Program Optimization

Peter Lammich

WS 2016/17

Overview by Lecture

Organizational Issues

Lectures Wed 10:15-11:45 and Thu 10:15-11:45 in MI 00.13.009A

Organizational Issues

Lectures Wed 10:15-11:45 and Thu 10:15-11:45 in MI 00.13.009A

Tutorial Fri 8:30-10:00 (Ralf Vogler <ralf.vogler@mytum.de>)

- Homework will be corrected

Organizational Issues

Lectures	Wed 10:15-11:45 and Thu 10:15-11:45 in MI 00.13.009A
Tutorial	Fri 8:30-10:00 (Ralf Vogler <ralf.vogler@mytum.de>)
	• Homework will be corrected
Exam	Written (or Oral), Bonus for Homework!
	• $\geq$ 50% of homework $\Longrightarrow$ 0.3/0.4 better grade On first exam attempt. Only if passed w/o bonus!

Organizational Issues

Lectures Wed 10:15-11:45 and Thu 10:15-11:45 in MI 00.13.009A

Tutorial Fri 8:30-10:00 (Ralf Vogler <ralf.vogler@mytum.de>)

- Homework will be corrected

Exam Written (or Oral), Bonus for Homework!

- $\geq$ 50% of homework $\implies$ 0.3/0.4 better grade
  On first exam attempt. Only if passed w/o bonus!

Material Seidl, Wilhelm, Hack: Compiler Design: Analysis and Transformation, Springer 2012

Organizational Issues

Lectures Wed 10:15-11:45 and Thu 10:15-11:45 in MI 00.13.009A

Tutorial Fri 8:30-10:00 (Ralf Vogler <ralf.vogler@mytum.de>)

- Homework will be corrected

Exam Written (or Oral), Bonus for Homework!

- $\geq$ 50% of homework $\Longrightarrow$ 0.3/0.4 better grade
  On first exam attempt. Only if passed w/o bonus!

Material Seidl, Wilhelm, Hack: Compiler Design: Analysis and Transformation, Springer 2012

How many of you are attending “Semantics” lecture?

Info-2 Tutors

We need tutors for Info II lecture. If you are interested, please contact Julian Kranz `julian.kranz@in.tum.de`.

Proposed Content

- Avoiding redundant computations
  - E.g. Available expressions, constant propagation, code motion
- Replacing expensive with cheaper computations
  - E.g. peep hole optimization, inlining, strength reduction
- Exploiting Hardware
  - E.g. instruction selection, register allocation, scheduling
- Analysis of parallel programs
  - E.g. threads, locks, data-races

Table of Contents

Observation 1

Intuitive programs are often inefficient

```
void swap (int i, int j) {
  int t;
  if (a[i] > a[j]) {
    t = a[j];
    a[j] = a[i];
    a[i] = t;
  }
}
```

Observation 1

Intuitive programs are often inefficient

```
void swap (int i, int j) {
  int t;
  if (a[i] > a[j]) {
    t = a[j];
    a[j] = a[i];
    a[i] = t;
  }
}
```

- Inefficiencies
  - Addresses computed 3 times
  - Values loaded 2 times

Observation 1

Intuitive programs are often inefficient

```
void swap (int i, int j) {
  int t;
  if (a[i] > a[j]) {
    t = a[j];
    a[j] = a[i];
    a[i] = t;
  }
}
```

- Inefficiencies
  - Addresses computed 3 times
  - Values loaded 2 times
- Improvements
  - Use pointers for array indexing
  - Store the values of `a[i]`, `a[j]`

```
void swap (int *p, int *q) {
  int t, ai, aj;
  ai=*p; aj=*q;
  if (ai > aj) {
    t = aj;
    *q = ai;
    *p = t; // t can also be eliminated
  }
}
```

```c
void swap (int *p, int *q) {
  int ai, aj;
  ai=*p; aj=*q;
  if (ai > aj) {
    *q = ai;
    *p = aj;
  }
}
```

```
void swap (int *p, int *q) {
  int ai, aj;
  ai=*p; aj=*q;
  if (ai > aj) {
    *q = ai;
    *p = aj;
  }
}
```

Caveat: Program less intuitive

Observation 2

High-level languages (even C) abstract from hardware (and efficiency)

Compiler needs to transform intuitively written programs to hardware.

Examples

- Filling of delay slots
- Utilization of special instructions
- Re-organization of memory accesses for better cache behavior
- Removal of (useless) overflow/range checks

Observation 3

Program improvements need not always be correct

- E.g. transform `f() + f()` to `2*f()`

Observation 3

Program improvements need not always be correct

- E.g. transform `f() + f()` to `2*f()`
- Idea: Save second evaluation of `f`

Observation 3

Program improvements need not always be correct

- E.g. transform `f() + f()` to `2*f()`
- Idea: Save second evaluation of `f`
- But what if `f` has side-effects or reads input?

Insight

- Program optimizations have preconditions
- These must be
  - Formalized
  - Checked
- It must be proved that optimization is correct
  - I.e., preserves semantics

Observation 4

Optimizations techniques depend on programming language

- What inefficiencies occur
- How analyzable is the language
- How difficult it is to prove correctness

Example: Java

- (Unavoidable) inefficiencies
  - Array bound checks
  - Dynamic method invocation
  - Bombastic object organization

Example: Java

- (Unavoidable) inefficiencies
  - Array bound checks
  - Dynamic method invocation
  - Bombastic object organization
- Analyzability
  - \+ No pointer arithmetic, no pointers into stack
  - \- Dynamic class loading
  - \- Reflection, exceptions, threads

Example: Java

- (Unavoidable) inefficiencies
  - Array bound checks
  - Dynamic method invocation
  - Bombastic object organization
- Analyzability
  - + No pointer arithmetic, no pointers into stack
  - - Dynamic class loading
  - - Reflection, exceptions, threads
- Correctness proof
  - + Well-defined semantics (more or less)
  - - Features, features, features
  - - Libraries with changing behavior

In this course

- Simple imperative programming language

`R = e`	Assignment
`R = M[e]`	Load
`M[e₁] = e₂` → $M[e_1] = e_2$	Store
`if` `(e) ...` **`else`** `...`	Conditional branching
`goto` `label`	Unconditional branching

R Registers, assuming infinite supply

e Integer-valued expressions over constants, registers, operators

M Memory, addressed by integer $\geq$ 0, assuming infinite memory

Note

- For the beginning, we omit procedures
  - Focus on intra-procedural optimizations
  - External procedures taken into account via statement `f()`
    - unknown procedure
    - may arbitrarily mess around with memory and registers
- Intermediate Language, in which (almost) everything can be translated

Example: Swap

```
void swap (int i, int j) {
  int t;
  if (a[i] > a[j]) {
    t = a[j];
    a[j] = a[i];
    a[i] = t;
  }
}
```

```
A1 = A0 + 1*i   //R1 = a[i]
R1 = M[A1]
A2 = A0 + 1*j   //R2 = a[j]
R2 = M[A2]
if (R1 > R2) {
  A3 = A0 + 1*j //t=a[j]
  t = M[A3]
  A4 = A0 + 1*j //a[j] = a[i]
  A5 = A0 + 1*i
  R3 = M[A5]
  M[A4] = R3
  A6 = A0 + 1*i //a[i]=t
  M[A6] = t
}
```

Example: Swap

```
void swap (int i, int j) {
  int t;
  if (a[i] > a[j]) {
    t = a[j];
    a[j] = a[i];
    a[i] = t;
  }
}
```

