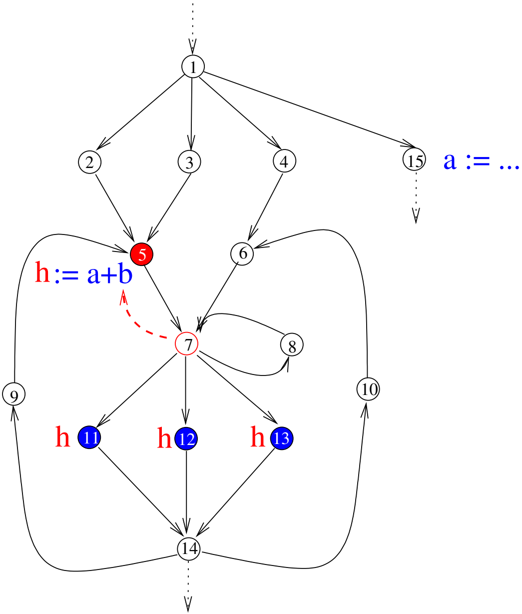
Let $n \in DownSafe/Upsafe$. Then the Down-Safety Closure $Closure(n)$ is the smallest set of nodes such that

1. $n \in Closure(n)$
2. $\forall\, m \in Closure(n) \setminus Comp.\ succ(m) \subseteq Closure(n)$
3. $\forall\, m \in Closure(n).\ pred(m) \cap Closure(n) \neq \emptyset \;\Rightarrow$ $$pred(m) \setminus UpSafe \subseteq Closure(n)$$

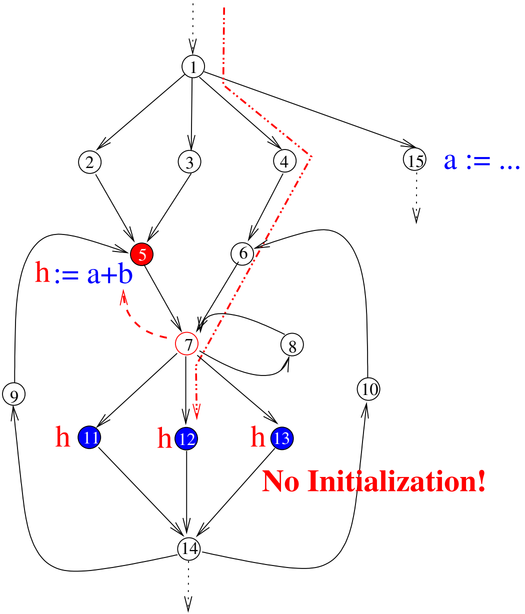
DownSafety Closures – The Central Idea (1)



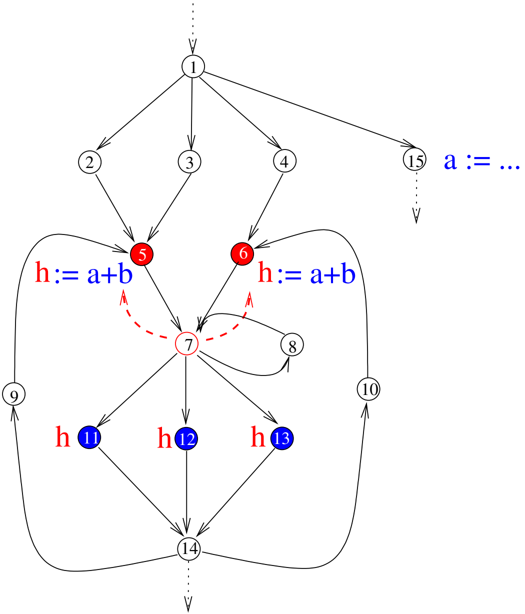
DownSafety Closures – The Central Idea (2)



DownSafety Closures – The Central Idea (3)



DownSafety Closures – The Central Idea (4)



Reminder: Down-Safety Closures

Definition (6.8.1, Down-Safety Closure)

Let $n \in \mathit{DownSafe}/\mathit{Upsafe}$. Then the Down-Safety Closure $\mathit{Closure}(n)$ is the smallest set of nodes such that

1. $n \in \mathit{Closure}(n)$
2. $\forall\, m \in \mathit{Closure}(n) \setminus \mathit{Comp}.\ \mathit{succ}(m) \subseteq \mathit{Closure}(n)$
3. $\forall\, m \in \mathit{Closure}(n).\ \mathit{pred}(m) \cap \mathit{Closure}(n) \neq \emptyset \;\Rightarrow$
$$\mathit{pred}(m) \setminus \mathit{UpSafe} \subseteq \mathit{Closure}(n)$$

Down-Safety Regions

Some subsets of nodes are distinguished. We call these subsets Down-Safety Regions.

Definition (6.8.2, Down-Safety Region)

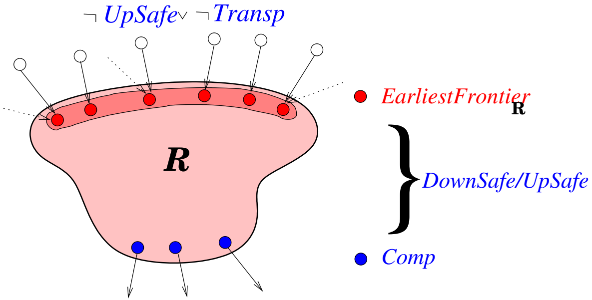
A set $\mathcal{R} \subseteq N$ of nodes is a down-safety region iff

1. $\mathit{Comp} \backslash \mathit{UpSafe} \subseteq \mathcal{R} \subseteq \mathit{DownSafe} \backslash \mathit{UpSafe}$
2. $\mathit{Closure}(\mathcal{R}) = \mathcal{R}$

Fundamental

Theorem (6.8.3, 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.

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 / Question 1

1. Where to insert computations, why is it correct?

Intuitively: At the earliestness frontier of the DS-region induced by the tight set...

Theorem (6.8.4, Tight Sets: Insertion Points)

Let $TS \subseteq S_{DS}$ be a tight set.
Then $\mathcal{R}_{TS} =_{df} \Gamma(TS) \cup (Comp \backslash UpSafe)$
is a down-safety region w/ $Body_{\mathcal{R}_{TS}} = TS$

Correctness

- An immediate corollary of Theorem 6.8.4 and the Initialization Theorem 6.8.3

Main Results / Question 2

2. What is the impact on the code size?

Intuitively: The difference between the number of inserted and replaced computations...

Theorem (6.8.5, Down-Safety Regions: Space Gain)

Let $\mathcal{R}$ be a down-safety region w/
$Body_{\mathcal{R}} =_{df} \mathcal{R} \setminus EarliestFrontier_{\mathcal{R}}$

Then

- *Space Gain by Inserting at $EarliestFrontier_{\mathcal{R}}$:*
  $|Comp \setminus UpSafe| - |EarliestFrontier_{\mathcal{R}}| =$
  $|Body_{\mathcal{R}}| - |\Gamma(Body_{\mathcal{R}})| \quad {}_{df}= defic(Body_{\mathcal{R}})$

Main Results / Question 3

3. Why is the result optimal, i.e., code-size minimal?

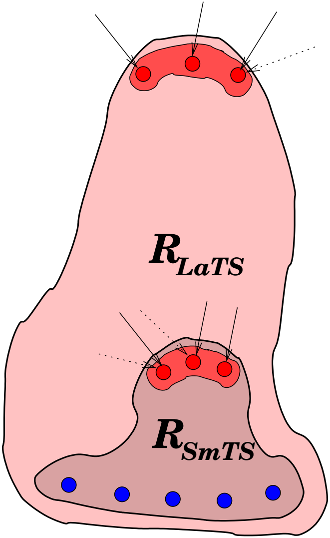
Intuitively: Due to a property inherent to tight sets (non-negative deficiency!)...

Theorem (Optimality Theorem / 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*$_{\mathbf{R}_{SmTS}}$

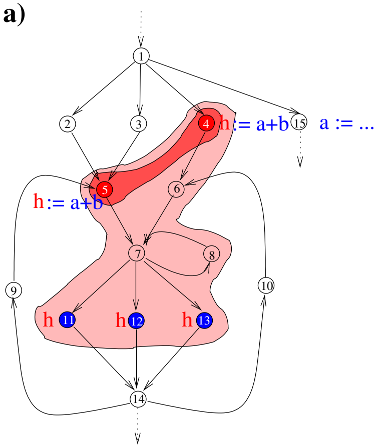
Smallest tight sets favor

Lifetime Quality

⇒ **Latestness Principle**

● *Comp*

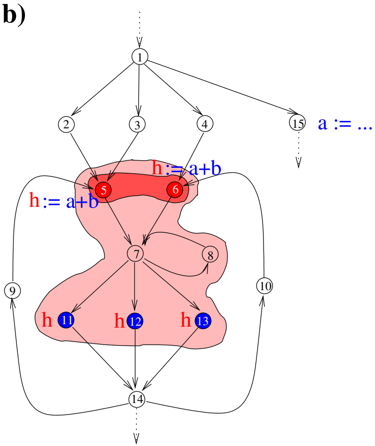
The Impact illustrated on the Running Exam.



Largest Tight Set

(SQ > CQ)

Earliestness Principle



Smallest Tight Set

(SQ > LQ)

Latestness Principle

Code-size Sensitive PRE at a Glance (1)

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

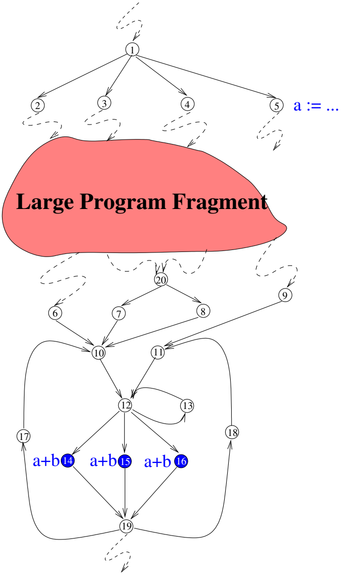
- Compute Largest/Smallest Tight Set
- Determine Insertion Points

Code-size Sensitive PRE at a Glance (2)

Choice of Priority	Apply	To	Using	Yields	Auxiliary Information Required
$\mathcal{LQ}$	Not meaningful: The identity, i.e., G itself is optimal !				
$\mathcal{SQ}$	Subsumed by $\mathcal{SQ} > \mathcal{CQ}$ and $\mathcal{SQ} > \mathcal{LQ}$!				
$\mathcal{CQ}$	**BCM**	G			UpSafe(G), DownSafe(G)
$\mathcal{CQ} > \mathcal{LQ}$	**LCM**	G		**LCM**(G)	UpSafe(G), DownSafe(G), Delay(G)
$\mathcal{SQ} > \mathcal{CQ}$	**SpCM**	G	**Largest tight set**	**SpCM**$_{LTS}$(G)	UpSafe(G), DownSafe(G)
$\mathcal{SQ} > \mathcal{LQ}$	**SpCM**	G	**Smallest tight set**		UpSafe(G), DownSafe(G)
$\mathcal{CQ} > \mathcal{SQ}$	**SpCM**	**LCM**(G)	**Largest tight set**		UpSafe(G), DownSafe(G), Delay(G) UpSafe(**LCM**(G)), DownSafe(**LCM**(G))
$\mathcal{CQ} > \mathcal{SQ} > \mathcal{LQ}$	**SpCM**	**LCM**(G)	**Smallest tight set**		UpSafe(G), DownSafe(G), Delay(G) UpSafe(**LCM**(G)), DownSafe(**LCM**(G))
$\mathcal{SQ} > \mathcal{CQ} > \mathcal{LQ}$	**SpCM**	**DL**(**SpCM**$_{LTS}$(G))	**Smallest tight set**		UpSafe(G), DownSafe(G), Delay(**SpCM**$_{LTS}$(G)), UpSafe(**DL**(**SpCM**$_{LTS}$(G))), DownSafe(**DL**(**SpCM**$_{LTS}$(G)))

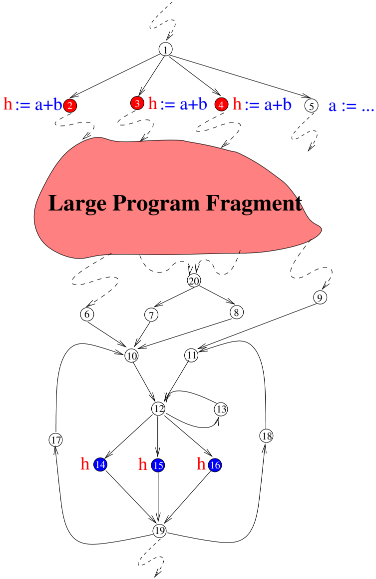
Flexibility (1)

The original program:



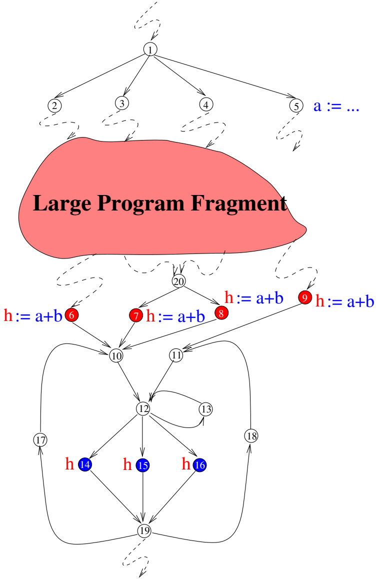
Flexibility (2)

BCM: A computationally optimal program ($\mathcal{CQ}$)



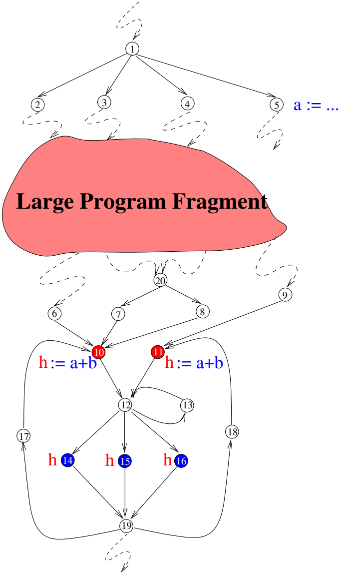
Flexibility (3)

LCM: A computationally & lifetime opt. program ($\mathcal{CQ} > \mathcal{LQ}$)



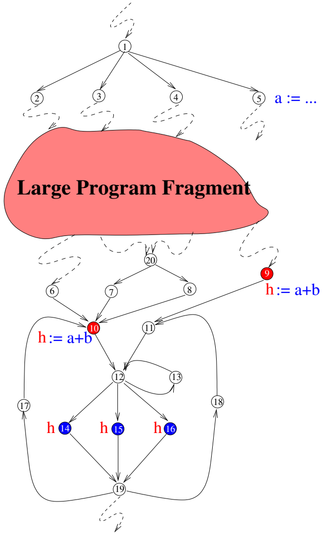
Flexibility (4)

SpCM: A code-size & lifetime opt. program ($\mathcal{SQ} > \mathcal{LQ}$)



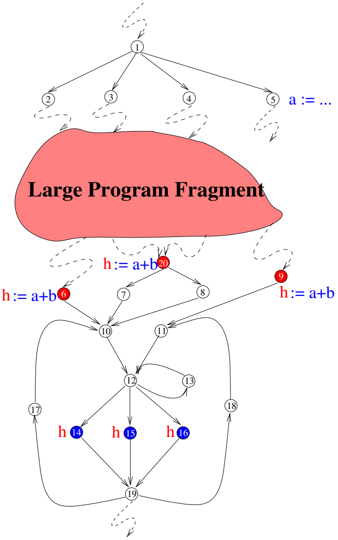
Flexibility (5)

SpCM: A computationally & lifetime best code-size optimal program ($\mathcal{SQ} > \mathcal{CQ} > \mathcal{LQ}$)



Flexibility (6)

SpCM: A code-size and lifetime best computationally optimal program ($\mathcal{CQ} > \mathcal{SQ} > \mathcal{LQ}$)



The History and Progress of PRE (1)

- 1958: A first glimpse of PRE
  - $\leadsto$ Ershov's work on "On Programming of Arithmetic Operations."
- $<$ 1979: Special techniques
  - $\leadsto$ Total redundancy elimination, loop invariant code motion
- 1979: The origin of modern PRE
  - $\leadsto$ Morel/Renvoise's seminal 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],...