Assume A_0 contains address of array a

1: $A_1 = A_0 + 1*i$ *//R_1 = a[i]*
2: $R_1 = M[A_1]$
3: $A_2 = A_0 + 1*j$ *//R_2 = a[j]*
4: $R_2 = M[A_2]$
5: **if** $(R_1 > R_2)$ {
6:    $A_3 = A_0 + 1*j$ *//t=a[j]*
7:    $t = M[A_3]$
8:    $A_4 = A_0 + 1*j$ *//a[j] = a[i]*
9:    $A_5 = A_0 + 1*i$
0:    $R_3 = M[A_5]$
1:    $M[A_4] = R_3$
2:    $A_6 = A_0 + 1*i$ *//a[i]=t*
3:    $M[A_6] = t$
}

Optimizations

1. $\texttt{1 * R} \mapsto \texttt{R}$
2. Re-use of sub-expressions

$$\texttt{A}_1 \texttt{ == } \texttt{A}_5\texttt{== } \texttt{A}_6\texttt{, }\texttt{A}_2 \texttt{ == } \texttt{A}_3\texttt{== } \texttt{A}_4$$
$$\texttt{M[A}_1\texttt{] == M[A}_5\texttt{],M[A}_2\texttt{] == M[A}_3\texttt{]}$$
$$\texttt{R}_1 \texttt{ == } \texttt{R}_3$$
$$\texttt{R}_2 \texttt{ = t}$$

Now we have

```
A1 = A0 + i
R1 = M[A1]
A2 = A0 + j
R2 = M[A2]
if (R1 > R2) {
  M[A2] = R1
  M[A1] = R2
}
```

Original was:

```
A1 = A0 + 1*i   //R1 = a[i]
R1 = M[A1]
A2 = A0 + 1*j   //R2 = a[j]
R2 = M[A2]
if (R1 > R2) {
  A3 = A0 + 1*j //t=a[j]
  t = M[A3]
  A4 = A0 + 1*j //a[j] = a[i]
  A5 = A0 + 1*i
  R3 = M[A5]
  M[A4] = R3
  A6 = A0 + 1*i //a[i]=t
  M[A6] = t
}
```

Gain

	before	after
$+$	6	2
$*$	6	0
$>$	1	1
load	4	2
store	2	2
$R =$	6	2

Table of Contents

Table of Contents

Idea

If same value is computed repeatedly

- Store it after first computation
- Replace further computations by look-up

Idea

If same value is computed repeatedly

- Store it after first computation
- Replace further computations by look-up

Method

- Identify repeated computations
- Memorize results
- Replace re-computation by memorized value

Example

```
    x = 1
    y = M[42]
A:  r1 = x + y
    ...
B:  r2 = x + y
```

Example

```
     x  =  1
     y  =  M[42]
A:   r1 = x + y
     ...
B:   r2 = x + y
```

- Repeated computation of `x+y` at `B`, if
  - `A` is always executed before `B`
  - `x+y` has the same value at *A* and *B*.

Example

```
    x = 1
    y = M[42]
A:  r1 = x + y
    ...
B:  r2 = x + y
```

- Repeated computation of `x+y` at `B`, if
  - `A` is always executed before `B`
  - `x+y` has the same value at *A* and *B*.
- We need
  - Operational semantics
  - Method to identify (at least some) repeated computations

Table of Contents

Rice's theorem (informal)

All non-trivial semantic properties of a Turing-complete programming language are undecidable.

Rice's theorem (informal)

All non-trivial semantic properties of a Turing-complete programming language are undecidable.

Consequence We cannot write the ideal program optimizer :(

Rice's theorem (informal)

All non-trivial semantic properties of a Turing-complete programming language are undecidable.

Consequence We cannot write the ideal program optimizer :(

But Still can use approximate approaches

- Approximation of semantic property
- Show that transformation is still correct

Rice's theorem (informal)

All non-trivial semantic properties of a Turing-complete programming language are undecidable.

Consequence We cannot write the ideal program optimizer :(

But Still can use approximate approaches

- Approximation of semantic property
- Show that transformation is still correct

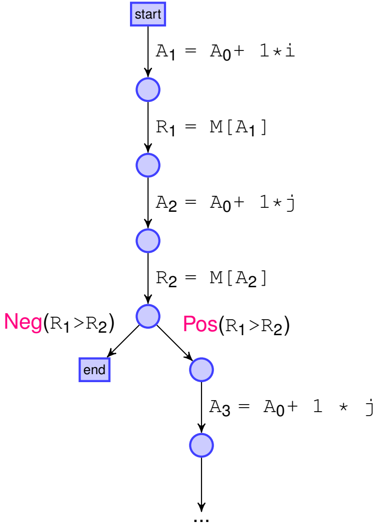
Example: Only identify subset of repeated computations.

Table of Contents

Small-step operational semantics

Intuition: Instructions modify state (registers, memory)

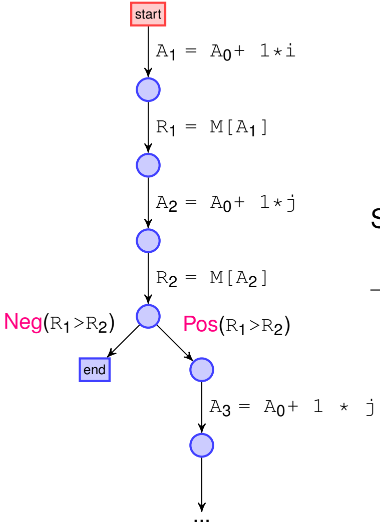
Represent program as control flow graph (CFG)



Small-step operational semantics

Intuition: Instructions modify state (registers, memory)

Represent program as control flow graph (CFG)



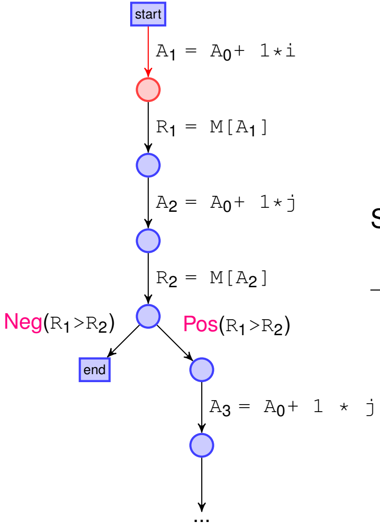
State:

A_0	M[0..4]	i	j	A_1	A_2	R_1	R_2
0	1,2,3,4,5	2	4	-	-	-	-

Small-step operational semantics

Intuition: Instructions modify state (registers, memory)

Represent program as control flow graph (CFG)



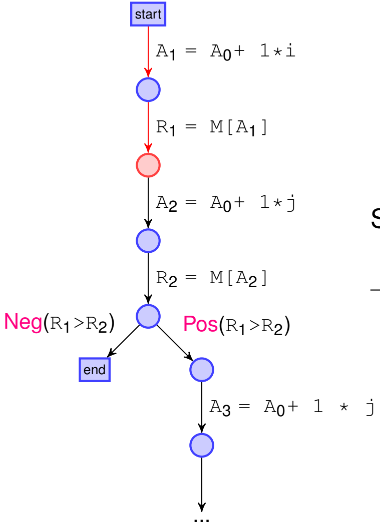
State:

A_0	M[0..4]	i	j	A_1	A_2	R_1	R_2
0	1,2,3,4,5	2	4	2	-	-	-

Small-step operational semantics

Intuition: Instructions modify state (registers, memory)

Represent program as control flow graph (CFG)



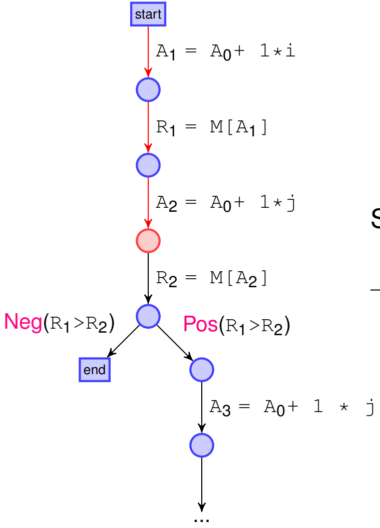
State:

A_0	M[0..4]	i	j	A_1	A_2	R_1	R_2
0	1,2,3,4,5	2	4	2	-	3	-

Small-step operational semantics

Intuition: Instructions modify state (registers, memory)

Represent program as control flow graph (CFG)



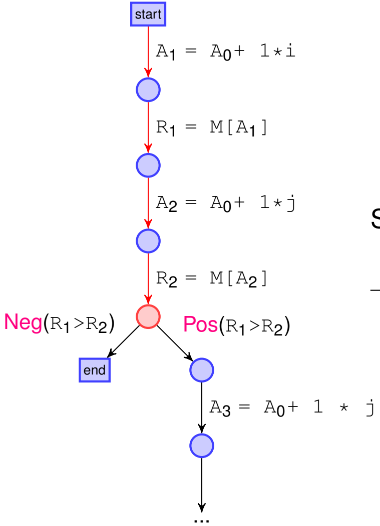
State:

A_0	M[0..4]	i	j	A_1	A_2	R_1	R_2
0	1,2,3,4,5	2	4	2	4	3	-

Small-step operational semantics

Intuition: Instructions modify state (registers, memory)

Represent program as control flow graph (CFG)



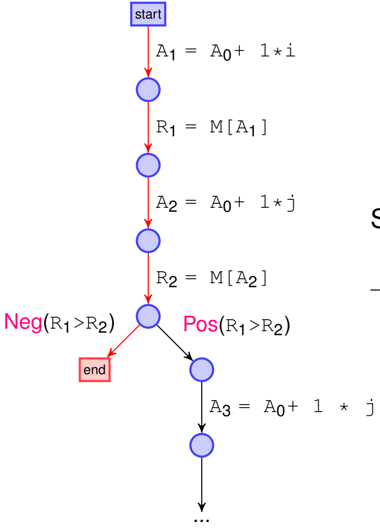
State:

A_0	M[0..4]	i	j	A_1	A_2	R_1	R_2
0	1,2,3,4,5	2	4	2	4	3	5

Small-step operational semantics

Intuition: Instructions modify state (registers, memory)

Represent program as control flow graph (CFG)



State:

A_0	M[0..4]	i	j	A_1	A_2	R_1	R_2
0	1,2,3,4,5	2	4	2	4	3	5

Formally (I)

Definition (Registers and Expressions)

Reg is an infinite set of register names. Expr is the set of expressions over these registers, constants and a standard set of operations.

Note: We do not formally define the set of operations here

Formally (I)

Definition (Registers and Expressions)

Reg is an infinite set of register names. Expr is the set of expressions over these registers, constants and a standard set of operations.

Note: We do not formally define the set of operations here

Definition (Action)

$\mathrm{Act} = \mathrm{Nop} \mid \mathrm{Pos}(e) \mid \mathrm{Neg}(e) \mid R = e \mid R = M[e] \mid M[e_1] = e_2$
where $e, e_1, e_2 \in \mathrm{Expr}$ are expressions and $R \in \mathrm{Reg}$ is a register.

Formally (I)

Definition (Registers and Expressions)

Reg is an infinite set of register names. Expr is the set of expressions over these registers, constants and a standard set of operations.

Note: We do not formally define the set of operations here

Definition (Action)

$\mathrm{Act} = \mathrm{Nop} \mid \mathrm{Pos}(e) \mid \mathrm{Neg}(e) \mid R = e \mid R = M[e] \mid M[e_1] = e_2$
where $e, e_1, e_2 \in \mathrm{Expr}$ are expressions and $R \in \mathrm{Reg}$ is a register.

Definition (Control Flow Graph)

An edge-labeled graph $G = (V, E, v_0, V_{\mathrm{end}})$ where $E \subseteq V \times \mathrm{Act} \times V$, $v_0 \in V$, $V_{\mathrm{end}} \subseteq V$ is called control flow graph (CFG).

Formally (I)

Definition (Registers and Expressions)

Reg is an infinite set of register names. Expr is the set of expressions over these registers, constants and a standard set of operations.

Note: We do not formally define the set of operations here

Definition (Action)

$\mathrm{Act} = \mathrm{Nop} \mid \mathrm{Pos}(e) \mid \mathrm{Neg}(e) \mid R = e \mid R = M[e] \mid M[e_1] = e_2$
where $e, e_1, e_2 \in \mathrm{Expr}$ are expressions and $R \in \mathrm{Reg}$ is a register.

Definition (Control Flow Graph)

An edge-labeled graph $G = (V, E, v_0, V_{\mathrm{end}})$ where $E \subseteq V \times \mathrm{Act} \times V$, $v_0 \in V$, $V_{\mathrm{end}} \subseteq V$ is called control flow graph (CFG).

Definition (State)

A state $s \in \mathrm{State}$ is represented by a pair $s = (\rho, \mu)$, where

ρ : $\mathrm{Reg} \to \mathrm{int}$ is the content of registers

μ : $\mathrm{int} \to \mathrm{int}$ is the content of memory

Formally (II)

Definition (Value of expression)

$[\![e]\!] \rho : \mathrm{int}$ is the value of expression e under register content ρ.

Formally (II)

Definition (Value of expression)

$[\![e]\!]\rho : \mathrm{int}$ is the value of expression e under register content ρ.

Definition (Effect of action)

The effect $[\![a]\!]$ of an action is a partial function on states:

Formally (II)

Definition (Value of expression)

$[\![e]\!]\rho : \mathrm{int}$ is the value of expression *e* under register content ρ.

Definition (Effect of action)

The effect $[\![a]\!]$ of an action is a partial function on states:

$$[\![\mathrm{Nop}]\!](\rho, \mu) := (\rho, \mu)$$

Formally (II)

Definition (Value of expression)

$[\![e]\!]\rho : \mathrm{int}$ is the value of expression e under register content ρ.

Definition (Effect of action)

The effect $[\![a]\!]$ of an action is a partial function on states:

$$[\![\mathrm{Nop}]\!](\rho, \mu) := (\rho, \mu)$$

$$[\![\mathrm{Pos}(e)]\!](\rho, \mu) := \begin{cases} (\rho, \mu) & \text{if } [\![e]\!]\rho \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Formally (II)

Definition (Value of expression)

$[\![e]\!]\rho : \mathrm{int}$ is the value of expression e under register content ρ.

Definition (Effect of action)

The effect $[\![a]\!]$ of an action is a partial function on states:

$$[\![\mathrm{Nop}]\!](\rho,\mu) := (\rho,\mu)$$

$$[\![\mathrm{Pos}(e)]\!](\rho,\mu) := \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\mathrm{Neg}(e)]\!](\rho,\mu) := \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Formally (II)

Definition (Value of expression)

$[\![e]\!]\rho : \mathrm{int}$ is the value of expression e under register content ρ.