The History and Progress of PRE (2)

- 1992: *BCM* and *LCM* [Knoop Rüthing, Steffen (PLDI'92)]
  - ⇝ *BCM* first to achieve computational optimality based on the earliestness principle
  - ⇝ *LCM* first to achieve computational optimality with minimum register pressure based on the latestness principle
  - ⇝ first to rigorously be proven correct and optimal
- 2000: *SpCM*: The origin of code-size sensitive PRE [Knoop, Rüthing, Steffen (POPL 2000)]
  - ⇝ first to allow prioritization of goals
  - ⇝ rigorously be proven correct and optimal
  - ⇝ first to bridge the gap between traditional compilation and compilation for embedded systems

The History and Progress of PRE (3)

- Since ca. 1997: A new strand of research on PRE
  - ⇝ Speculative PRE: Gupta, Horspool, Soffa, Xue, Scholz, Knoop,...
- 2005: Another fresh look at PRE (as maximum flow problem)
  - ⇝ Unifying PRE and Speculative PRE [Xue, Knoop (CC 2006)]

Why is it rewarding to consider PRE? (1)

It is...

- Relevant: Widely used in practice
- General: A family of optimizations rather than a single optimization
- Well understood: Proven correct and optimal
- Challenging: Conceptually simple but exhibits a variety of thought provoking phenomenons

Why is it rewarding to consider PRE (2)

Last but not least, PRE is...

- Truly classical: Looks back to a long history beginning with
  - Etienne Morel, Claude Renvoise. Global Optimization by Suppression of Partial Redundancies. Communications of the ACM 22(2):96-103, 1979.
  - Andrei P. Ershov. On Programming of Arithmetic Operations. Communications of the ACM 1(8):3-6, 1958.

Further Reading for Chapter 6 (1)

- Andrei P. Ershov. *On Programming of Arithmetic Operations*. Communications of the ACM 1(8):3-6, 1958.
- 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 6 (2)

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Retrospective: Lazy Code Motion*. In "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection", ACM SIGPLAN Notices 39(4):460-461&462-472, 2004.
- Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.
- 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.

Chapter 7

More on Code Motion

Code Motion Reconsidered

Traditionally:

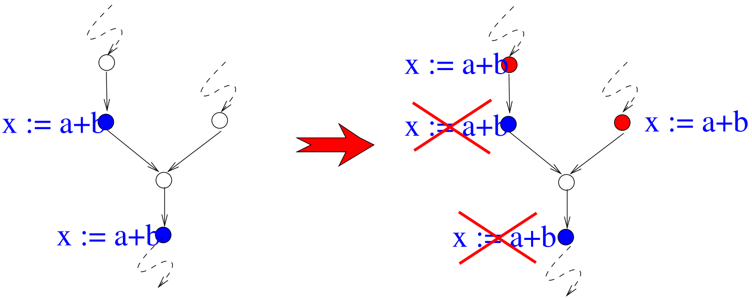
- Code (C) means expressions
- Motion (M) means hoisting

But:

- CM is more than hoisting of expressions and PR(E)E!

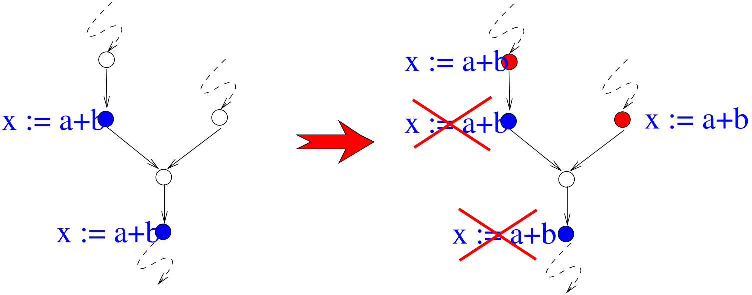
Obviously

...assignments are code, too.



Obviously

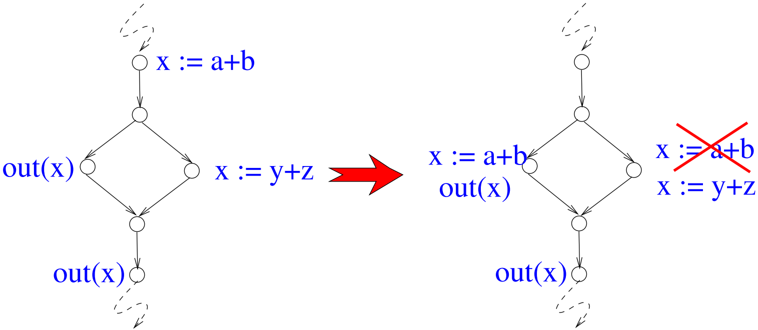
...assignments are code, too.



- Here, CM means eliminating partially redundant assignments (PRAE)

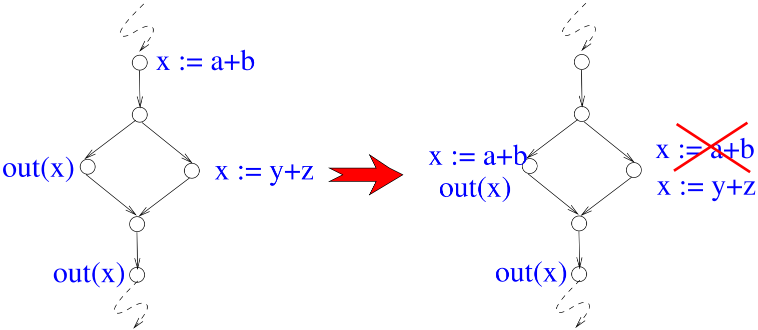
Differently from expressions...

...assignments can also be sunk.



Differently from expressions...

...assignments can also be sunk.



- Here, CM means eliminating partially dead code (PDCE)

Design Space of CM-Algorithms (1)

This results in the following design space of CM-algorithms:

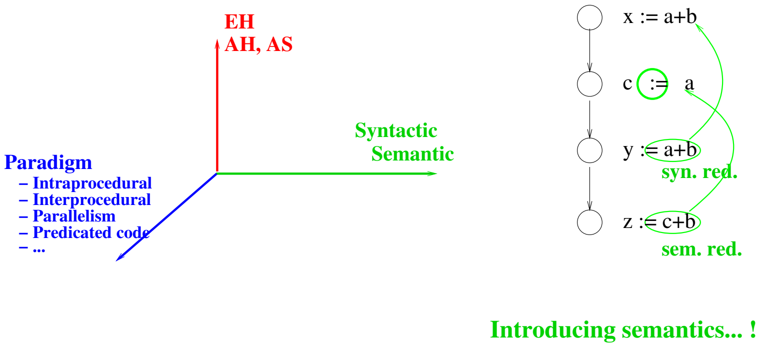
Generally:

- Code means expressions/assignments
- Motion means hoisting/sinking

Code / Motion	Hoisting	Sinking
Expressions	EH	·/·
Assignments	AH	AS

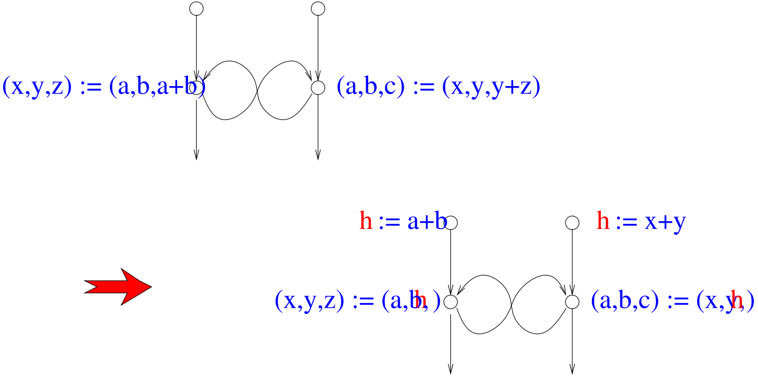
Design Space of CM-Algorithms (2)

Adding further dimensions to the design space of CM-algorithms:



Semantic Code Motion

...enables more powerful optimizations!



(Example from B. Steffen, TAPSOFT'87)

What is the Impact on Optimality?

Optimality statements are quite sensitive towards setting changes!

Three examples shall provide evidence for this:

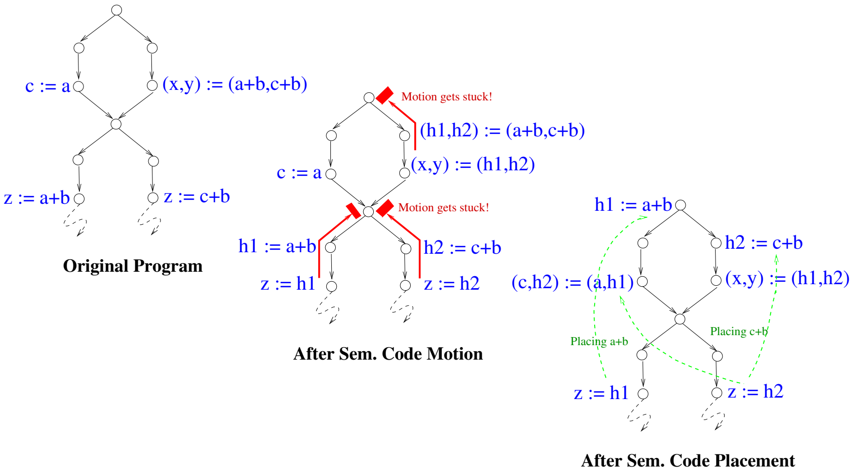
- Code motion vs. code placement
- Interdependencies of elementary transformations
- Paradigm dependencies

Chapter 7.1

Code Motion vs. Code Placement

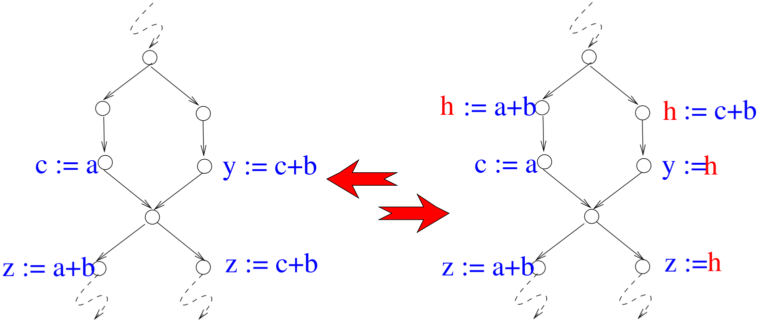
Code Motion (CM) vs. Code Placement (CP)

CM and CP are no synonyms!



Even worse

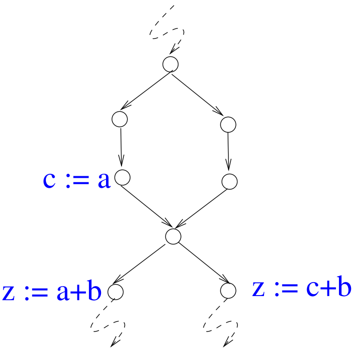
Optimality is lost!



Incomparable!

Even more worse

The performance can be impaired, when applied naively!

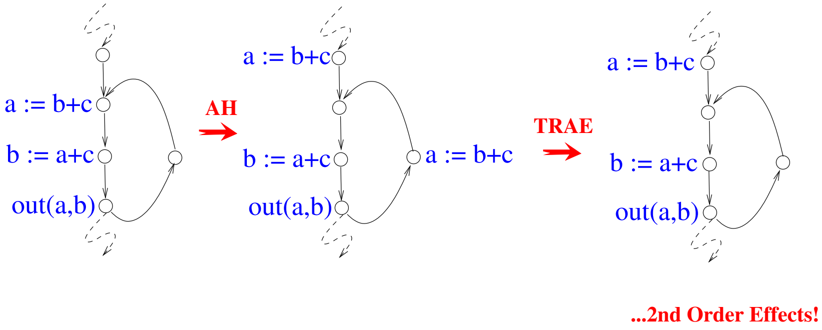


Chapter 7.2

Interactions of Elementary Transformations

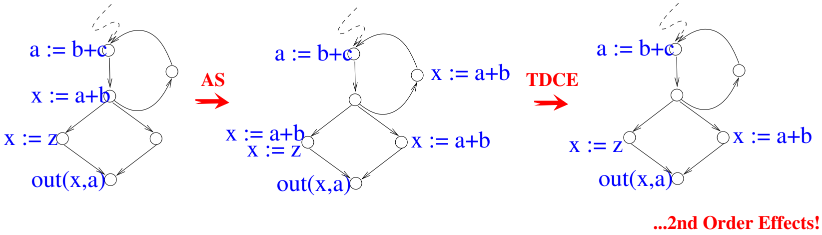
Assignment Hoisting (AH) plus Totally Redundant Assignment Elimination (TRAE)

...leads to Partially Redundant Assignment Elimination (PRAE):



Assignment Sinking (AS) plus Total Dead-Code Elimination (TDCE)

...leads to Partial Dead-Code Elimination (PDCE):



Conceptually

...we can understand PREE, PRAE, and PDCE as follows:

- PREE $=$ AH ; TREE
- PRAE $=$ (AH $+$ TRAE)*
- PDCE $=$ (AS $+$ TDCE)*

Optimality Results for PREE

Theorem (7.2.1, 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

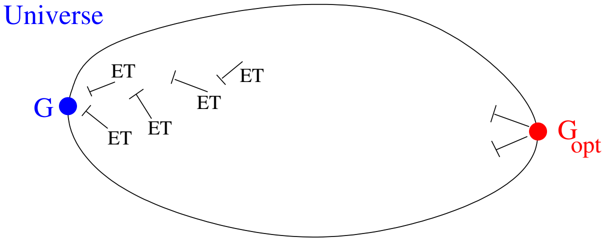
Deriving relation $\vdash$...

- PRAE... $G \vdash_{AH,TRAE} G'$ (ET={AH,TRAE})
- PDCE... $G \vdash_{AS,TDCE} G'$ (ET={AS,TDCE})

We can prove:

Theorem (7.2.2, 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):

- $\mathrm{AP} = (\mathrm{AH} + \mathrm{TRAE} + \mathrm{AS} + \mathrm{TDCE})^*$

...AP should be more powerful than PRAE and PDCE alone!

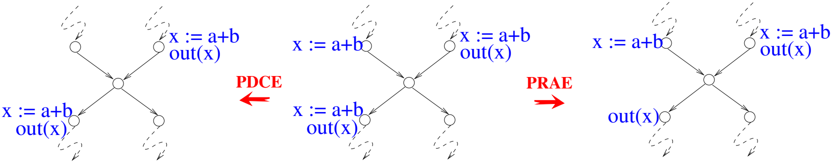
Now

...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!

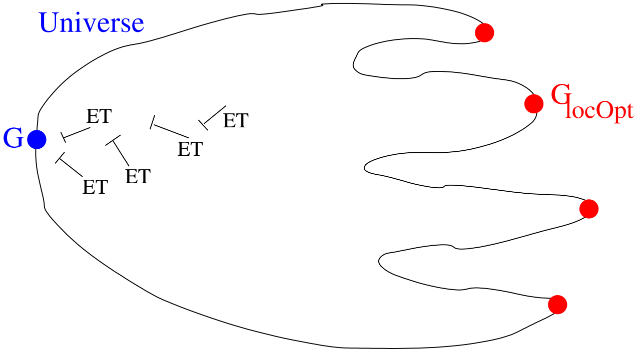
Indeed, it is but:



The resulting two programs are incomparable.

Confluence

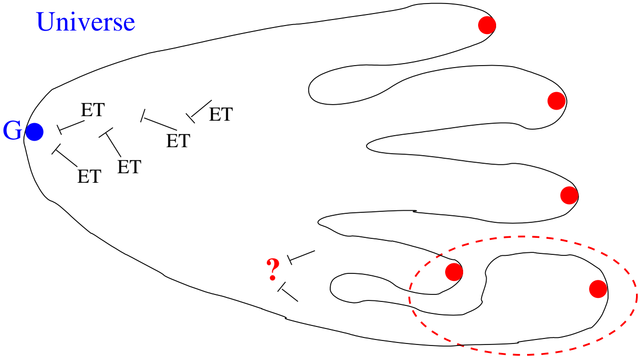
...and hence (global) optimality is lost!



Fortunately, we retain local optimality!

However

...there are settings, where we end up w/ universes like the following:

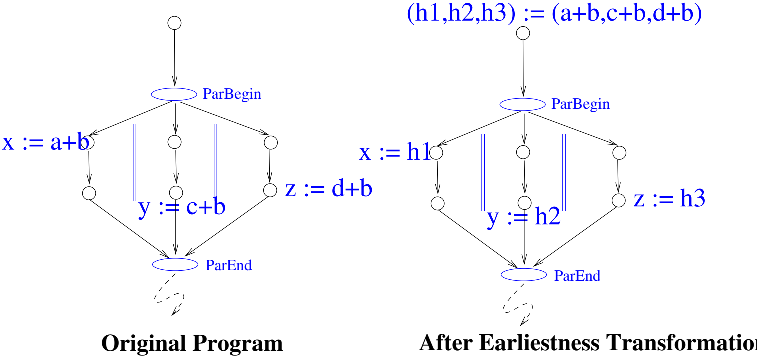


Here, even local optimality is lost!