Definition (Effect of action)

The effect $[\![a]\!]$ of an action is a partial function on states:

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho,\mu) &:= (\rho,\mu) \\
[\![\mathrm{Pos}(e)]\!](\rho,\mu) &:= \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![\mathrm{Neg}(e)]\!](\rho,\mu) &:= \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho = 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![R = e]\!](\rho,\mu) &:= (\rho(R \mapsto [\![e]\!]\rho),\mu)
\end{aligned}
$$

Formally (II)

Definition (Value of expression)

$[\![e]\!]\rho : \mathrm{int}$ is the value of expression e under register content ρ.

Definition (Effect of action)

The effect $[\![a]\!]$ of an action is a partial function on states:

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho,\mu) &:= (\rho,\mu) \\
[\![\mathrm{Pos}(e)]\!](\rho,\mu) &:= \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![\mathrm{Neg}(e)]\!](\rho,\mu) &:= \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho = 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![R = e]\!](\rho,\mu) &:= (\rho(R \mapsto [\![e]\!]\rho),\mu) \\
[\![R = M[e]]\!](\rho,\mu) &:= (\rho(R \mapsto \mu([\![e]\!]\rho)),\mu)
\end{aligned}
$$

Formally (II)

Definition (Value of expression)

$[\![e]\!]\rho : \mathrm{int}$ is the value of expression e under register content ρ.

Definition (Effect of action)

The effect $[\![a]\!]$ of an action is a partial function on states:

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho,\mu) &:= (\rho,\mu) \\
[\![\mathrm{Pos}(e)]\!](\rho,\mu) &:= \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![\mathrm{Neg}(e)]\!](\rho,\mu) &:= \begin{cases} (\rho,\mu) & \text{if } [\![e]\!]\rho = 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![R = e]\!](\rho,\mu) &:= (\rho(R \mapsto [\![e]\!]\rho),\mu) \\
[\![R = M[e]]\!](\rho,\mu) &:= (\rho(R \mapsto \mu([\![e]\!]\rho)),\mu) \\
[\![M[e_1] = e_2]\!](\rho,\mu) &:= (\rho,\mu([\![e_1]\!]\rho \mapsto [\![e_2]\!]\rho))
\end{aligned}
$$

Formally (III)

Given a CFG $G = (V, E, v_0, V_{\mathrm{end}})$

Definition (Path)

A sequence of adjacent edges $\pi = (v_1, a_1, v_2)(v_2, a_2, v_3) \ldots (v_n, a_n, v_{n+1}) \in E^*$ is called path from v_1 to v_{n+1}.

Notation $v_1 \xrightarrow{\pi} v_{n+1}$

Convention π is called path to v iff $v_0 \xrightarrow{\pi} v$

Special case $v \xrightarrow{\varepsilon} v$ for any $v \in V$

Formally (III)

Given a CFG $G = (V, E, v_0, V_{\text{end}})$

Definition (Path)

A sequence of adjacent edges $\pi = (v_1, a_1, v_2)(v_2, a_2, v_3) \ldots (v_n, a_n, v_{n+1}) \in E^*$ is called path from v_1 to v_{n+1}.

Notation $v_1 \xrightarrow{\pi} v_{n+1}$

Convention π is called path to v iff $v_0 \xrightarrow{\pi} v$

Special case $v \xrightarrow{\varepsilon} v$ for any $v \in V$

Definition (Effect of edge and path)

The effect of an edge $k = (u, a, v)$ is the effect of its action:

$$[\![(u, a, v)]\!] := [\![a]\!]$$

The effect of a path $\pi = k_1 \ldots k_n$ is the composition of the edge effects:

$$[\![k_1 \ldots k_n]\!] := [\![k_n]\!] \circ \ldots \circ [\![k_1]\!]$$

Formally (IV)

Definition (Computation)

A path π is called computation for state s, iff its effect is defined on s, i.e.,

$$s \in \mathrm{dom}(\llbracket\pi\rrbracket)$$

Then, the state $s' = \llbracket\pi\rrbracket s$ is called result of the computation.

Summary

- Action: $\mathrm{Act} = \mathrm{Nop} \mid \mathrm{Pos}(e) \mid \mathrm{Neg}(e) \mid R = e \mid R = M[e] \mid M[e_1] = e_2$
- CFG: $G = (V, E, v_0, V_{\mathrm{end}})$, $E \subseteq V \times \mathrm{Act} \times V$
- State: $s = (\rho, \mu)$, $\rho : \mathrm{Reg} \to \mathrm{int}$ (registers), $\mu : \mathrm{int} \to \mathrm{int}$ (memory)
- Value of expression under ρ: $[\![e]\!]\rho : \mathrm{int}$
- Effect of action a: $[\![a]\!] : \mathrm{State} \to \mathrm{State}$ (partial)
- Path π: Sequence of adjacent edges
- Effect of edge $k = (u, a, v)$: $[\![k]\!] = [\![a]\!]$
- Effect of path $\pi = k_1 \dots k_n$: $[\![\pi]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$
- π is computation for s: $s \in \mathrm{dom}([\![\pi]\!])$
- Result of computation π for s: $[\![\pi]\!]s$

Table of Contents

Memorization

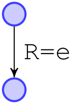
First, let's memorize every expression

- Register T_e memorizes value of expression e.
- Assumption: T_e not used in original program.

Memorization

First, let's memorize every expression

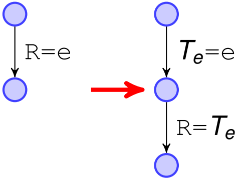
- Register T_e memorizes value of expression e.
- Assumption: T_e not used in original program.



Memorization

First, let's memorize every expression

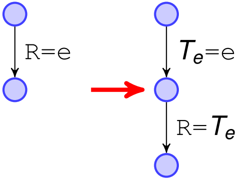
- Register T_e memorizes value of expression e.
- Assumption: T_e not used in original program.

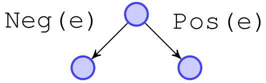


Memorization

First, let's memorize every expression

- Register T_e memorizes value of expression e.
- Assumption: T_e not used in original program.

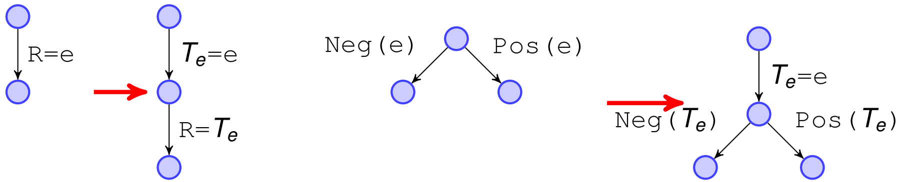




Memorization

First, let's memorize every expression

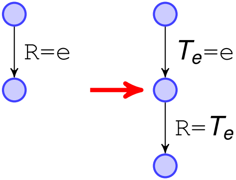
- Register T_e memorizes value of expression e.
- Assumption: T_e not used in original program.

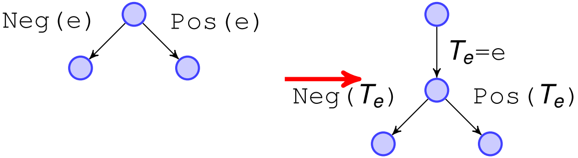


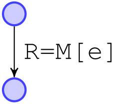
Memorization

First, let's memorize every expression

- Register T_e memorizes value of expression *e*.
- Assumption: T_e not used in original program.



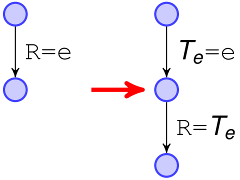


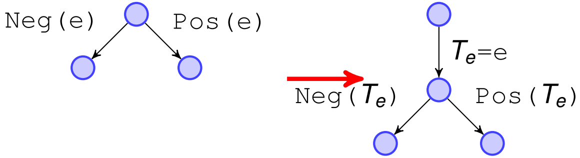


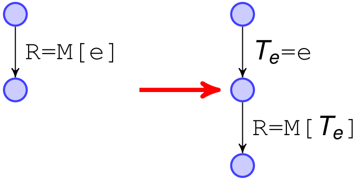
Memorization

First, let's memorize every expression

- Register T_e memorizes value of expression e.
- Assumption: T_e not used in original program.



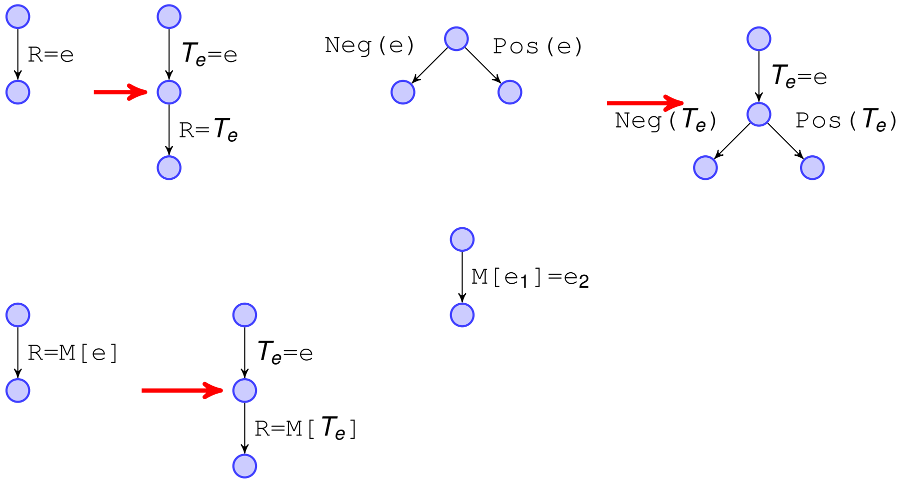




Memorization

First, let's memorize every expression

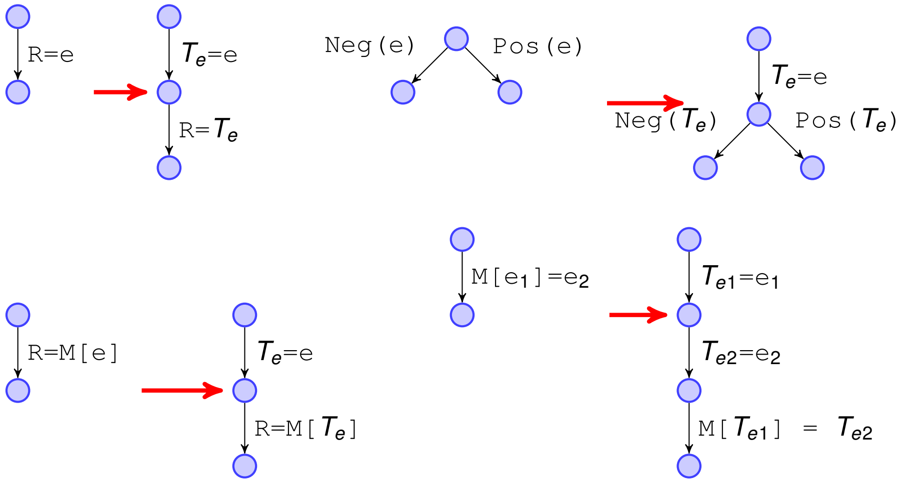
- Register T_e memorizes value of expression e.
- Assumption: T_e not used in original program.



Memorization

First, let's memorize every expression

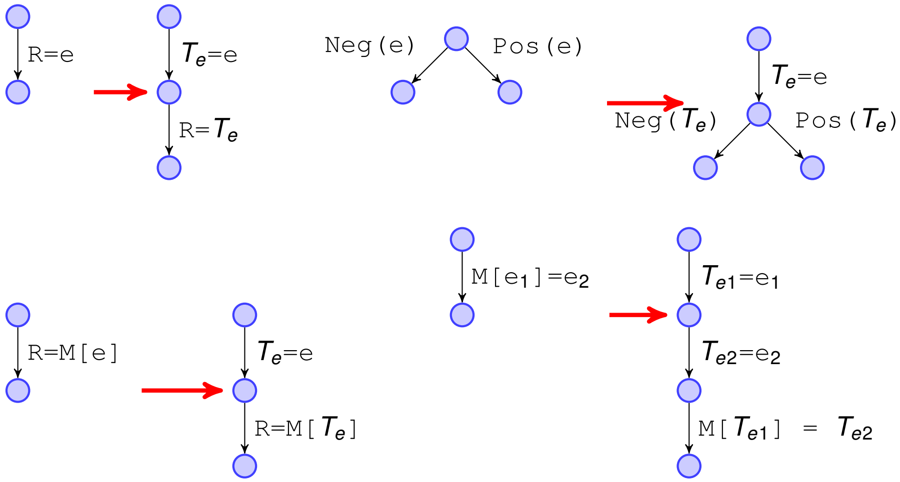
- Register T_e memorizes value of expression *e*.
- Assumption: T_e not used in original program.



Memorization

First, let's memorize every expression

- Register T_e memorizes value of expression *e*.
- Assumption: T_e not used in original program.



- Transformation obviously correct

Last Lecture (Oct 20)

- Simple intermediate language (IL)
  - Registers, memory, cond/ucond branching
  - Compiler: Input $\rightarrow$ Intermediate Language $\rightarrow$ Machine Code
  - Suitable for analysis/optimization

Last Lecture (Oct 20)

- Simple intermediate language (IL)
  - Registers, memory, cond/ucond branching
  - Compiler: Input $\to$ Intermediate Language $\to$ Machine Code
  - Suitable for analysis/optimization
- Control flow graphs, small-step operational semantics
  - Representation for programs in IL
  - Graphs labeled with actions
    - Nop,Pos/Neg,Assign,Load,Store
  - State = Register content, memory content
  - Actions are partial transformation on states
    - undefined - Test failed

Last Lecture (Oct 20)

- Simple intermediate language (IL)
  - Registers, memory, cond/ucond branching
  - Compiler: Input $\rightarrow$ Intermediate Language $\rightarrow$ Machine Code
  - Suitable for analysis/optimization
- Control flow graphs, small-step operational semantics
  - Representation for programs in IL
  - Graphs labeled with actions
    - Nop,Pos/Neg,Assign,Load,Store
  - State = Register content, memory content
  - Actions are partial transformation on states
    - undefined - Test failed
- Memorization Transformation
  - Memorize evaluation of e in register T_e

Available Expressions (Semantically)

Definition (Available Expressions in state)

The set of semantically available expressions in state (ρ, μ) is defined as

$$\mathrm{Aexp}(\rho, \mu) := \{e \mid [\![e]\!]\rho = \rho(T_e)\}$$

Intuition Register T_e contains correct value of e.

Available Expressions (Semantically)

Definition (Available Expressions in state)

The set of semantically available expressions in state (ρ, μ) is defined as

$$\mathrm{Aexp}(\rho, \mu) := \{e \mid [\![e]\!]\rho = \rho(T_e)\}$$

Intuition: Register T_e contains correct value of e.