Chapter 7.3

Paradigm Impacts

Adding Parallelism



...a naive transfer of the "place computations as early as possible" transformation strategy leads here to an essentially sequential program!

Adding Procedures

Similar phenomena are encountered when naively applying successful transformation strategies from the intraprocedural to the interprocedural setting.

Chapter 7.4

Extending Code Motion to Strength Reduction

Objective

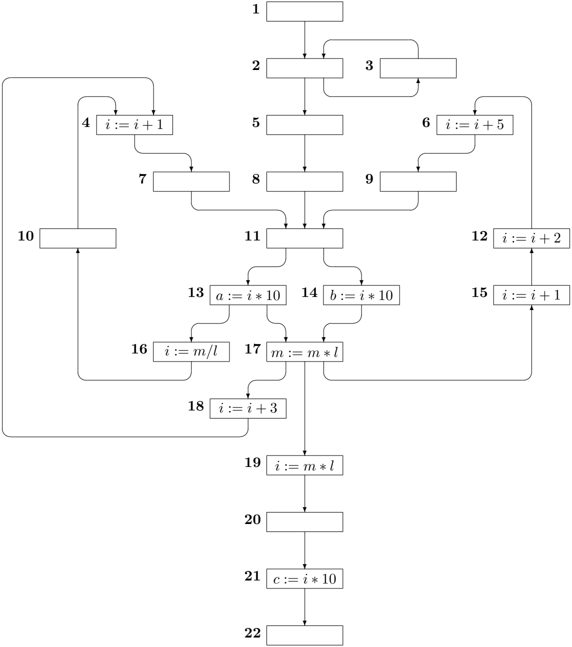
Developing a program optimization that

- uniformly covers
  - Partial Redundancy Elimination (PRE) and
  - Strength Reduction (SR)
- avoids superfluous register pressure due to unnecessary code motion
- requires only uni-directional data flow analyses

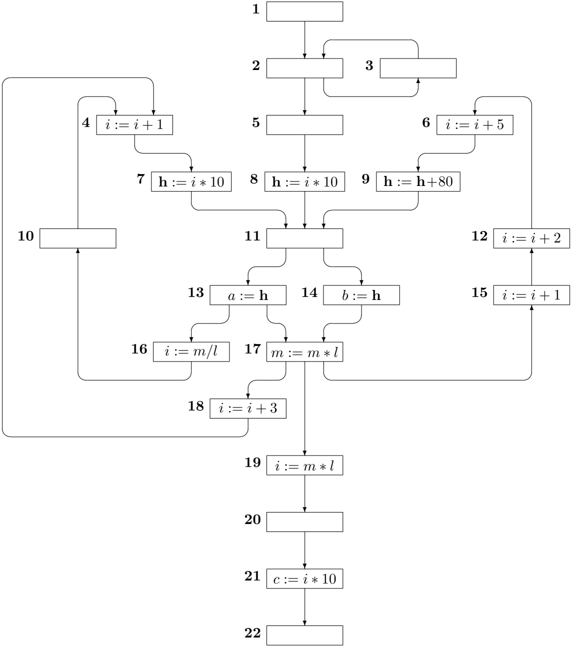
The Approach:

- Stepwise extending the *BCM* and the *LCM* to arrive at the
  - Busy (*BSR*) and Lazy Strength Reduction (*LSR*)

Illustration: The Original Program



The Result of Lazy Strength Reduction



From PRE towards LSR (1)

First, the notion of a candidate expression has to be adapted:

Candidate expressions for

- PRE: Each term t
- SR: Terms of the form $v * c$, where
  - v is a variable
  - c is a source code constant

From PRE towards LSR (2)

Second, the set of local predicates has to be extended:

- $Used(n) =_{df} v * c \in SubTerms(t)$
- $Transp(n) =_{df} x \not\equiv v$
- $SR\text{-}Transp(n) =_{df} Transp(n) \vee t \equiv v + d$ with $d \in \mathbf{C}$

Intuitively

The value of a candidate expression is

- killed at a node n, if $\neg(Transp(n) \vee SR\text{-}Transp(n))$
- injured at a node n, if $\neg Transp(n) \wedge SR\text{-}Transp(n)$

Important: Injured but not killed values can be

- cured by inserting an update assignment of the form $\mathbf{h} := \mathbf{h} + Eff(n)$ where $Eff(n) =_{df} c * d$.

  Note that $Eff(n)$ can be computed at compile time since both c and d are source code constants.

Extending BCM straigthforward to SR

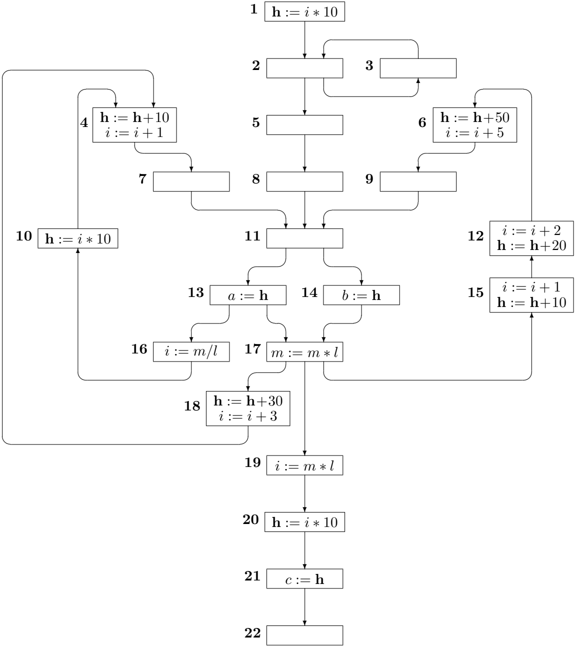
...leads to Simple Strength Reduction (*SSR*).

The *SSR*-Transformation:

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   2.1 $\texttt{Ins}_{\texttt{SSR}}$ the assignment $\mathbf{h} := v * c$
   2.2 $\texttt{InsUpd}_{\texttt{SSR}}$ the assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

Note: If both $\texttt{Ins}_{\texttt{SSR}}$ and $\texttt{InsUpd}_{\texttt{SSR}}$ hold, the initialization statement $\mathbf{h} := v * c$ must precede the update assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$.

The Result of *SSR*



Discussing the Effect of *SSR*

Shortcoming

- The multiplication-addition-deficiency

Remedy:

- Moving critical insertion points in the direction of the control flow to "earliest" non-critical ones.

Intuitively:

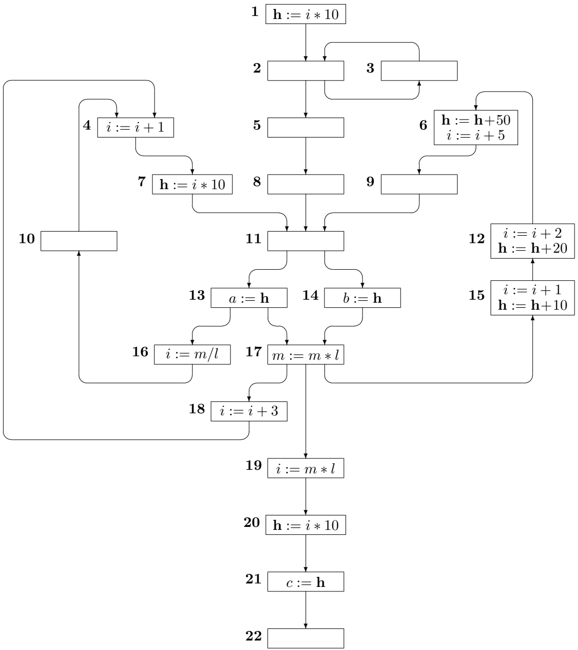
- A program point is critical if there is a $v * c$-free program path from this point to a modification of v

The 1st Refinement of *SSR*

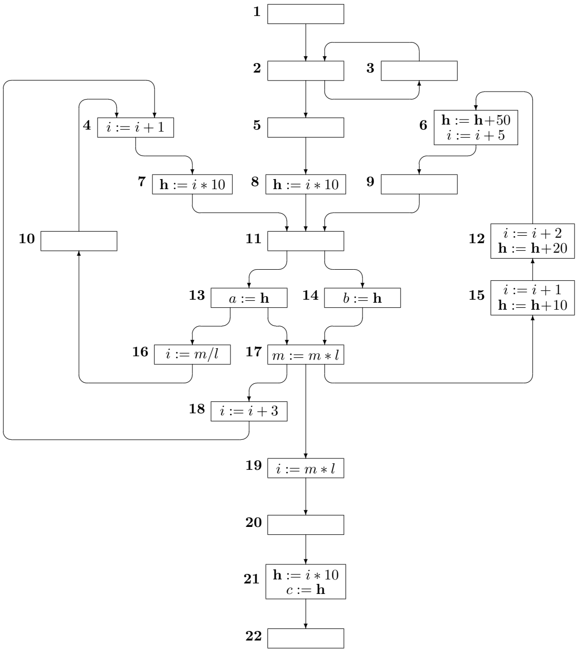
The SSR_{FstRef}-Transformation:

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   2.1 $\texttt{Ins}_{FstRef}$ the assignment $\mathbf{h} := v * c$
   2.2 $\texttt{InsUpd}_{FstRef}$ the assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

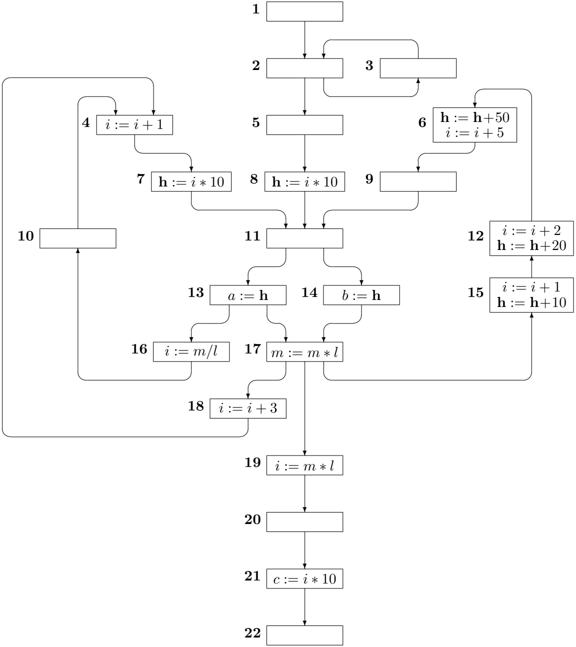
The Result of SSR_{FstRef}



Adding Laziness

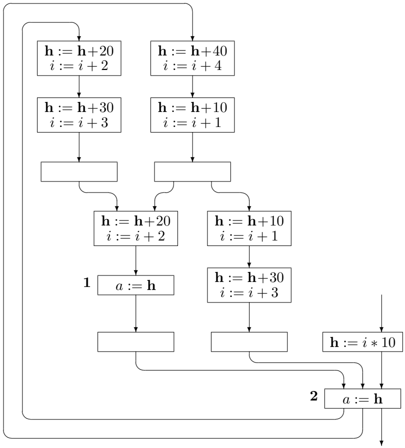


The Result of SSR_{SndRef}



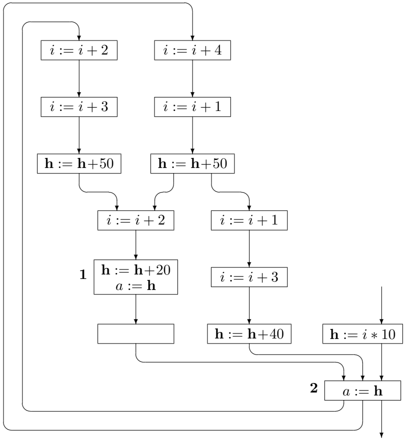
The Multiple-Addition Deficiency

Illustration:

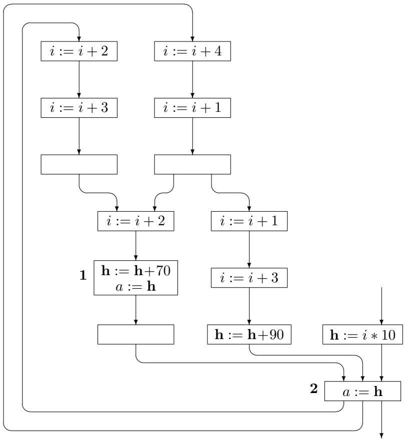


Overcoming the Multiple-Addition Deficiency

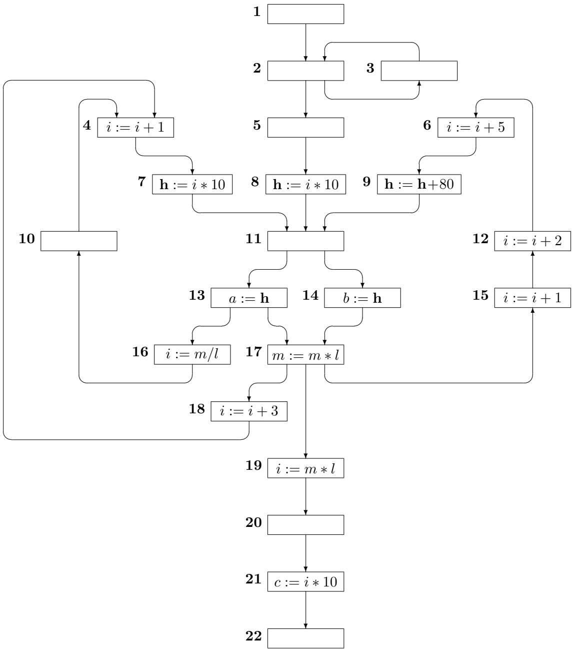
Accumulating the effect of *cure* assignments:



Refined Accumulation of Cure Assignments



The 3rd Refinement of *SSR*: *LSR*



Homework

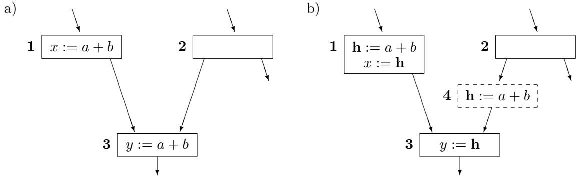
Assignment 4:

1. Specify the data flow analyses and transformations for
   - SSR
   - SSR_{FstRef} (overcoming the multiplication-addition deficiency)
   - SSR_{SndRef} (overcoming the register-pressure deficiency)
   - $SSR_{ThdRef} = LSR$ (overcoming the multiple-addition deficiency)
2. implement them in PAG, and
3. validate them on the running example of this chapter (or an example coming close to it).

Critical Edges

Like for *BCM* and *LCM* critical edges need to be split in order to get the full power of

- Lazy Strength Reduction (LSR)

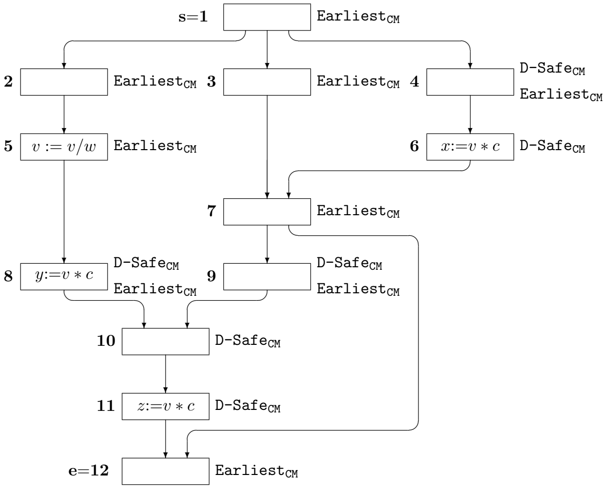


Summary of Predicate Values

...of the analyses of the LSR transformation:

	Node Number																					
Predicate	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
$\mathit{Safe}_{\mathtt{CM}}$	1	1	1	0	1	0	1	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0
$\mathtt{Earliest}_{\mathtt{CM}}$	1	0	0	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0
$\mathtt{Insert}_{\mathtt{CM}}$	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
$\mathit{Safe}_{\mathtt{SR}}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	0
$\mathtt{Earliest}_{\mathtt{SR}}$	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
$\mathtt{Insert}_{\mathtt{SSR}}$	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
$\mathtt{Insert}_{\mathtt{FstRef}}$	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
$\mathtt{Update}_{\mathtt{SndRef}}$	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	1	0
$\mathtt{Insert}_{\mathtt{SndRef}}$	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
$\mathtt{Insert}_{\mathtt{LSR}}$	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\mathtt{InsUpd}_{\mathtt{LSR}}$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$\mathtt{Delete}_{\mathtt{LSR}}$	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



Further Reading for Chapter 7 (1)

- 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. *Code Motion and Code Placement: Just Synonyms?* In Proceedings of the 7th European Symposium on Programming (ESOP'98), Springer-Verlag, LNCS 1381, 154-169, 1998.