Border case: All expressions available in undefined state

$$\mathrm{Aexp}(\text{undefined}) := \mathrm{Expr}$$

(See next slide why this makes sense)

Available Expressions (Semantically)

Definition (Available Expression at program point)

The set $\mathrm{Aexp}(u)$ of semantically available expressions at program point u is the set of expressions that are available in all states that may occur when the program is at u.

$$\mathrm{Aexp}(u) := \bigcap\{\mathrm{Aexp}([\![\pi]\!]s) \mid \pi, s.\ v_0 \xrightarrow{\pi} u\}$$

Available Expressions (Semantically)

Definition (Available Expression at program point)

The set $\mathrm{Aexp}(u)$ of semantically available expressions at program point u is the set of expressions that are available in all states that may occur when the program is at u.

$$\mathrm{Aexp}(u) := \bigcap\{\mathrm{Aexp}([\![\pi]\!]s) \mid \pi, s.\ v_0 \xrightarrow{\pi} u\}$$

Note Actual start state unknown, so all start states s are considered.

Available Expressions (Semantically)

Definition (Available Expression at program point)

The set $\mathrm{Aexp}(u)$ of semantically available expressions at program point u is the set of expressions that are available in all states that may occur when the program is at u.

$$\mathrm{Aexp}(u) := \bigcap\{\mathrm{Aexp}([\![\pi]\!]s) \mid \pi, s.\ v_0 \xrightarrow{\pi} u\}$$

Note Actual start state unknown, so all start states s are considered.

Note Above definition is smoother due to $\mathrm{Aexp}(\text{undefined}) := \mathrm{Expr}$

Simple Redundancy Elimination

Transformation Replace edge $(u, T_e = e, v)$ by (u, Nop, v) if e semantically available at u.

Simple Redundancy Elimination

Transformation Replace edge $(u, T_e = e, v)$ by (u, Nop, v) if e semantically available at u.

Correctness
- Whenever program reaches u with state (ρ, μ), we have $[\![e]\!]\rho = \rho(T_e)$ (That's exactly how semantically available is defined)
- Hence, $[\![T_e = e]\!](\rho, \mu) = (\rho, \mu) = [\![Nop]\!](\rho, \mu)$

Simple Redundancy Elimination

Transformation Replace edge $(u, T_e = e, v)$ by (u, Nop, v) if e semantically available at u.

Correctness
- Whenever program reaches u with state (ρ, μ), we have $[\![e]\!]\rho = \rho(T_e)$ (That's exactly how semantically available is defined)
- Hence, $[\![T_e = e]\!](\rho, \mu) = (\rho, \mu) = [\![Nop]\!](\rho, \mu)$

Remaining Problem How to compute available expressions

Simple Redundancy Elimination

Transformation Replace edge $(u, T_e = e, v)$ by (u, Nop, v) if e semantically available at u.

Correctness
- Whenever program reaches u with state (ρ, μ), we have $[\![e]\!]\rho = \rho(T_e)$ (That's exactly how semantically available is defined)
- Hence, $[\![T_e = e]\!](\rho, \mu) = (\rho, \mu) = [\![Nop]\!](\rho, \mu)$

Remaining Problem How to compute available expressions

Precisely No chance (Rice's Theorem)

Simple Redundancy Elimination

Transformation: Replace edge $(u, T_e = e, v)$ by (u, Nop, v) if e semantically available at u.

Correctness:
- Whenever program reaches u with state (ρ, μ), we have $[\![e]\!]\rho = \rho(T_e)$ (That's exactly how semantically available is defined)
- Hence, $[\![T_e = e]\!](\rho, \mu) = (\rho, \mu) = [\![Nop]\!](\rho, \mu)$

Remaining Problem: How to compute available expressions

Precisely: No chance (Rice's Theorem)

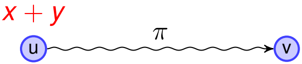
Observation: Enough to compute subset of semantically available expressions

- Transformation still correct

Available Expressions (Syntactically)

Idea Expression e (syntactically) available after computation π

- if e has been evaluated, and no register of e has been assigned afterwards

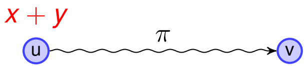


π does not contain assignment to x nor y

Available Expressions (Syntactically)

Idea Expression e (syntactically) available after computation π

- if e has been evaluated, and no register of e has been assigned afterwards



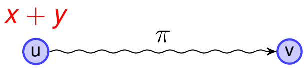
π does not contain assignment to x nor y

Purely syntactic criterion

Available Expressions (Syntactically)

Idea Expression *e* (syntactically) available after computation π

- if *e* has been evaluated, and no register of *e* has been assigned afterwards



π *does not contain assignment to x nor y*

Purely syntactic criterion

Can be computed incrementally for every edge

Available Expressions (Computation)

Let A be a set of available expressions.

Recall: Available $\Longleftarrow$ Already evaluated and no reg. assigned afterwards

Available Expressions (Computation)

Let A be a set of available expressions.

Recall: Available $\Longleftarrow$ Already evaluated and no reg. assigned afterwards

An action a transforms this into the set $[\![a]\!]^{\#} A$ of expressions available after a has been executed

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#} A &:= A \\
[\![\mathrm{Pos}(e)]\!]^{\#} A &:= A \\
[\![\mathrm{Neg}(e)]\!]^{\#} A &:= A \\
[\![T_e = e]\!]^{\#} A &:= A \cup \{e\} \\
[\![R = T_e]\!]^{\#} A &:= A \setminus \mathrm{Expr}_R \qquad \mathrm{Expr}_R := \text{expressions containing } R \\
[\![R = M[e]]\!]^{\#} A &:= A \setminus \mathrm{Expr}_R \\
[\![M[e_1] = e_2]\!]^{\#} A &:= A
\end{aligned}
$$

Available Expressions (Computation)

$[\![a]\!]^{\#}$ is called abstract effect of action a

Available Expressions (Computation)

$[\![a]\!]^\#$ is called abstract effect of action a

Again, the effect of an edge is the effect of its action

$$[\![(u, a, v)]\!]^\# = [\![a]\!]^\#$$

and the effect of a path $\pi = k_1 \dots k_n$ is

$$[\![\pi]\!]^\# := [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$$

Available Expressions (Computation)

$[\![a]\!]^\#$ is called abstract effect of action a

Again, the effect of an edge is the effect of its action

$$[\![(u, a, v)]\!]^\# = [\![a]\!]^\#$$

and the effect of a path $\pi = k_1 \ldots k_n$ is

$$[\![\pi]\!]^\# := [\![k_n]\!]^\# \circ \ldots \circ [\![k_1]\!]^\#$$

Definition (Available at v)

The set $A[v]$ of (syntactically) available expressions at v is

$$A[v] := \bigcap \{[\![\pi]\!]^\# \emptyset \mid \pi.\ v_0 \xrightarrow{\pi} v\}$$

Available Expressions (Correctness)

Idea Abstract effect corresponds to concrete effect

Lemma

$A \subseteq \mathit{Aexp}(s) \implies [\![a]\!]^{\#} A \subseteq \mathit{Aexp}([\![a]\!] s)$

Proof Check for every type of action.

Available Expressions (Correctness)

Idea Abstract effect corresponds to concrete effect

Lemma

$$A \subseteq Aexp(s) \implies [\![a]\!]^{\#} A \subseteq Aexp([\![a]\!] s)$$

Proof Check for every type of action.