Further Reading for Chapter 7 (2)

- Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-Verlag, LNCS 1428, 1998. (Chapter xxx)

- 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-Verlag, LNCS 1575, 91-106, 1999.

- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Strength Reduction*. Journal of Programming Languages 1(1):71-91, 1993.

Further Reading for Chapter 7 (3)

- Jens Knoop, Bernhard Steffen. *Code Motion for Explicitly Parallel Programs*. In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), ACM SIGPLAN Notices 34(8):13-24, 1999.
- Bernhard Steffen. *Optimal Run Time Optimization – Proved by a New Look at Abstract Interpretation*. In Proceedings of the 2nd Joint Conference on Theory and Practice of Software Development (TAPSOFT'87), Springer-Verlag, LNCS 249, 52-68, 1987.
- Bernhard Steffen. *Property-Oriented Expansion*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-Verlag, LNCS 1145, 22-41, 1996.

Further Reading for Chapter 7 (4)

- 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-Verlag, LNCS 432, 389-405, 1990.
- Bernhard Steffen, Jens Knoop, Oliver Rüthing. *Efficient Code Motion and an Adaption to Strength Reduction*. In Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), Springer-Verlag, LNCS 494, 394-415, 1991.

Part III

Interprocedural Data Flow Analysis

Outline

We consider:

- The Functional Approach (cf. Chapter 8)
  - The Base Setting
    Adding Procedures but no parameters and local variables
  - The General Setting
    Adding value parameters and local variables
  - Extensions
    Adding reference parameters and procedural parameters
- The Call String Approach (cf. Chapter 9)

Chapter 8

IDFA – The Functional Approach

Chapter 8.1

The Base Setting

Representing Programs

Programs w/ procedures will be represented in terms of

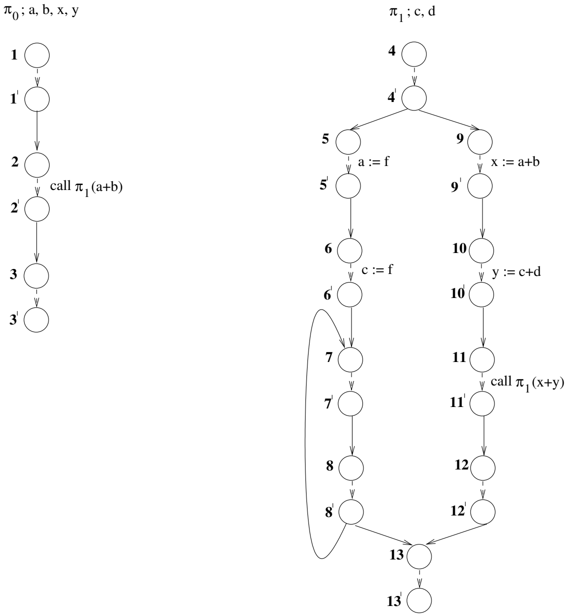
- Flow graph systems
- Interprocedural flow graphs

Flow Graph Systems

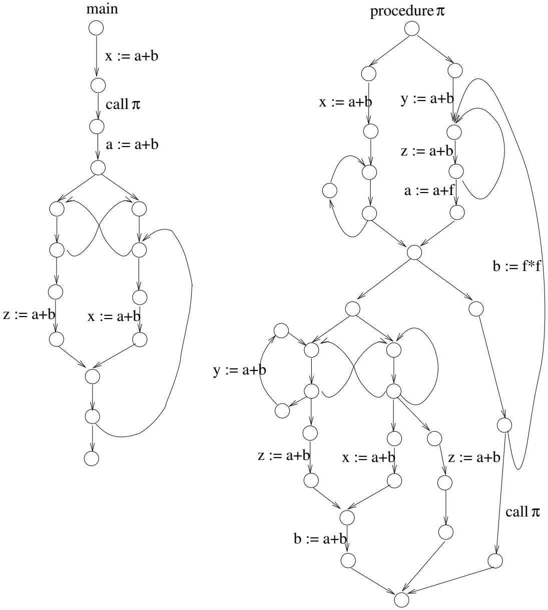
Definition (8.1.1, Flow Graph System)

A flow graph system $S =_{df} \langle G_0, \ldots, G_k \rangle$ is a system of (intraprocedural) flow graphs in the sense of Chapter 4, where each flow graph G_i represents a procedure of the underlying program Π; in particular, G_0 represents the main procedure or the main program of Π.

Illustration: Flow Graph System



Unnecessary nodes and edges may be removed



Notations and Conventions

- G_0 represents the main procedure.
- The start node $\mathbf{s}_0$ of G_0 is often abbreviated by $\mathbf{s}$.
- The sets of nodes and edges N_i und E_i, $0 \leq i \leq k$, are assumed to be pairwise disjoint.
- $N =_{df} \bigcup \{ N_i \mid i \in \{0, \ldots, k\}\}$ and $E =_{df} \bigcup \{ E_i \mid i \in \{0, \ldots, k\}\}$ denote the set of all nodes and edges of a flow graph system.
- $E_{call} \subseteq E$ denotes the set of edges, which represent a procedure call, for short, the set of call edges.

Interprocedural Flow Graphs

Definition (8.1.2, Interprocedural Flow Graph)

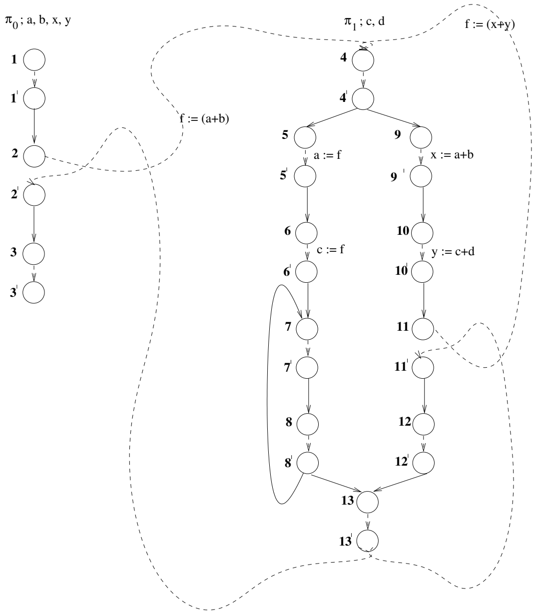
A flow graph system S induces an interprocedural flow graph, where the flow graphs of S are melted to a single flow graph $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$.

G^* evolves from S by replacing each call edge e of a procedure π by two new edges e_c and e_r.

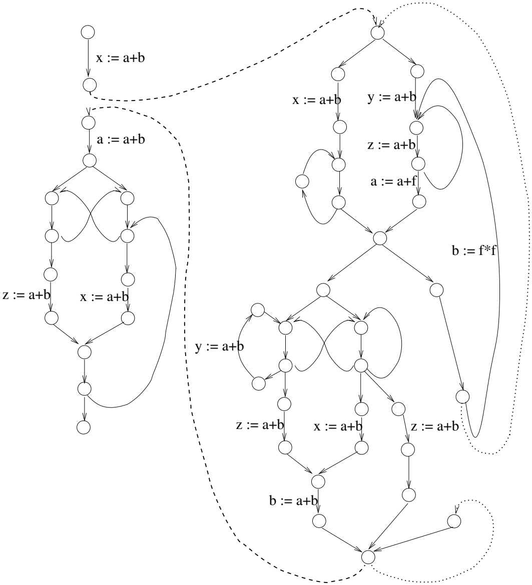
The edge e_c connects the source node of e w/ the start node of the called procedure.

The edge e_r connects the end node of the called procedure w/ the final node of e.

Illustration: Interprocedural Flow Graph



Unnecessary nodes and edges may be removed



Notations and Conventions (Cont'd)

- The set of new edges in an interprocedural flow graph are called the call edges and return edges of G^*, and are denoted by E_c^* und E_r^*.
- $E_{call}^* =_{df} E_c^* \cup E_r^*$ denotes the set of call and return edges of G^*.

Towards Interprocedural DFA (1)

Lifting the analysis level from elements to functions leads to the "functional" *MaxFP* approach. It relies on:

The "functional" *MaxFP* equation system:

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

By

$$[\![\;\;]\!]^{*} : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

we denote the greatest solution of the above equation system.

Towards Interprocedural DFA (2)

Intuitively

The functional *MaxFP* approach lifts the (basic) *MaxFP* approach to the level of functions, i.e.

- The *MaxFP* solution is not computed for a single lattice element as start information but simultaneously for all.

(Basic) *MaxFP* approach vs. functional *MaxFP* approach:
The Equivalence Theorem 8.1.3 characterizes the relationship of the basic *MaxFP* approach and the functional *MaxFP* approach:

Theorem (8.1.3, Equivalence)

$$\forall\, n \in N\ \forall\, c_{\mathsf{s}} \in \mathcal{C}.\ \mathit{MaxFP}_{(G,[\![\ ]\!])}(n)(c_s) = [\![\ n\ ]\!]^*(c_{\mathsf{s}})$$

In the following we will overload the symbol $[\![\ ]\!]$ and use it to also denote the greatest fixed point $[\![\ n\ ]\!]^*$ of the functional *MaxFP* equation system.

Outlook

The functional perspective of the *MaxFP* approach is the key to

- interprocedural (i.e., of programs w/ procedures)
- parallel (i.e., of programs w/ parallelism)

data flow analysis.

Chapter 8.1.1

Local Abstract Semantics

(Local) Abstract Semantics

Two components:

- Data flow analysis lattice $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top)$
- Data flow analysis functional $[\![\ ]\!]' : E^* \to (\mathcal{C} \to \mathcal{C})$

Note: In the parameterless base setting call edges and return edges of E^* are given the identity function on $\mathcal{C}$ as their semantics.

Chapter 8.1.2

The *IMOP* Approach

Interprocedurally Valid Paths (1)

Observations:

- The notion of finite paths for intraprocedural flow graphs extends naturally to interprocedural flow graphs.
- Unlike, however, as in intraprocedural flow graphs, where each path connecting two nodes represents (up to non-determinism) a possible execution of the program, this does not hold for interprocedural flow graphs.
- In interprocedural DFA this is taken care of by focusing on interprocedurally valid paths.

Interprocedurally Valid Paths (2)

Intuitively:

Interprocedurally valid paths respect the call/return behaviour of procedures.

Definition (8.1.2.1, Interprocedurally Valid Path)

Identifying call and return edges of G^* with opening and closing brackets "(" and ")", the set of interprocedurally valid paths is given by the set of prefix-closed expressions of the language of balanced bracket expressions.

Notation:

In the following we denote the set of interprocedurally valid paths (for short: interprocedural paths) from a node m to a node n by $\mathbf{IP}[m, n]$.

Interprocedurally Valid Paths (3)

Observation:

If we consider the sequences of edge labelings (we suppose that each edge is uniquely labeled by some mark) of a path as word of a formal language, then the set of intraprocedurally valid paths is given by a regular language, the one of interprocedurally valid paths by a context-free language.

Note:

- Sharir and Pnueli gave an algorithmic definition of interprocedurally valid paths in 1981.
- An immediate definition of interprocedurally valid paths in terms of a context-free language is possible, too.
- The definition of interprocedurally valid paths as in Definition 8.1.2.1 is due to Reps, Horwitz, and Sagiv, POPL'95.

The *IMOP* Approach

The *IMOP* Solution:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ \mathit{IMOP}_{c_{\mathbf{s}}}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!]'(c_{\mathbf{s}}) \mid p \in \mathbf{IP}[\mathbf{s}, n] \,\}$$

where $\mathrm{IP}[\mathbf{s}, n]$ denotes the set of interprocedurally valid paths from **s** to n.

Chapter 8.1.3

The *IMaxFP* Approach

The *IMaxFP* Approach (1)

The *IMaxFP* approach is a two-stage approach.

Stage 1: Preprocess – The Semantics of Procedures

The 2nd Order *IMaxFP* Equation System 8.1.3.1

$[\![\, n \,]\!] =$

$$\begin{cases} Id_{\mathcal{C}} & \text{if } n \in \{\, \mathbf{s}_0, \ldots, \mathbf{s}_k \,\} \\ \bigsqcap\{[\![\, (m,n) \,]\!] \circ [\![\, m \,]\!] \mid m \in pred_{flowGraph(n)}(n)\} & \text{otherwise} \end{cases}$$

and

$$[\![\, e \,]\!] = \begin{cases} [\![\, e \,]\!]' & \text{if } e \in E \backslash E_{call} \\ [\![\, end(caller(e)) \,]\!] & \text{otherwise} \end{cases}$$

The *IMaxFP* Approach (2)

Stage 2: Main Process – The "Actual" Interprocedural DFA

The 1st Order *IMaxFP* Equation System 8.1.3.2

$$inf\ (n) = \begin{cases} c_{\mathbf{s}} \quad \text{if } n = \mathbf{s}_0 \\ \bigsqcap \{\, inf\ (src(e)) \mid e \in caller(flowGraph(n)) \,\} \quad \text{if } n \in \{\mathbf{s}_1, \ldots, \mathbf{s}_k\} \\ \bigsqcap \{\, [\![(m,n)]\!](inf\ (m)) \mid m \in pred_{flowGraph(n)}(n) \,\} \quad \text{otherwise} \end{cases}$$

The *IMaxFP* Approach (3)

The *IMaxFP* Solution:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{c_{\mathbf{s}}}(n) =_{df} inf\ {}^{*}_{c_{\mathbf{s}}}(n)$$

Notations

We introduce the following mappings on a flow graph system S:

- *flowGraph* : $N \cup E \rightarrow S$ maps the nodes and edges of S to the flow graph containing them.
- *callee* : $E_{call} \rightarrow S$ maps every call edge to the flow graph of the called procedure.
- *caller* : $S \rightarrow \mathcal{P}(E_{call})$ maps every flow graph to the set of call edges calling it.
- *start* : $S \rightarrow \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ and *end* : $S \rightarrow \{\mathbf{e}_0, \ldots, \mathbf{e}_k\}$ map every flow graph of S to its start node and stop node.

Chapter 8.1.4

Main Results

Main Results – 1st Stage

Safety&coincidence results of the 2nd-order 1st-stage analysis:

Theorem (8.1.4.1, 2nd Order)

For all $e \in E_{call}$ hold:

1. $[\![\, e \,]\!] \sqsubseteq \bigsqcap \{ [\![\, p \,]\!]' \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, *if the data flow analysis functional* $[\![\ ]\!]'$ *is monotonic.*
2. $[\![\, e \,]\!] = \bigsqcap \{ [\![\, p \,]\!]' \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, *if the data flow analysis functional* $[\![\ ]\!]'$ *is distributive.*

where the mappings *src* and *dst* yield the start and final node of an edge.

Complete Interprocedural Paths

Definition (8.1.4.2, 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, the completeness requirement states that the occurrences of $\mathbf{s}_i$ and n belong to the same incarnation of the procedure.
- We have that the subpaths of a complete interprocedural path that belong to a procedure call, are either disjoint or properly nested.

Main Results – 2nd Stage

Safety&coincidence results of the 1st-order 2nd-stage analysis:

Theorem (8.1.4.3, Interprocedural Safety)

The IMaxFP solution is a safe, i.e., a lower approximation of the IMOP solution, i.e.

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{c_{\mathbf{s}}}(n) \sqsubseteq IMOP_{c_{\mathbf{s}}}(n)$$

if the data flow analysis functional $[\![\ ]\!]'$ *is monotonic.*

Theorem (8.1.4.4, Interprocedural Coincidence)

The IMaxFP solution coincides with the IMOP solution, i.e.

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{c_{\mathbf{s}}}(n) = IMOP_{c_{\mathbf{s}}}(n)$$

if the data flow analysis functional $[\![\ ]\!]'$ *is distributive.*

Chapter 8.1.5

Algorithms

The 2nd Order *IMaxFP*-Alg. 8.1.5.1 – Preprocess

Input: (1) A flow-graph system S, and (2) an abstract semantics consisting of a data-flow lattice $\mathcal{C}$, and a data-flow functional $[\![\ ]\!]' : E^* \to (\mathcal{C} \to \mathcal{C})$.

Output: Under the assumption of termination (cf. Theorem 8.1.5.4), an annotation of S with functions $[\![\![n]\!]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *gtr* , which stands for *global transformation*), and $[\![e]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *ltr* , which stands for *local transformation*) representing the greatest solution of the 2nd order Equation System 8.1.3.1.

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.