This generalizes to paths

$$A \subseteq \mathrm{Aexp}(s) \implies [\![\pi]\!]^{\#} A \subseteq \mathrm{Aexp}([\![\pi]\!] s)$$

Available Expressions (Correctness)

Idea Abstract effect corresponds to concrete effect

Lemma

$A \subseteq Aexp(s) \implies [\![a]\!]^\# A \subseteq Aexp([\![a]\!]s)$

Proof Check for every type of action.

This generalizes to paths

$$A \subseteq \mathrm{Aexp}(s) \implies [\![\pi]\!]^\# A \subseteq \mathrm{Aexp}([\![\pi]\!]s)$$

And to program points

$$A[u] \subseteq \mathrm{Aexp}(u)$$

Recall:

$$\mathrm{Aexp}(u) = \bigcap\{\mathrm{Aexp}([\![\pi]\!]s) \mid \pi, s.\ v_0 \xrightarrow{\pi} u\}$$
$$A[u] = \bigcap\{[\![\pi]\!]^\# \emptyset \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

Summary

1. Transform program to memorize everything
   - Introduce registers T_e
2. Compute $A[u]$ for every program point u
   - $A[u] = \bigcap\{[\![\pi]\!]^\# \emptyset \mid \pi.\ v_0 \xrightarrow{\pi} u\}$
3. Replace redundant computations by `Nop`
   - $(u, T_e = e, v) \mapsto (u, \mathrm{Nop}, v)$ if $e \in A[u]$

Summary

1. Transform program to memorize everything
   - Introduce registers T_e
2. Compute $A[u]$ for every program point u
   - $A[u] = \bigcap\{[\![\pi]\!]^{\#}\emptyset \mid \pi.\ v_0 \xrightarrow{\pi} u\}$
3. Replace redundant computations by `Nop`
   - $(u, T_e = e, v) \mapsto (u, \mathrm{Nop}, v)$ if $e \in A[u]$

Warning Memorization transformation for $R = e$ should only be applied if

- $R \notin \mathrm{Reg}(e)$ (Otherwise, expression immediately unavailable)
- $e \notin \mathrm{Reg}$ (Otherwise, only one more register introduced)
- Evaluation of e is nontrivial (Otherwise, re-evaluation cheaper than memorization)

Remaining Problem

How to compute $A[u] = \bigcap\{[\![\pi]\!]^{\#}\emptyset \mid v_0 \xrightarrow{\pi} u\}$

- There may be infinitely many paths to u

Remaining Problem

How to compute $A[u] = \bigcap\{[\![\pi]\!]^\# \emptyset \mid v_0 \xrightarrow{\pi} u\}$

- There may be infinitely many paths to u

Solution: Collect restrictions to $A[u]$ into a constraint system

$$A[v_0] \subseteq \emptyset$$

$$A[v] \subseteq [\![a]\!]^\#(A[u]) \qquad \text{for edge } (u, a, v)$$

Remaining Problem

How to compute $A[u] = \bigcap\{\llbracket\pi\rrbracket^{\#}\emptyset \mid v_0 \xrightarrow{\pi} u\}$

- There may be infinitely many paths to u

Solution: Collect restrictions to $A[u]$ into a constraint system

$$A[v_0] \subseteq \emptyset$$

$$A[v] \subseteq \llbracket a\rrbracket^{\#}(A[u]) \qquad \text{for edge } (u, a, v)$$

Intuition

Nothing available at start node
For edge (u, a, v): At v, at most those expressions are available that would be available if we come from u.

Example

Let's regard a slightly modified available expression analysis

- Available expressions before memorization transformation has been applied
- Yields smaller examples, but more complicated proofs :)

Example

Let's regard a slightly modified available expression analysis

- Available expressions before memorization transformation has been applied
- Yields smaller examples, but more complicated proofs :)

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#} A &:= A \\
[\![\mathrm{Pos}(e)]\!]^{\#} A &:= A \cup \{e\} \\
[\![\mathrm{Neg}(e)]\!]^{\#} A &:= A \cup \{e\} \\
[\![R = e]\!]^{\#} A &:= (A \cup \{e\}) \setminus \mathrm{Expr}_R \\
[\![R = M[e]]\!]^{\#} A &:= (A \cup \{e\}) \setminus \mathrm{Expr}_R \\
[\![M[e_1] = e_2]\!]^{\#} A &:= A \cup \{e_1, e_2\}
\end{aligned}
$$

Example

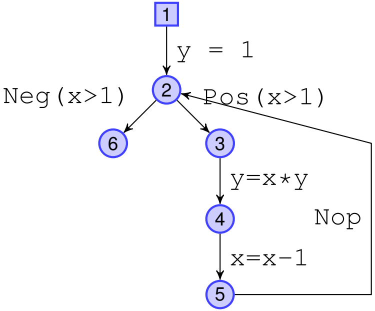
Let's regard a slightly modified available expression analysis

- Available expressions before memorization transformation has been applied
- Yields smaller examples, but more complicated proofs :)

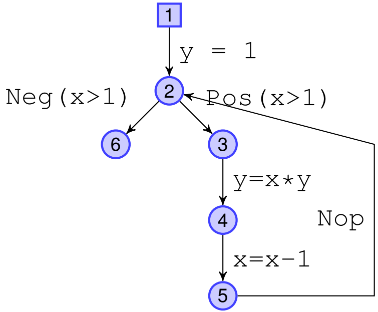
$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^\# A &:= A \\
[\![\mathrm{Pos}(e)]\!]^\# A &:= A \cup \{e\} \\
[\![\mathrm{Neg}(e)]\!]^\# A &:= A \cup \{e\} \\
[\![R = e]\!]^\# A &:= (A \cup \{e\}) \setminus \mathrm{Expr}_R \\
[\![R = M[e]]\!]^\# A &:= (A \cup \{e\}) \setminus \mathrm{Expr}_R \\
[\![M[e_1] = e_2]\!]^\# A &:= A \cup \{e_1, e_2\}
\end{aligned}
$$

Effect of transformation already included in constraint system

Example

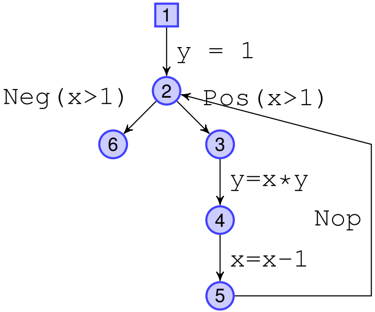


Example



$A[1] \subseteq \emptyset$

Example

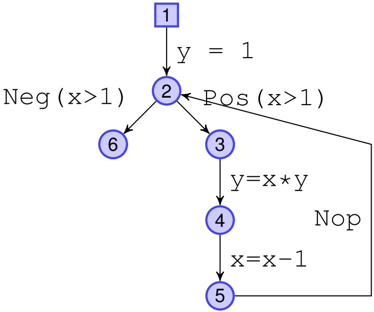


$$A[1] \subseteq \emptyset$$

$$A[2] \subseteq A[1] \cup \{1\} \setminus \text{Expr}_y$$

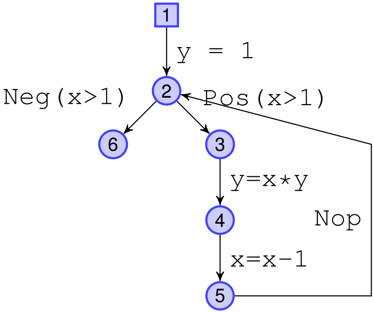
$$A[2] \subseteq A[5]$$

Example



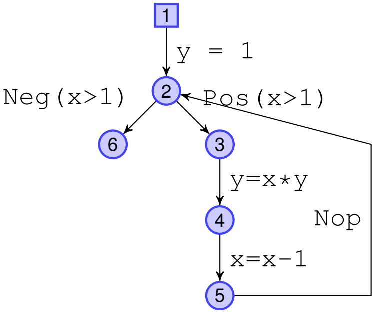
$$A[1] \subseteq \emptyset$$
$$A[2] \subseteq A[1] \cup \{1\} \setminus \mathrm{Expr}_y$$
$$A[2] \subseteq A[5]$$
$$A[3] \subseteq A[2] \cup \{x > 1\}$$