The 2nd Order *IMaxFP*-Alg. 8.1.5.1 – Preprocess

(Prologue: Initializing the annotation arrays *gtr* and *ltr* and the variable *workset*)

FORALL $n \in N$ DO
    IF $n \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ THEN $gtr\,[n] := Id_{\mathcal{C}}$
        ELSE $gtr\,[n] := \top_{[\,\mathcal{C} \to \mathcal{C}\,]}$ FI OD;
FORALL $e \in E$ DO
    IF $e \in E_{call}$ THEN $ltr\,[e] := \top_{[\,\mathcal{C} \to \mathcal{C}\,]}$ ELSE $ltr\,[e] := [\![\, e \,]\!]'$ FI OD;
$workset := \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$;

The 2nd Order *IMaxFP*-Alg. 8.1.5.1 – 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

The 2nd Order *lMaxFP*-Alg. 8.1.5.1 – Preprocess

```
        ELSE ( i.e., m ∉ {e_1, ..., e_k} )
          FORALL n ∈ succ_flowGraph(m)(m) DO
            meet := ltr[(m, n)] ∘ gtr[m] ⊓ gtr[n];
            IF gtr[n] ⊐ meet
              THEN
                gtr[n] := meet;
                workset := workset ∪ {n}
            FI
          OD
      FI
    ESOOHC
OD.
```

The 1st Order *IMaxFP*-Alg. 8.1.5.2 – Main Process

Input: (1) A flow-graph system S, (2) an abstract semantics consisting of a data-flow lattice $\mathcal{C}$, and a data-flow functional $[\![\ ]\!]$ computed by Algorithm 8.1.5.1, and (3) a context information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 8.5.1.4), the *IMaxFP*-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is *distributive*: variable *inf* stores for every node the strongest component information valid there wrt the context information $c_{\mathbf{s}}$.
(2) $[\![\ ]\!]$ is *monotonic*: variable *inf* stores for every node a valid component information wrt the context information $c_{\mathbf{s}}$, i.e., a lower bound of the strongest component information valid there.

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S. The temporary *meet* stores the result of the most recent meet operation.

The 1st Order *IMaxFP*-Alg. 8.1.5.2 – Main Process

(Prologue: Initialization of the annotation array *inf* and the variable *workset*)
FORALL $n \in N \backslash \{\mathbf{s}_0\}$ DO $inf[n] := \top$ OD;
$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;
$workset := \{ \mathbf{s}_0 \}$;

The 1st Order *IMaxFP*-Alg. 8.1.5.2 – 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
            $meet := [\![(m, n)]\!](inf[m]) \sqcap inf[n]$;
            IF $inf[n] \sqsupset meet$
                THEN
                    $inf[n] := meet$;
                    $workset := workset \cup \{n\}$
            FI;

The 1st Order *IMaxFP*-Alg. 8.1.5.2 – Main Process

```
            IF (m, n) ∈ E_call
               THEN
                  meet := inf[m] ⊓ inf[start(callee((m, n)))];
                  IF inf[start(callee((m, n)))] ⊐ meet
                     THEN
                        inf[start(callee((m, n)))] := meet;
                        workset := workset ∪ { start(callee((m, n))) }
                  FI
            FI
         OD
      ESOOHC
OD.
```

A 1st Variant of the *IMaxFP*-Algorithm

- Algorithm 8.1.5.3 uses the semantics functions computed by Algorithm 8.1.5.1 more efficiently.
- Algorithm 8.1.5.1 and 8.1.5.3 constitute a pair of algorithms computing the *IMaxFP* solution, too.
- Replacing Algorithm 8.5.1.2 by Algorithm 8.1.5.3 has no impact on Algorithm 8.1.5.1.
- Unlike Algorithm 8.1.5.2, Algorithm 8.1.5.3 does not iterate over all nodes but only over procedure start nodes. After stabilization of the solution for the start nodes, a single run over all other nodes in the epilogue suffices to compute the *IMaxFP* solution at every node.

The 1st Order *IMaxFP*-Alg. 8.1.5.3 – The "Functional" Main Process

Input: (1) A flow-graph system S, (2) an abstract semantics consisting of a data-flow lattice $\mathcal{C}$, and the data-flow functionals $[\![\ ]\!] =_{df} gtr$ and $[\![\ ]\!] =_{df} ltr$ with respect to $\mathcal{C}$ (computed by Algorithm 8.1.5.1), and (4) a context information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 8.1.5.4), the *IMaxFP*-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is *distributive*: variable *inf* stores for every node the strongest component information valid there wrt the context information $c_{\mathbf{s}}$.
(2) $[\![\ ]\!]$ is *monotonic*: variable *inf* stores for every node a valid component information wrt the context information $c_{\mathbf{s}}$, i.e., a lower bound of the strongest component information valid there.

Remark: The variable *workset* controls the iterative process, and the temporary *meet* stores the most recent approximation.

The 1st Order *IMaxFP*-Alg. 8.1.5.3 – The "Functional" Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)

FORALL $\mathbf{s} \in \{\mathbf{s}_i \,|\, i \in \{1, \ldots, k\}\}$ DO $inf[\mathbf{s}] := \top$ OD;

$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;

$workset := \{\mathbf{s}_i \,|\, i \in \{1, 2, \ldots, k\}\}$;

The 1st Order *IMaxFP*-Alg. 8.1.5.3 – The "Functional" Main Process

(Main process: Iterative fixed point computation)

WHILE $workset \neq \emptyset$ DO
    CHOOSE $\mathbf{s} \in workset$;
        $workset := workset \backslash \{\mathbf{s}\}$;
        $meet := inf[\mathbf{s}] \sqcap \bigsqcap \{ [\![\, src(e) \,]\!](inf[start(flowGraph(e))]) \mid e \in caller(flowGraph(\mathbf{s})) \}$;
        IF $inf[\mathbf{s}] \sqsupset meet$
            THEN
                $inf[\mathbf{s}] := meet$;
                $workset := workset \cup \{ start(callee(e)) \mid e \in E_{call}.\ flowGraph(e) = flowGraph(\mathbf{s}) \}$
        FI
    ESOOHC
OD;

The 1st Order *IMaxFP*-Alg. 8.1.5.3 – The "Functional" Main Process

(Epilogue)
FORALL $n \in N \backslash \{\mathbf{s}_i \mid i \in \{0, \ldots, k\}\}$ DO
$\quad inf[n] := [\![\, n \,]\!](inf[start(flowGraph(n))])$ OD.

Termination

Theorem (8.1.5.4, Termination)

The sequential composition of Algorithm 8.1.5.1 and Algorithm 8.1.5.2 resp. Algorithm 8.1.5.3 terminates with the IMaxFP solution, if the data flow analysis functional $[\![\ ]\!]'$ is monotonic and the function lattice $[\mathcal{C} \to \mathcal{C}]$ satisfies the descending chain condition.

Note: The descending chain condition on $[\mathcal{C} \to \mathcal{C}]$ implies the descending chain condition on $\mathcal{C}$.

A 2nd Variant of the *IMaxFP*-Algorithm (1)

Partial instead of total computation of the semantics of the procedures:

- Unlike to the previous two algorithm variants, the new variant allows an interleaving of preprocess and main process.
- The computation starts with the main process algorithm.
- If a procedure call is encountered during the iterative process, the preprocess algorithm is started for this procedure and the current data flow fact.
- After completion of the computation of the effect of the procedure for this data flow fact, the main process algorithm is continued with the computed result.

A 2nd Variant of the *IMaxFP*-Algorithm (2)

Note:

- The computation of the semantics of the procedures is performed demand-drivenly.
- The semantics of procedures are only computed as as far as necessary.
- Overall, this results in some efficiency gain in practice, which, however, is difficult to quantify.

Chapter 8.1.6

Applications

Applications

- For the parameterless base setting the specifications of intraprocedural DFA problems can be reused unmodified.
- In order to be effective, the descending chain condition must hold both for the data flow analysis lattice and its corresponding function lattice.
- This condition holds in particular for all bitvector problems (availability of expressions, lifeness of variables, reaching definitions, etc.) but not for simple constants. Therefore, weaker and simpler classes of constants are used interprocedurally, e.g., the set of linear constants.

Chapter 8.2

The General Setting

Outline

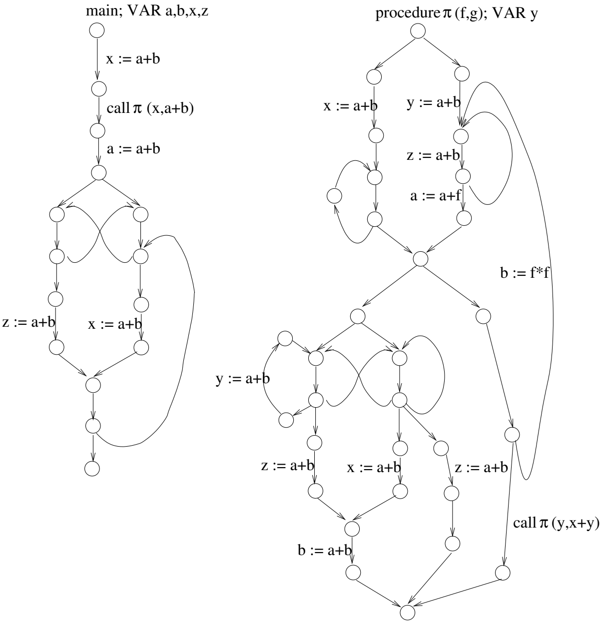
We extend our setting by adding

- Value parameters
- Local variables

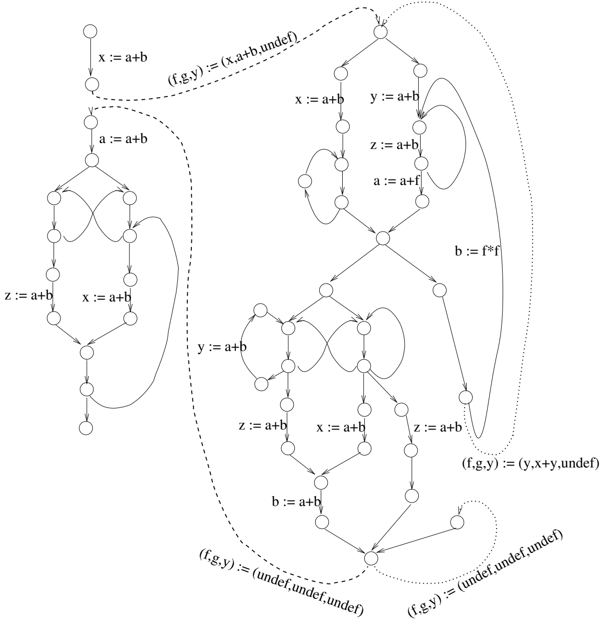
This requires to adjust our program representations towards

- Flow graph systems (FGS) w/ value parameters and local variables
- Interprocedural flow graphs (IFG) w/ value parameters and local variables

FGS w/ Value Parameters and Local Variables



IFG w/ Value Parameters and Local Variables



New Phenomena

...related to procedures, value parameters, and local variables.

Conceptually most important:

- Existence of an unlimited number of copies (incarnations) of local variables and value parameters at run-time due to recursive procedures.
- After termination of a recursive procedure call the local variables and value parameters of the proceding not yet finished procedure call become valid again.
- The run-time system handles this phenomena by means of of a run-time stack which stores the activation records of the various procedure incarnations.

For program analysis, we have to take these phenomena into account and to model them properly.

Data Flow Analysis Stacks

Intuitively:

- DFA stacks are a compile-time equivalent of run-time stacks.
- Entries in DFA stacks are data flow facts of an underlying DFA lattics $\mathcal{C}$.
- We denote the set of all non-empty DFA stacks by $STACK$.

Manipulating DFA Stacks

DFA stacks can be manipulated by:

1. newstack : $\mathcal{C} \rightarrow STACK$
   newstack(c) generates a new DFA stack with entry c.
2. push : $STACK \times \mathcal{C} \rightarrow STACK$
   push stores a new entry on top of a DFA stack.
3. pop : $STACK \rightarrow STACK$
   pop removes the top-most entry of a DFA stack.
4. top : $STACK \rightarrow \mathcal{C}$
   top yields the contents of the top-most entry of a DFA stack w/out modifying the stack.

Remarks (1)

- The usual stack function emptystack : $\rightarrow$ *STACK* is replaced by newstack. Empty DFA stacks are not considered since they do not occur in interprocedural DFA.
- push and pop allow to manipulate the top-most entries of a DFA stack. This is different to and less flexible as for run-time stacks but suffices for interprocedural DFA.
- In fact, DFA stacks are only conceptually relevant, i.e., for the specifying, i.e., for the specifying *IMOP* approach but not for the algorithmic *IMaxFP* approach.

Remarks (2)

- Like run-time stacks DFA stacks store that part of the history of a program path that is relevant after finishing a procedure call.
- DFA stack entries can be considered abstractions of the activation records of procedure calls.
- The top-most entry of a DFA stack represents always the currently valid activation record (therefore, DFA stacks are never empty).
- Other than the top-most DFA stack entries represent the activation records of already started but not yet finished procedure calls.

Chapter 8.2.1

Local Abstract Semantics

Basic Local Abstract Semantics

Basic Local Abstract Semantics on DFA Lattice

1. DFA lattics $\hat{\mathcal{C}} = (\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \perp, \top)$
2. DFA functional $[\![\ ]\!]' : E^* \to (\mathcal{C} \to \mathcal{C})$
3. Return functional $\mathcal{R} : E_{call} \to (\mathcal{C} \times \mathcal{C} \to \mathcal{C})$

Induced Local Abstract Semantics

Induced Local Abstract Semantics on DFA Stacks

- DFA functional $[\![\ ]\!]^* : E^* \to (STACK \to STACK)$ on DFA stacks induced by a basic local abstract semantics that is defined by

  $\forall\, e \in E^*\ \forall\, stk \in STACK.\ [\![\, e\,]\!]^*(stk) =_{df}$

$$\begin{cases} \text{push}(\,\text{pop}(stk),\ [\![\, e\,]\!]'(\text{top}(stk))\,) & \text{if } e \in E^* \backslash E^*_{call} \\ \text{push}(\,stk,\ [\![\, e\,]\!]'(\text{top}(stk))\,) & \text{if } e \in E^*_c \\ \text{push}(\,\text{pop}(\text{pop}(stk)),\ \mathcal{R}_e(\,\text{top}(\text{pop}(stk)),\ [\![\, e\,]\!]'(\text{top}(stk))\,)\,) & \\ & \text{if } e \in E^*_r \end{cases}$$

Notations related to DFA Stacks

- $STACK_{\geq i}$ ($STACK_{\leq i}$, etc.), $i \in \mathbb{N}$ denotes the set of all DFA Stacks w/ at last (at most, etc.) i entries (hence $STACK$ equals $STACK_{\geq 1}$.
- $STACK_i$, $i \in \mathbb{N}$, denotes the set of all DFA Stacks w/ exactly i entries.
- ϑ_{stk} denotes the number of entries of the DFA stack stk.
- stk_i, $1 \leq i \leq \vartheta_{stk}$, denotes the ith entry of the DFA stack stk.

Properties

Lemma (8.2.1.1)

Let $e \in E^*$ and $stk \in STK$. Then we have:

1. $[\![\, e \,]\!]^*(stk) \in \begin{cases} STK_{\vartheta_{stk}} & \text{if } e \in E^* \backslash E^*_{call} \\ STK_{\vartheta_{stk}+1} & \text{if } e \in E^*_c \\ STK_{\vartheta_{stk}-1} & \text{if } e \in E^*_r \wedge \vartheta_{stk} \geq 2 \end{cases}$
2. $pop([\![\, e \,]\!]^*(stk)) = pop(stk)$, *if* $e \in E^* \backslash E^*_{call}$
3. $pop([\![\, e \,]\!]^*(stk)) = stk$, *if* $e \in E^*_c$
4. $pop([\![\, e \,]\!]^*_R(stk)) = pop(pop(stk))$, *if* $e \in E^*_r \wedge \vartheta_{stk} \geq 2$

Structure of the Semantic Functions

All semantic functions occurring in interprocedural DFA are an element of the following subsets of the set of all functions $\mathcal{F} =_{df} [\, STACK \to STACK \,]$ on DFA stacks:

- $\mathcal{F}_{ord}$
- $\mathcal{F}_{psh}$
- $\mathcal{F}_{pop}$

These functions are characterized by:

$$\mathcal{F}_{ord} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK.\ \mathrm{pop}(f(stk)) = pop(stk) \,\}$$

$$\mathcal{F}_{psh} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK.\ \mathrm{pop}(f(stk)) = stk \,\}$$

$$\mathcal{F}_{pop} =_{df} \{\, f \in \mathcal{F} \mid \forall\, stk \in STACK_{\geq 2}.\ \mathrm{pop}(f(stk)) = \mathrm{pop}(\mathrm{pop}(stk)) \,\}$$

Properties

Lemma (8.2.1.2)

$\forall\, f_{pp} \in \mathcal{F}_{pop} \;\; \forall\, f_o, f'_o \in \mathcal{F}_{ord} \;\; \forall\, f_{ph} \in \mathcal{F}_{psh}.\;\; f_o \circ f'_o\,,\;\; f_{pp} \circ f_o \circ f_{ph} \in \mathcal{F}_{ord}$

Lemma (8.2.1.3)

1. $\forall\, e \in E^* \backslash E^*_{call}.\; [\![\, e \,]\!]^* \in \mathcal{F}_{ord}$
2. $\forall\, e \in E^*_c.\; [\![\, e \,]\!]^* \in \mathcal{F}_{psh}$
3. $\forall\, e \in E^*_r.\; [\![\, e \,]\!]^* \in \mathcal{F}_{pop}$

Significant Part of DFA Functions

Only the two top-most entries of DFA stacks are modified by DFA functions. This gives rise to the following definition:

Definition (8.2.1.4, Significant Part)

- $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh}$: Then $f_s : \mathcal{C} \to \mathcal{C}$ is defined by:
  $f_s(c) =_{df} \text{top}(f(\text{newstack}(c)))$
- $f \in \mathcal{F}_{pop}$: Then $f_s : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined by:
  $f_s(c_1, c_2) =_{df} \text{top}(f(\text{push}(\text{newstack}(c_1), c_2)))$ (Note that $\mathcal{C} \times \mathcal{C}$ is a lattice, if $\mathcal{C}$ is a lattice.)

We have:

Lemma (8.2.1.5)

1. $\forall\, e \in E^* \backslash E^*_r.\ [\![\, e \,]\!]^*_s = [\![\, e \,]\!]'$
2. $\forall\, e \in E^*_r\ \forall\, c_1, c_2 \in \mathcal{C} \times \mathcal{C}.\ [\![\, e \,]\!]^*_s = \mathcal{R}_e(c_1, [\![\, e \,]\!]'(c_2))$

Properties

Lemma (8.2.1.6)

1. $\forall\, e \in E^* \backslash E^*_{call}\ \forall\, stk \in STK.\ [\![\, e \,]\!]^*(stk) = stk'$ *with* $\vartheta_{stk'} = \vartheta_{stk}$ *and*

$$\forall\, 1 \leq i \leq \vartheta_{stk'}.\ stk_i' =_{df} \begin{cases} stk_i & \textit{if } i < \vartheta_{stk} \\ [\![\, e \,]\!]^*_s(stk_{\vartheta_{stk}}) & \textit{if } i = \vartheta_{stk} \end{cases}$$

2. $\forall\, e \in E^*_c\ \forall\, stk \in STK.\ [\![\, e \,]\!]^*(stk) = stk'$ *with* $\vartheta_{stk'} = \vartheta_{stk} + 1$ *and*

$$\forall\, 1 \leq i \leq \vartheta_{stk'}.\ stk_i' =_{df} \begin{cases} stk_i & \textit{if } i < \vartheta_{stk} + 1 \\ [\![\, e \,]\!]^*_s(stk_{\vartheta_{stk}}) & \textit{if } i = \vartheta_{stk} + 1 \end{cases}$$

3. $\forall\, e \in E^*_r\ \forall\, stk \in STK_{\geq 2}.\ [\![\, e \,]\!]^*(stk) = stk'$ *with* $\vartheta_{stk'} = \vartheta_{stk} - 1$ *and*

$$\forall\, 1 \leq i \leq \vartheta_{stk'}.\ stk_i' =_{df} \begin{cases} stk_i & \textit{if } i < \vartheta_{stk} - 1 \\ [\![\, e \,]\!]^*_s(stk_{\vartheta_{stk}-1}, stk_{\vartheta_{stk}}) & \textit{if } i = \vartheta_{stk} - 1 \end{cases}$$

S-Monotonicity, S-Distributivity

Definition (8.2.1.7, S-Monotonicity, S-Distributivity)

A DFA function $f \in \mathcal{F}_{ord} \cup \mathcal{F}_{psh} \cup \mathcal{F}_{pop}$ is

1. s-monotonic iff f_s is monotonic
2. s-distributive iff f_s is distributive

Properties

Lemma (8.2.1.8)

For all $e \in E^*$ *the function* $[\![\, e \,]\!]^*$ *is s-monotonic* (*s-distributive*), *if*

$$e \in E^* \backslash E_r^* : \quad [\![\, e \,]\!]' \;\; \textit{is monotonic (distributive)}$$

$$e \in E_r^* : \quad [\![\, e \,]\!]' \;\; \textit{and} \;\; \mathcal{R}_e \;\; \textit{are monotonic (distributive)}$$

Conventions

In the following medskip

- we consider s-monotonicity and s-distributivity as generalizations of the usual monotonicity and distributivity.

To this end, we

- identify lattice elements with their representation as a DFA stack with just a single entry.
- extend the meet and join operation $\sqcap$ and $\sqcup$ in the following fashion to (the top most entries of) DFA stacks:

$$\bigsqcap STK =_{df} \text{newstack}(\bigsqcap\{top(stk) \,|\, stk \in STK\})$$

$$\bigsqcup STK =_{df} \text{newstack}(\bigsqcup\{top(stk) \,|\, stk \in STK\})$$

Chapter 8.2.2

The $IMOP_{Stk}$ Approach

The IMOP_{Stk} Approach

The IMOP_{Stk} Solution:

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ \mathit{IMOP}_{Stk\,c_{\mathbf{s}}}(n) =_{df} \\ \bigsqcap \{\, [\![\, p \,]\!]^{*}(\mathrm{newstack}(c_{\mathbf{s}})) \mid p \in \mathbf{IP}[\mathbf{s}, n] \,\}$$

Chapter 8.2.3

The $IMaxFP_{Stk}$ Approach

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: $f \sqsubseteq f'$ gdw. $f \sqcap f' = f$.

The $IMaxFP_{Stk}$ Approach (1)

The effects of procedures (2nd Order):

The 2nd Order Equation System 8.2.3.1

$$[\![n]\!] = \begin{cases} Id_{STACK} & \text{if } n \in start(S) \\ \bigsqcap \{ [\![(m,n)]\!] \circ [\![m]\!] \mid m \in pred_{flowGraph(n)}(n) \} & \\ & \text{otherwise} \end{cases}$$

and

$$[\![e]\!] = \begin{cases} [\![e]\!]^* & \text{if } e \in E \backslash E_{call} \\ [\![e_r]\!]^* \circ [\![end(callee(e))]\!] \circ [\![e_c]\!]^* & \text{otherwise} \end{cases}$$

The $IMaxFP_{Stk}$ Approach (2)

The 1st Order $IMaxFP_{Stk}$ Equation System 8.2.3.2:

$$inf\ (n) = \begin{cases} \text{newstack}(c_{\mathbf{s}}) & \text{if } n = \mathbf{s}_0 \\ \bigsqcap \{ [\![\, e_c \,]\!]^*(inf\ (src(e))) \mid e \in caller(flowGraph(n)) \} & \\ & \text{falls } n \in start(S) \backslash \{\mathbf{s}_0\} \\ \bigsqcap \{ [\![\, (m,n) \,]\!](inf\ (m)) \mid m \in pred_{flowGraph(n)}(n) \} & \\ & \text{otherwise} \end{cases}$$

The $IMaxFP_{Stk}$ Solution:

$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{Stk\, c_{\mathbf{s}}}(n) =_{df} inf^{\,*}_{c_{\mathbf{s}}}(n)$

Chapter 8.2.4

Main Results

Main Results (1)

Important:

Lemma (8.2.4.1)

For all $n \in N$ we have that the semantic functions $[\![\ e\ ]\!]^*$, $e \in E^*$, are

1. *s-monotonic:* $[\![\ n\ ]\!] \sqsubseteq imop_n$
2. *s-distributive:* $[\![\ n\ ]\!] = imop_n$

where $imop_n : N \to (STACK \to STACK)$ denotes a functional that is defined by:

$$\forall\, n \in N.\ imop_n =_{df} \begin{cases} Id_{STACK} & \text{if } n \in start(S) \\ \bigsqcap\{[\![\ p\ ]\!]^* \mid p \in \mathbf{CIP}[start(flowGraph(n)), n]\} & \text{otherwise} \end{cases}$$

Main Results (2)

Theorem (8.2.4.2, 2nd-Order)

For all $e \in E_{call}$ we have:

1. $[\![\ e\ ]\!] \sqsubseteq \bigsqcap \{ [\![\ p\ ]\!]^* \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, *if* $[\![\ \ ]\!]^*$ *is s-monotonic.*
2. $[\![\ e\ ]\!] = \bigsqcap \{ [\![\ p\ ]\!]^* \mid p \in \mathbf{CIP}[src(e), dst(e)] \}$, *if* $[\![\ \ ]\!]^*$ *is s-distributive.*

Main Results (3)

Theorem (8.2.4.3, Interprocedural Safety)

The $IMaxFP_{Stk}$ solution is a lower (i.e., safe) approximation of the $IMOP_{Stk}$ solution, i.e.

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{Stk_{c_{\mathbf{s}}}}(n) \sqsubseteq IMOP_{Stk_{c_{\mathbf{s}}}}(n)$$

if $[\![\ ]\!]^*$ is s-monotonic.

Theorem (8.2.4.4, Interprocedural Coincidence)

The $IMaxFP_{Stk}$ solution coincides with the $IMOP_{Stk}$ solution, i.e.

$$\forall\, c_{\mathbf{s}} \in \mathcal{C}\ \forall\, n \in N.\ IMaxFP_{Stk_{c_{\mathbf{s}}}}(n) = IMOP_{Stk_{c_{\mathbf{s}}}}(n)$$

if $[\![\ ]\!]^*$ is s-distributive.

Chapter 8.2.5

Algorithms

Algorithms

- The algorithms of chapter xx can straightforwardly be extended to stack-based functions.
- This way we receive
  - The standard variant of pre- and post-process
  - The more efficient variant of pre- and functional main process
  - a demand-driven "by-need" variant
- In the following we present another stackless variant. The clou of this variant is that stacks have at most 2 entries during analysis time.

  Therefore, a single temporary storing the temporarily existing stack entry during procedure calls is sufficient for the implementation.

Stackless $IMaxFP_{Stk}$ Alg. 8.2.5.1 / Preprocess (2nd Order)

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 8.2.5.4), an annotation of S with functions $[\![\![\ n\ ]\!]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *gtr*, which stands for *global transformation*), and $[\![\ e\ ]\!] : \mathcal{C} \to \mathcal{C}$ (stored in *ltr*, which stands for *local transformation*) representing the greatest solution of Equation System 8.2.3.1.

Remark: The variable *workset* controls the iterative process. Its elements are nodes of the flow-graph system S. Note that due to the mutual interdependence of the definitions of $[\![\![\ ]\!]\!]$ and $[\![\ ]\!]$ the iterative approximation of $[\![\![\ ]\!]\!]$ is superposed by an interprocedural iteration step, which updates the current approximation of the effect $[\![\ ]\!]$ of call edges. The temporary *meet* stores the result of the most recent meet operation.

Stackless $IMaxFP_{Stk}$ Alg. 8.2.5.1 / Preprocess (2nd Order)

(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; $\langle\star\rangle$
$workset := \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$;

Stackless $IMaxFP_{Stk}$ Alg. 8.2.5.1 / Preprocess (2nd Order)

(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

Stackless IMaxFP_{Stk} Alg. 8.2.5.1 / Preprocess (2nd Order)

```
        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.
```

Stackless $IMaxFP_{Stk}$ Alg. 8.5.2.2 / Main Process (1st Order)

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 8.5.2.1, and (3) a context information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 8.5.2.4), the $IMaxFP_{StkLss}$-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is *distributive*: variable *inf* stores for every node the strongest component information valid there with respect to the context information $c_{\mathbf{s}}$.
(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.

Stackless $IMaxFP_{Stk}$-Alg. 8.2.5.2 / Main Process (1st Order)

(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;

Stackless IMaxFP_{Stk} Alg. 8.2.5.2 / Main Process (1st Order)

$$
\begin{array}{l}
\quad\quad \text{IF } (m,n) \in E_{call} \\
\quad\quad\quad \text{THEN} \\
\quad\quad\quad\quad \mathit{meet} := [\![\, (m,n)_c \,]\!]'(\mathit{inf}[m]) \sqcap \\
\quad\quad\quad\quad\quad\quad \mathit{inf}[\mathit{start}(\mathit{callee}((m,n)))]; \qquad \langle\star\rangle \\
\quad\quad \text{IF } (m,n) \in E_{call} \\
\quad\quad\quad \text{THEN} \\
\quad\quad\quad\quad \mathit{meet} := [\![\, (m,n)_c \,]\!]'(\mathit{inf}[m]) \sqcap \\
\quad\quad\quad\quad\quad\quad \mathit{inf}[\mathit{start}(\mathit{callee}((m,n)))]; \qquad \langle\star\rangle \\
\quad\quad\quad\quad \text{IF } \mathit{inf}[\mathit{start}(\mathit{callee}((m,n)))] \sqsupset \mathit{meet} \\
\quad\quad\quad\quad\quad \text{THEN} \\
\quad\quad\quad\quad\quad\quad \mathit{inf}[\mathit{start}(\mathit{callee}((m,n)))] := \mathit{meet}; \\
\quad\quad\quad\quad\quad\quad \mathit{workset} := \mathit{workset} \cup \{\, \mathit{start}(\mathit{callee}((m,n))) \,\} \\
\quad\quad\quad\quad \text{FI} \\
\quad\quad \text{FI} \\
\quad \text{OD} \\
\text{ESOOHC OD.}
\end{array}
$$

Stackless $IMaxFP_{Stk}$ Alg. 8.2.5.3 / "Functional" Main Process

Input: (1) A flow-graph system S, (2) an abstract semantics consisting of a data-flow lattice $\mathcal{C}$, and the data-flow functionals $[\![\ ]\!] =_{df} gtr$ and $[\ ] =_{df} ltr$ with respect to $\mathcal{C}$ (computed by Algorithm 8.5.2.1), and (4) a context information $c_{\mathbf{s}} \in \mathcal{C}$.

Output: Under the assumption of termination (cf. Theorem 8.5.2.4), the $IMaxFP_{StkLss}$-solution. Depending on the properties of the data-flow functional, this has the following interpretation:
(1) $[\![\ ]\!]$ is distributive: variable *inf* stores for every node the strongest component information valid there with respect to the context information $c_{\mathbf{s}}$.
(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.

Stackless $IMaxFP_{Stk}$-Alg. 8.2.5.3 / "Functional" Main Process

(Prologue: Initialization of the annotation array *inf*, and the variable *workset*)
FORALL $\mathbf{s} \in \{\mathbf{s}_i \mid i \in \{1, \ldots, k\}\}$ DO $inf[\mathbf{s}] := \top$ OD;
$inf[\mathbf{s}_0] := c_{\mathbf{s}}$;
$workset := \{\mathbf{s}_i \mid i \in \{1, 2, \ldots, k\}\}$;

Stackless $\mathit{IMaxFP}_{\mathit{Stk}}$-Alg. 8.2.5.3 / "Functional" Main Process

(Main process: Iterative fixed point computation)

WHILE $\mathit{workset} \neq \emptyset$ DO

CHOOSE $\mathbf{s} \in \mathit{workset}$;

$\mathit{workset} := \mathit{workset} \backslash \{\mathbf{s}\}$;

$\mathit{meet} := \mathit{inf}[\mathbf{s}] \sqcap$

$\bigsqcap \{ [\![\, e_c \,]\!]' \circ [\![\![\, \mathit{src}(e) \,]\!]\!] (\mathit{inf}[\mathit{start}(\mathit{flowGraph}(e))]) \mid$

$e \in \mathit{caller}(\mathit{flowGraph}(\mathbf{s})) \}$; $\langle \star \rangle$

IF $\mathit{inf}[\mathbf{s}] \sqsupset \mathit{meet}$

THEN

$\mathit{inf}[\mathbf{s}] := \mathit{meet}$;

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

$\{ \mathit{start}(\mathit{callee}(e)) \mid e \in E_{\mathit{call}}.$

$\mathit{flowGraph}(e) = \mathit{flowGraph}(\mathbf{s}) \}$

FI

ESOOHC

OD;

Stackless $IMaxFP_{Stk}$-Alg. 8.2.5.3 / "Functional" Main Process

(Epilogue)

FORALL $n \in N \backslash \{\mathbf{s}_i \mid i \in \{0, \ldots, k\}\}$ DO
    $inf[n] := [\![\, n \,]\!](inf[start(flowGraph(n))])$
OD.

Termination

Theorem (8.5.2.4, Termination)

The sequential composition of Algorithm 8.5.2.1 and Algorithmus 8.5.2.2 resp. Algorithm 8.5.2.3 terminates with the IMaxFP$_{Stk}$ solution, if the DFA functional $[\![\ ]\!]'$ and the return functional $\mathcal{R}$ are monotonic and the lattice of functions $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition.

Note: If $[\mathcal{C} \rightarrow \mathcal{C}]$ satisfies the descending chain condition, then $\mathcal{C}$ does so as well.

Chapter 8.3

Extensions

Extensions

- Further parameter transfer mechanisms
  - Reference parameters
  - Procedural parameters, for short: procedure parameters
- Static nesting of procedures

Reference Parameters

Intuitively:

- The effect of reference parameters is encoded in the local semantic functionals of the application problems.
- Reference parameters can thus be handled and computed by suitable preprocess computing may and must aliases of variables and parameters.
- The computed alias information is then fed into the definitions of the local semantics functions of the application problems (cf. Chapter 8.4)

Procedure Parameter

Intuitively:

- A formal procedure call is replaced by the set of all ordinary procedure calls that it may call.
- This set of procedures can be computed by a suitable preprocess; dependingy on the program or programming language class this can be either a safe approximation or an exact solution.
- The computed calling information for formal procedure call reduces then the analysis of programs w/ formal procedure calls to the analysis of programs w/out formal procedure calls.

Static Procedure Nesting

Various variants are possible.

- De-nesting of procedures by a suitable preprocess; this way the analysis of programs w/ static procedure nesting is reduced to analysing programs w/out static procedure nesting.
- Taking into account the effect of relatively global variables in the definition of the local semantics functions of the application problems.

Chapter 8.4

Applications

Preliminaries

In the following we assume:

- No static procedure nesting, no procedure parameters.
- $MstAliases_G(v)$ und $MayAliases_G(v)$ denote the sets of must-Aliases and may-Aliases different from v.

These notions can straightforward be extended to terms t:

- A term t' is a must-alias (may-alias) of t, if t' results from t by replacing of variables by variables that are must-aliases (may-aliases) of each other.

This allows us to feed alias information in a parameterized fashion into the definitions of DFA functionals and return functionals and to take their effects during the analysis into account.

Notations (1)

- $\mathit{GlobVar}(S)$: 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.
- $\mathit{Var}(t)$: the set of variables occurring in t.
- $\mathit{LhsVar}(e)$: the left hand side variable of the assignment of edge e.
- $\mathit{GlobId}(t)$ and $\mathit{LocId}(t)$: abbreviations of $\mathit{GlobVar}(S) \cap \mathit{Var}(t)$ and $\mathit{Var}(t) \backslash \mathit{GlobVar}(S)$.

Notations (2)

- $\mathit{NoGlobalChanges} : E^* \to \mathbb{B}$: indicates that if a variable $v \in \mathit{Var}(t)$ is modified by e, then this modification will not be visible after finishing the call as the relevant memory location of v is local for the currently active call.
- $\mathit{PotAccessible} : S \to \mathbb{B}$: indicates that the memory locations of all variables $v \in \mathit{Var}(t)$, 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 relies on the predicates $Transp_{LocId}$ and $Transp_{GlobId}$ that are defined as follows:

$$Transp_{LocId}(e) =_{df} \; LocId(t) \cap MayAliases_{flowGraph(e)}(LhsVar(e)) = \emptyset$$

$$Transp_{GlobId}(e) =_{df} GlobId(t) \cap (LhsVar(e) \cup MayAliases_{flowGraph(e)}(LhsVar(e))) = \emptyset$$

This allows us to define:

$$\forall\, e \in E^*.\; NoGlobalChanges(e) =_{df} \begin{cases} \textbf{true} & \text{if } e \in E_c^* \cup E_r^* \\ Transp_{LocId}(n) \;\wedge\; Transp_{GlobId}(n) & \text{otherwise} \end{cases}$$

Alias-Information Parameterized Local Predicates (1)

$$\forall\, e \in E^*.\; A\text{-}Comp_{\,e} =_{df} Comp_{\,e} \;\vee\; Comp_{\,e}^{MstAI}$$

$$\forall\, e \in E^*.\, A\text{-}Transp_{\,e} =_{df}\; Transp_{\,e} \;\wedge\; \begin{cases} \mathbf{true} & \text{if } e \in E^*_{call} \\ Transp_{\,e}^{MayAI} & \text{otherwise} \end{cases}$$

Alias-Information Parameterized Local Predicates (2)

Intuitively:

- $A\text{-}Comp_e$ is true for t, if t itself (i.e., $Comp_e$) or one of its must-aliases is computed at edge e (i.e., $Comp_e^{MstAl}$).
- $A\text{-}Transp_e$, $e \in E^* \backslash E^*_{call}$, is true, if neither an operand of t (i.e., $Transp_e$) nor one of its may-aliases is modified by the statement at edge e (i.e., $Transp_e^{MayAl}$).
- For call and return edges $e \in E^*_{call}$, $A\text{-}Transp_e$ is true, if no operand of t is modified (i.e., $Transp_e$). This makes the difference between ordinary assignments and reference parameters and parameter transfers to reference parameters; the latter are updates of pointers leaving the memory except of that invariant.

Finally

- $\mathcal{B}_X =_{df} \{\textbf{false}, \textbf{true}, \textit{failure}\}$

Note: The element *failure* is introduced as an artifical $\top$-element in $\mathbb{B}$ in order to be prepared for reverse data flow analysis as required for demand-driven data flow analysis (cf. LVA 185.276 Analysis and Verification).

Interprocedural Availability (1)

Local Abstract Semantics:

1. Data flow lattice:
   $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{B}^2_X, \wedge, \vee, \leq, (\textbf{false}, \textbf{false}), (\mathit{failure}, \mathit{failure}))$
2. Data flow functional: $[\![\ ]\!]'_{av} : E^* \rightarrow (\mathcal{B}^2_X \rightarrow \mathcal{B}^2_X)$ defined by

$$\forall\, e \in E^*\ \forall\, (b_1, b_2) \in \mathcal{B}^2_X.\ [\![\, e\,]\!]'_{av}(b_1, b_2) =_{df} (b'_1, b'_2)$$

where

$$b'_1 =_{df} \mathit{A\text{-}Transp}_{\,e} \wedge (\mathit{A\text{-}Comp}_{\,e} \vee b_1)$$

$$b'_2 =_{df} \begin{cases} b_2 \wedge \mathit{NoGlobalChanges}_e & \text{if } e \in E^* \backslash E^*_c \\ \textbf{true} & \text{otherwise} \end{cases}$$

Interprocedural Availability (2)

3. Return functional: $\mathcal{R}_{av} : E_{call} \to (\mathcal{B}_X^2 \times \mathcal{B}_X^2 \to \mathcal{B}_X^2)$ defined by $\forall\, e \in E_{call}\ \forall\, ((b_1, b_2), (b_3, b_4)) \in \mathcal{B}_X^2 \times \mathcal{B}_X^2$.
   $\mathcal{R}_{av}(e)((b_1, b_2), (b_3, b_4)) =_{df} (b_5, b_6)$ where

$$b_5 =_{df} \begin{cases} b_3 & \text{if } \mathit{PotAccessible}(\mathit{callee}(e)) \\ (b_1 \vee \mathit{A\text{-}Comp}_{\,e}) \wedge b_4 & \text{otherwise} \end{cases}$$

$$b_6 =_{df} b_2 \wedge b_4$$

Interprocedural Availability (3)

Lemma (8.4.1)

1. *The lattice $\mathcal{B}_X^2$ and the induced lattice of functions satisfy the descending chain condition.*
2. *The functionals $[\![\ ]\!]_{av}'$ and $\mathcal{R}_{av}$ are distributive.*

$\rightsquigarrow$ Hence, the preconditions of the Interprocedural Coincidence Theorem and the Termination Theorem are satisfied.

Interprocedural Simple Constants

Local Abstract Semantics:

1. Data flow lattice:
   $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\Sigma_X, \sqcap, \sqcup, \sqsubseteq, \sigma_\bot, \sigma_{failure})$
2. Data flow functional: $[\![\ ]\!]'_{sc} : E \to (\Sigma_X \to \Sigma_X)$ defined by
   $\forall\, e \in E.\ [\![\, e\,]\!]'_{sc} =_{df} \theta_e$
3. Return functional: $\mathcal{R}_{sc} : E_{call} \to (\Sigma_X \times \Sigma_X \to \Sigma_X)$
   defined by

   $$\forall\, e \in E_{call}\ \forall\, (\sigma_1, \sigma_2)) \in \Sigma_X \times \Sigma_X.\ \mathcal{R}_{sc}(e)(\sigma_1, \sigma_2) =_{df} \sigma_3$$

   where

   $$\forall\, x \in Var.\ \sigma_3(x) =_{df} \begin{cases} \sigma_2(x) & \text{if } x \in GlobVar(S) \\ \sigma_1(x) & \text{otherwise} \end{cases}$$

Problems and Solutions/Work-Arounds

In practice

- the preceding analysis specification for simple interprocedural constants does not induce a terminating analysis since the lattice of functions does not satisfy the descending chain condition
- thus simpler constant propagation problems are considered like copy constants and linear constants

Copy Constants and Linear Constants

A term is a

- copy constant at a program point, if it is a source-code constant or an operator-less term that is itself a copy constant
- linear constant at a program point, if it is a source-code constant or of the form $a * x + b$ w/ a, b source-code constants and x a linear constant.

Interprocedural Copy Constants (1)

The specification of copy constants is based on the following simpler evaluation function of terms:

$$\mathcal{E}_{cc} : \mathbf{T} \to (\Sigma_X \to \mathbf{D})$$

$\mathcal{E}_{cc}$ is undefined for the failure state $\sigma_{failure}$; otherwise it is defined as follows:

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

Note that Σ_X is analogously to $\mathcal{B}_X$ extended by an artificial top-element.

Interprocedural Copy Constants (2)

- Replacing θ_ι in $\mathcal{E}$ by $\mathcal{E}_{cc}$ yields the data flow analysis functional $[\![\ ]\!]'_{cc}$.
- Replacing of $[\![\ ]\!]'_{sc}$ by $[\![\ ]\!]'_{cc}$ yields the definition of the local abstract semantics of interprocedural copy constants.

Interprocedural Copy Constants (3)

Note:

- The number of source-code constants is finite.
- Hence, the lattice of functions that belongs to the relevant sublattice Σ_{cc_X} of Σ_X satisfies the descending chain condition.
- Thus, the *IMaxFP* algorithm terminates.
- Unlike as interprocedural simple constants are copy constants distributive; thus, the $IMaxFP_{Stk}$ solution and the $IMOP_{Stk}$ solution coincide.

Interprocedural Copy Constants (4)

Lemma (8.4.2)

1. *The lattice Σ_{cc_X} and the induced lattice of functions satisfy the descending chain condition.*
2. *The functionals $[\![\ ]\!]'_{cc}$ and $\mathcal{R}_{cc}$ are distributive.*

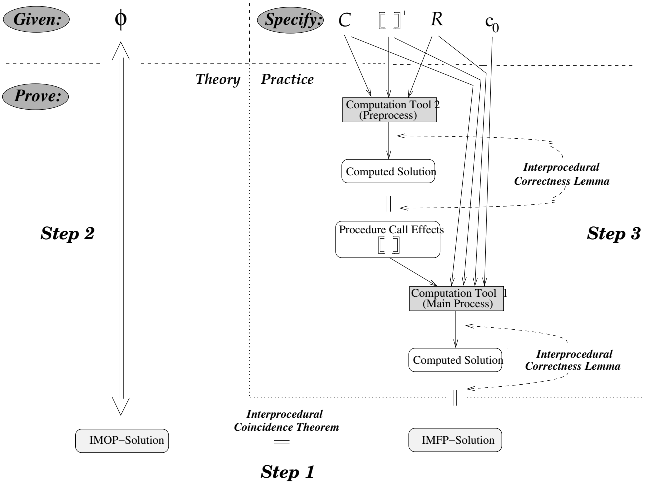
Therefore, the preconditions of the Interprocedural Coincidence Theorem 8.2.4.4 and the Termination Theorem are satisfied.

Chapter 8.5

Interprocedural DFA: Framework and Toolkit

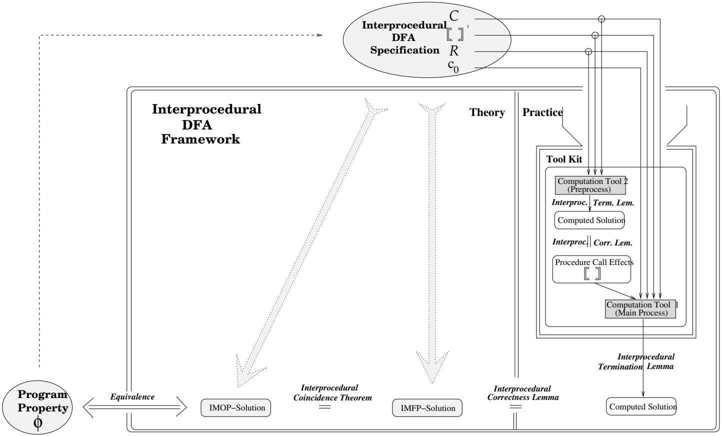
Interprocedural DFA: The Framework View

The interprocedural DFA Framework at a glance:



Interprocedural DFA: The Toolkit View

The Toolkit View of the interprocedural DFA Framework:



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 12, Interprocedural Analysis)
- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002. (Chapter 11, Interprocedural Analysis and Optimization)
- Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-Verlag, LNCS 1428, 1998. (Chapter 10, Interprocedural Code Motion: The Transformations, Chapter 11, Interprocedural Code Motion: The IDFA-Algorithms)

Further Reading for Chapter 8 (2)

- Jens Knoop. *Formal Callability and its Relevance and Application to Interprocedural Data Flow Analysis*. In Proceedings of the 6th IEEE International Conference on Computer Languages (ICCL'98), 252-261, 1998.
- Jens Knoop. *From DFA-Frameworks to DFA-Generators: A Unifying Multiparadigm Approach*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Springer-Verlag, 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-Verlag, LNCS 641, 125-140, 1992.

Chapter 9

IDFA – The Call String Approach

See Separate Slide Package.

Part IV

Extensions, Other Settings

Chapter 10
Aliasing

See Separate Slide Package.

Chapter 11

Optimizations for Object-Oriented Languages

See Separate Slide Package.

Chapter 12

Slicing

See Separate Slide Package.

Part V

Conclusions and Prospectives

Chapter 13

Summary and Outlook

Ein Fazit bzw...

Die Frage nach dem Sinn des Lebens, was haben wir alles erreicht bzw...

- Was haben wir alles betrachtet?
  Das wenigste!

Oder umgekehrt...

- Was haben wir alles nicht betrachtet?
  Das meiste!

Insbesondere nicht (oder nicht im Detail) (1)

- *Erweiterungen syntaktischer PRE neben PDCE/PRAE*
  - Lazy Strength Reduction
  - ...
- *Semantische Erweiterungen*
  - Semantic Code Motion/Code Placement
  - Semantic Strength Reduction
  - ...
- *Sprachausweitungen*
  - Interprozeduralität
  - Parallelität
  - ...

Insbesondere nicht (oder nicht im Detail) (1)

- *Dynamische, profilgestützte Erweiterungen*
  - Spekulative PRE
  - ...
- ...

Literaturhinweise (1)

- *Syntaktische PRE*

  - Knoop, J., Rüthing, O., and Steffen, B. Retrospective: Lazy Code Motion. In "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection", ACM SIGPLAN Notices 39, 4 (2004), 460 - 461 & 462-472.
  - Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. ACM Transactions on Programming Languages and Systems 16, 4 (1994), 1117 - 1155.
  - Rüthing, O., Knoop, J., and Steffen, B. Sparse code motion. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000) (Boston, MA, Jan. 19 - 21, 2000), ACM New York, (2000), 170 - 183.

Literaturhinweise (2)

- *Elimination partiell toten Codes*
  - Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94) (Orlando, FL, USA, June 20 - 24, 1994), ACM SIGPLAN Notices 29, 6 (1994), 147 - 158.
- *Elimination partiell redundanter Anweisungen*
  - Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95) (La Jolla, CA, USA, June 18 - 21, 1995), ACM SIGPLAN Notices 30, 6 (1995), 233 - 245.

Literaturhinweise (3)

- *BB- vs. EA-Graphen*
  - Knoop, J., Koschützki, D., and Steffen, B. Basic-block graphs: Living dinosaurs? In Proceedings of the 7th International Conference on Compiler Construction (CC'98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1383 (1998), 65 - 79.
- *Schieben vs. Platzieren*
  - Knoop, J., Rüthing, O., and Steffen, B. Code motion and code placement: Just synonyms? In Proceedings of the 7th European Symposium On Programming (ESOP'98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1381 (1998), 154 - 169.

Literaturhinweise (4)

- *Spekulative vs. klassische PRE*
  - Scholz, B., Horspool, N. and Knoop, J. Optimizing for space and time usage with speculative partial redundancy elimination. In Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004) (Washington, DC, June 11 - 13, 2004), ACM SIGPLAN Notices 39, 7 (2004), 221 -230.
  - Xue, J., Knoop, J. A fresh look at PRE as a maximum flow problem. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006) (Vienna, Austria, March 25 - April 2, 2006), Springer-Verlag, Heidelberg, LNCS 3923 (2006), 139 - 154.

Literaturhinweise (5)

- *Weitere Techniken und spezielle Verfahren*
  - Geser, A., Knoop, J., Lüttgen, G., Rüthing, O., and Steffen, B. Non-monotone fixpoint iterations to resolve second order effects. In Proceedings of the 6th International Conference on Compiler Construction (CC'96) (Linköping, Sweden, April 24 - 26, 1996), Springer-Verlag, Heidelberg, LNCS 1060 (1996), 106 - 120.
  - Knoop, J., and Mehofer, E. Optimal distribution assignment placement. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97) (Passau, Germany, August 26 - 29, 1997), Springer-V., Heidelberg, LNCS 1300 (1997), 364 - 373.
  - Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. Journal of Programming Languages 1, 1 (1993), 71 - 91.
  - ...: siehe auch `www.complang.tuwien.ac.at/knoop`

Further Reading for Chapter 13

Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-Verlag, 2005. (Chapter 1, Introduction; Chapter 2, Data Flow Analysis; Chapter 6, Algorithms)

Bibliography

Recommended Reading

...for deepened and independent studies.

- I Textbooks
- II Monographs
- III Volumes
- III Articles

I Textbooks (1)

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2002.
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufman Publishers, 2004.
- Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.
- Janusz Laski, William Stanley. *Software Verification and Analysis*. Springer-Verlag, 2009.

I Textbooks (2)

- Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag, 2007.
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-Verlag, 2005.
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-Verlag, 2006.

II Monographs

- Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and Its Application*. Springer-Verlag, LNCS 1428, 1998.
- Stephen S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1997.

III Volumes

- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 1st edition, CRC Press, 2002.
- Y. N. Srikant, Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. 2nd edition, CRC Press, 2008.

III Articles (1)

- Andrei P. Ershov. *On Programming of Arithmetic Operations*. Communications of the ACM 1(8):3-6, 1958.
- John B. Kam, Jeffrey D. Ullman. *Monotone Data Flow Analysis Frameworks*. Acta Informatica 7:305-317, 1977.
- Gary A. Kildall. *A Unified Approach to Global Program Optimization*. In Conference Record of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73), 194-206, 1973.
- 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.

III Articles (2)

- 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-Verlag, 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-Verlag, LNCS 1383, 65-79, 1998).
- Jens Knoop, Oliver Rüthing, Bernhard Steffen. *Lazy Code Motion*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), ACM SIGPLAN Notices 27(7):224-234, 1992.

III Articles (3)

- 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-Verlag, 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-Verlag, LNCS 1575, 91-106, 1999.

III Articles (4)

- 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.

III Articles (5)

- Jens Knoop, Bernhard Steffen. *The Interprocedural Coincidence Theorem*. In Proceedings of the 4th International Conference on Compiler Construction (CC'92), Springer-Verlag, 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.
- Etienne Morel, Claude Renvoise. *Global Optimization by Suppression of Partial Redundancies*. Communications of the ACM 22(2):96-103, 1979.

III Articles (6)

- 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 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-Verlag, LNCS 249, 52-68, 1987.
- Bernhard Steffen. *Property-Oriented Expansion*. In Proceedings of the 3rd Static Analysis Symposium (SAS'96), Springer-Verlag, LNCS 1145, 22-41, 1996.

III Articles (7)

- 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-Verlag, LNCS 432, 389-405, 1990.
- 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-Verlag, LNCS 494, 394-415, 1991.

Appendix

A

Mathematical Foundations

A.1

Sets and Relations

Sets and Relations

Let M be a set and R a relation on M, i.e. $R \subseteq M \times M$.

Then R is called

- reflexive iff $\forall\, m \in M.\ m\, R\, m$
- transitive iff $\forall\, m, n, p \in M.\ m\, R\, n\ \wedge\ n\, R\, p\ \Rightarrow\ m\, R\, p$
- anti-symmetric iff $\forall\, m, n \in M.\ m\, R\, n\ \wedge\ n\, R\, m\ \Rightarrow\ m = n$

Related notions (though less important for us here)...

- symmetric iff $\forall\, m, n \in M.\ m\, R\, n\ \iff\ n\, R\, m$
- total iff $\forall\, m, n \in M.\ m\, R\, n\ \vee\ n\, R\, m$

A.2

Partially Ordered Sets

Partially Ordered Sets

A relation R on M is called a

- quasi-order iff R is reflexive and transitive
- partial order iff R is reflexive, transitive, and anti-symmetric

For the sake of completeness we recall

- equivalence relation iff R is reflexive, transitive, and symmetric

...i.e., a partial order is an anti-symmetric quasi-order, an equivalence relation a symmetric quasi-order.

Note: We here use terms like "partial order" as a short hand for the more accurate term "partially ordered set."

Bounds, least and greatest Elements

Let $(Q, \sqsubseteq)$ be a quasi-order, let $q \in Q$ and $Q' \subseteq Q$.

Then q is called

- upper (lower) bound of Q', in signs: $Q' \sqsubseteq q$ $(q \sqsubseteq Q')$, if for all $q' \in Q'$ holds: $q' \sqsubseteq q$ $(q \sqsubseteq q')$
- least upper (greatest lower) bound of Q', if q is an upper (lower) bound of Q' and for every other upper (lower) bound $\hat{q}$ of Q' holds: $q \sqsubseteq \hat{q}$ $(\hat{q} \sqsubseteq q)$
- greatest (least) element of Q, if holds: $Q \sqsubseteq q$ $(q \sqsubseteq Q)$

Uniqueness of Bounds

- Given a partial order, least upper and greatest lower bounds are uniquely determined, if they exist.
- Given existence (and thus uniqueness), the least upper (greatest lower) bound of a set $P' \subseteq P$ of the basic set of a partial order $(P, \sqsubseteq)$ is denoted by $\bigsqcup P'$ ($\bigsqcap P'$). These elements are also called supremum and infimum of P'.
- Analogously this holds for least and greatest elements. Given existence, these elements are usually denoted by $\perp$ and $\top$.

A.3
Lattices

Lattices and Complete Lattices

Let $(P, \sqsubseteq)$ be a partial order.

Then $(P, \sqsubseteq)$ is called a

- lattice, if each finite subset P' of P contains a least upper and a greatest lower bound in P
- complete lattice, if each subset P' of P contains a least upper and a greatest lower bound in P

...(complete) lattices are special partial orders.

A.4

Complete Partially Ordered Sets

Complete Partial Orders

...a slightly weaker notion that in computer science, however, is often sufficient and thus often a more adequate notion:

Let $(P, \sqsubseteq)$ be a partial order.

Then $(P, \sqsubseteq)$ is called

- complete, or shorter a CPO (complete partial order), if each ascending chain $C \subseteq P$ has a least upper bound in P.

We have:

- A CPO $(C, \sqsubseteq)$ (more accurate would be: "chain-complete partially ordered set (CCPO)") has always a least element. This element is uniquely determined as supremum of the empty chain and usually denoted by $\bot$: $\bot =_{df} \bigsqcup \emptyset$.

Chains

Let $(P, \sqsubseteq)$ be a partial order.

A subset $C \subseteq P$ is called

- chain of P, if the elements of C are totally ordered. For $C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \ldots\}$ ($\{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \ldots\}$) we also speak more precisely of an ascending (descending) chain of P.

A chain C is called

- finite, if C is finite; infinite otherwise.

Finite Chains, finite Elements

A partial order $(P, \sqsubseteq)$ is called

- chain-finite (German: kettenendlich) iff P is free of infinite chains

An element $p \in P$ is called

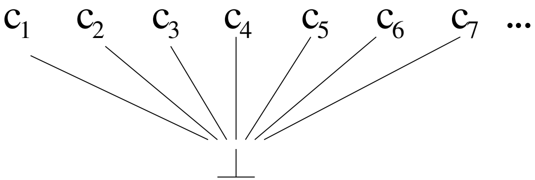
- finite iff the set $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ is free of infinite chains
- finite relative to $r \in P$ iff the set $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ is free of infinite chains

(Standard) CPO Constructions 1(4)

Flat CPOs:

Let $(C, \sqsubseteq)$ be a CPO. Then $(C, \sqsubseteq)$ is called

- flat, if for all $c, d \in C$ holds: $c \sqsubseteq d \Leftrightarrow c = \perp \ \vee \ c = d$



(Standard) CPO Constructions 2(4)

Product construction.

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ be CPOs. Then...

- the non-strict (direct) product $(\times P_i, \sqsubseteq)$ with
  - $(\times P_i, \sqsubseteq) = (P_1 \times P_2 \times \ldots \times P_n, \sqsubseteq)$ with $\forall\, (p_1, p_2, \ldots, p_n), (q_1, q_2, \ldots, q_n) \in \times P_i.\ (p_1, p_2, \ldots, p_n) \sqsubseteq (q_1, q_2, \ldots, q_n) \Leftrightarrow \forall\, i \in \{1, \ldots, n\}.\ p_i \sqsubseteq_i q_i$
- and the strict (direct) product (smash product) with
  - $(\bigotimes P_i, \sqsubseteq) = (P_1 \otimes P_2 \otimes \ldots \otimes P_n, \sqsubseteq)$, where $\sqsubseteq$ is defined as above under the additional constraint:

$$(p_1, p_2, \ldots, p_n) = \bot \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\ p_i = \bot_i$$

are CPOs, too.

(Standard) CPO Constructions 3(4)

Sum construction.

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)$ CPOs. Then

- the direct sum $(\bigoplus P_i, \sqsubseteq)$ with...
  - $(\bigoplus P_i, \sqsubseteq) = (P_1 \dot\cup P_2 \dot\cup \ldots \dot\cup P_n, \sqsubseteq)$ disjoint union of P_i, $i \in \{1, \ldots, n\}$ and $\forall\, p, q \in \bigoplus P_i.\ p \sqsubseteq q \Leftrightarrow \exists\, i \in \{1, \ldots, n\}.\ p, q \in P_i \ \wedge\ p \sqsubseteq_i q$

  is a CPO.

  Note: The least elements of $(P_i, \sqsubseteq_i)$, $i \in \{1, \ldots, n\}$ are usually identified, i.e., $\bot =_{df} \bot_i$, $i \in \{1, \ldots, n\}$

(Standard) CPO Constructions 4(4)

Function space.

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be two CPOs and $[C \to D] =_{df} \{f : C \to D \mid f \text{ continuous}\}$ the set of continuous functions from C to D.

Then

- the continuous function space $([C \to D], \sqsubseteq)$ is a CPO where
  - $\forall\, f, g \in [C \to D].\ f \sqsubseteq g \iff \forall\, c \in C.\ f(c) \sqsubseteq_D g(c)$

Functions on CPOs / Properties

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be two CPOs and let $f : C \to D$ be a function from C to D.

Then f is called

- monotone iff $\forall\, c, c' \in C.\ c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$
  (Preservation of the ordering of elements)
- continuous iff $\forall\, C' \subseteq C.\ f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$
  (Preservation of least upper bounds)

Let $(C, \sqsubseteq)$ be a CPO and let $f : C \to C$ be a function on C. Then f is called

- inflationary (increasing) iff $\forall\, c \in C.\ c \sqsubseteq f(c)$

Functions on CPOs / Results

Using the notations introduced before

Lemma

f is monotone iff $\forall\, C' \subseteq C.\ f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$

Corollary

A continuous function is always monotone, i.e. f continuous $\Rightarrow$ f monotone.

A.5

Fixed Point Theorem

Least and greatest Fixed Points 1(2)

Let $(C, \sqsubseteq)$ be a CPO, $f : C \to C$ be a function on C and let c be an element of C, i.e., $c \in C$.

Then c is called

- fixed point of f iff $f(c) = c$

A fixed point c of f is called

- least fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow c \sqsubseteq d$
- greatest fixed point of f iff $\forall\, d \in C.\ f(d) = d \Rightarrow d \sqsubseteq c$

Least and greatest Fixed Points 2(2)

Let $d, c_d \in C$. Then c_d is called

- conditional (German: bedingter) least fixed point of f wrt d iff c_d is the least fixed point of C with $d \sqsubseteq c_d$, i.e. for all other fixed points x of f with $d \sqsubseteq x$ holds: $c_d \sqsubseteq x$.

Notations:

The least resp. greatest fixed point of a function f is usually denoted by μf resp. νf.

Fixed Point Theorem

Theorem (Knaster/Tarski, Kleene)

Let $(C, \sqsubseteq)$ be a CPO and let $f : C \to C$ be a continuous function on C.

Then f has a least fixed point μf, which equals the least upper bound of the chain (so-called Kleene-Chain) $\{\perp, f(\perp), f^2(\perp), \ldots\}$, i.e.

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{\perp, f(\perp), f^2(\perp), \ldots\}$$

Proof of the Fixed Point Theorem 1(4)

We have to prove: μf

1. exists
2. is a fixed point
3. is the least fixed point

Proof of the Fixed Point Theorem 2(4)

1. Existence
    - It holds $f^0 \perp = \perp$ and $\perp \sqsubseteq c$ for all $c \in C$.
    - By means of (natural) induction we can show: $f^n \perp \sqsubseteq f^n c$ for all $c \in C$.
    - Thus we have $f^n \perp \sqsubseteq f^m \perp$ for all n, m with $n \leq m$. Hence, $\{f^n \perp \mid n \geq 0\}$ is a (non-finite) chain of C.
    - The existence of $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is thus an immediate consequence of the CPO properties of $(C, \sqsubseteq)$.

Proof of the Fixed Point Theorem 3(4)

2. Fixed point property

$$\begin{array}{rcl} & & f(\bigsqcup_{i\in\mathbb{N}_0} f^i(\bot)) \\ (f \text{ continuous}) & = & \bigsqcup_{i\in\mathbb{N}_0} f(f^i\bot) \\ & = & \bigsqcup_{i\in\mathbb{N}_1} f^i\bot \\ (K \text{ chain} \Rightarrow \bigsqcup K = \bot \sqcup \bigsqcup K) & = & (\bigsqcup_{i\in\mathbb{N}_1} f^i\bot) \ \sqcup \ \bot \\ (f^0\bot = \bot) & = & \bigsqcup_{i\in\mathbb{N}_0} f^i\bot \end{array}$$

Proof of the Fixed Point Theorem 4(4)

3. Least fixed point
   - Let c be an arbitrarily chosen fixed point of f. Then we have $\perp \sqsubseteq c$, and hence also $f^n \perp \sqsubseteq f^n c$ for all $n \geq 0$.
   - Thus, we have $f^n \perp \sqsubseteq c$ because of our choice of c as fixed point of f.
   - Thus, we also have that c is an upper bound of $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$.
   - Since $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is the least upper bound of this chain by definition, we obtain as desired $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c$.

Conditional Fixed Points

Theorem (Conditional Fixed Points)

Let $(C, \sqsubseteq)$ *be a CPO, let* $f : C \to C$ *be a continuous, inflationary function on* C*, and let* $d \in C$*.*

Then f *has a unique conditional fixed point* μf_d*. This fixed point equals the least upper bound of the chain* $\{d, f(d), f^2(d), \ldots\}$*, i.e.*

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \ldots\}$$

Finite Fixed Points

Theorem (Finite Fixed Points)

Let $(C, \sqsubseteq)$ be a CPO and let $f : C \to C$ be a continuous function on C.

Then we have: If two elements in a row occurring in the Kleene-chain of f are equal, e.g. $f^i(\bot) = f^{i+1}(\bot)$, then we have: $\mu f = f^i(\bot)$.

Existence of Finite Fixed Points

Sufficient conditions for the existence of finite fixed points e.g. are

- Finiteness of domain and range of f
- f is of the form $f(c) = c \sqcup g(c)$ for monotone g on some chain-complete domain

Appendix A: Further Reading

- 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-Verlag, 2007. (Chapter 5, Denotational Semantics)
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-Verlag, 2005. (Appendix A, Partially Ordered Sets)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-Verlag, 2006. (Chapter 10, Beispiel: Berechnung von Fixpunkten)