Example



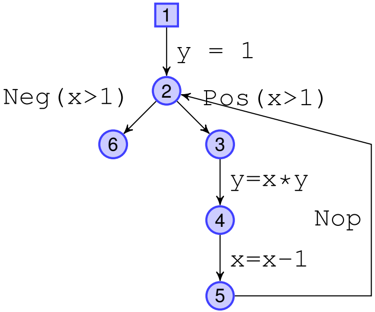
$$A[1] \subseteq \emptyset$$
$$A[2] \subseteq A[1] \cup \{1\} \setminus \mathrm{Expr}_y$$
$$A[2] \subseteq A[5]$$
$$A[3] \subseteq A[2] \cup \{x > 1\}$$
$$A[4] \subseteq A[3] \cup \{x * y\} \setminus \mathrm{Expr}_y$$

Example



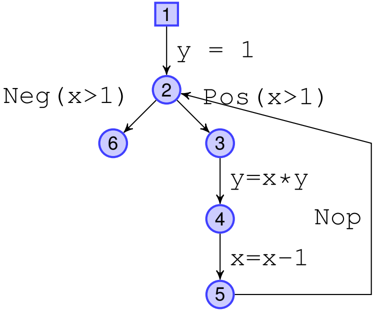
$$A[1] \subseteq \emptyset$$
$$A[2] \subseteq A[1] \cup \{1\} \setminus \mathrm{Expr}_y$$
$$A[2] \subseteq A[5]$$
$$A[3] \subseteq A[2] \cup \{x > 1\}$$
$$A[4] \subseteq A[3] \cup \{x * y\} \setminus \mathrm{Expr}_y$$
$$A[5] \subseteq A[4] \cup \{x - 1\} \setminus \mathrm{Expr}_x$$

Example



$$A[1] \subseteq \emptyset$$
$$A[2] \subseteq A[1] \cup \{1\} \setminus \mathrm{Expr}_y$$
$$A[2] \subseteq A[5]$$
$$A[3] \subseteq A[2] \cup \{x > 1\}$$
$$A[4] \subseteq A[3] \cup \{x * y\} \setminus \mathrm{Expr}_y$$
$$A[5] \subseteq A[4] \cup \{x - 1\} \setminus \mathrm{Expr}_x$$
$$A[6] \subseteq A[2] \cup \{x > 1\}$$

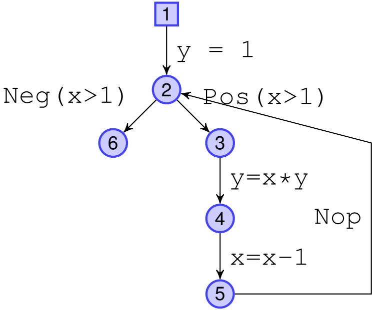
Example



Solution:

$$A[1] = \emptyset$$
$$A[2] = \{1\}$$
$$A[3] = \{1, x > 1\}$$
$$A[4] = \{1, x > 1\}$$
$$A[5] = \{1\}$$
$$A[6] = \{1, x > 1\}$$

Example



Also a solution:

$$A[1] = \emptyset$$
$$A[2] = \emptyset$$
$$A[3] = \emptyset$$
$$A[4] = \emptyset$$
$$A[5] = \emptyset$$
$$A[6] = \emptyset$$

Wanted

- Maximally large solution
  - Intuitively: Most precise information

Wanted

- Maximally large solution
  - Intuitively: Most precise information
- An algorithm to compute this solution

Naive Fixpoint Iteration (Sketch)

1. Initialize every $A[u] = \mathrm{Expr}$
   - Expressions actually occurring in program!
2. Evaluate RHSs
3. Update LHSs by intersecting with values of RHSs
4. Repeat (goto 2) until values of $A[u]$ stabilize

Naive Fixpoint Iteration (Example)

- On whiteboard!

Naive Fixpoint Iteration (Correctness)

Why does the algorithm terminate?

Naive Fixpoint Iteration (Correctness)

Why does the algorithm terminate?

- In each step, sets get smaller
- This can happen at most $|\mathrm{Expr}|$ times.

Naive Fixpoint Iteration (Correctness)

Why does the algorithm terminate?

- In each step, sets get smaller
- This can happen at most $|\mathrm{Expr}|$ times.

Why does the algorithm compute a solution?

Naive Fixpoint Iteration (Correctness)

Why does the algorithm terminate?

- In each step, sets get smaller
- This can happen at most $|\mathrm{Expr}|$ times.

Why does the algorithm compute a solution?

- If not arrived at solution yet, violated constraint will cause decrease of LHS

Naive Fixpoint Iteration (Correctness)

Why does the algorithm terminate?

- In each step, sets get smaller
- This can happen at most $|\mathrm{Expr}|$ times.

Why does the algorithm compute a solution?

- If not arrived at solution yet, violated constraint will cause decrease of LHS

Why does it compute the maximal solution?

Naive Fixpoint Iteration (Correctness)

Why does the algorithm terminate?

- In each step, sets get smaller
- This can happen at most $|\mathrm{Expr}|$ times.

Why does the algorithm compute a solution?

- If not arrived at solution yet, violated constraint will cause decrease of LHS

Why does it compute the maximal solution?

- Fixed-point theory. (Comes next)

Table of Contents

Partial Orders

Definition (Partial Order)

A partial order $(\mathbb{D}, \sqsubseteq)$ is a relation $\sqsubseteq$ on $\mathbb{D}$ that is reflexive, antisymmetric, and transitive, i.e., for all $a, b, c \in \mathbb{D}$:

$$\begin{aligned} a &\sqsubseteq a && \text{(reflexive)} \\ a \sqsubseteq b \wedge b \sqsubseteq a \implies a &= b && \text{(antisymmetric)} \\ a \sqsubseteq b \sqsubseteq c \implies a &\sqsubseteq c && \text{(transitive)} \end{aligned}$$

Partial Orders

Definition (Partial Order)

A partial order $(\mathbb{D}, \sqsubseteq)$ is a relation $\sqsubseteq$ on $\mathbb{D}$ that is reflexive, antisymmetric, and transitive, i.e., for all $a, b, c \in \mathbb{D}$:

$$\begin{aligned} a &\sqsubseteq a && \text{(reflexive)} \\ a \sqsubseteq b \wedge b \sqsubseteq a \implies a &= b && \text{(antisymmetric)} \\ a \sqsubseteq b \sqsubseteq c \implies a &\sqsubseteq c && \text{(transitive)} \end{aligned}$$

Examples $\leq$ on $\mathbb{N}$, $\subseteq$. Also $\geq$, $\supseteq$

Partial Orders

Definition (Partial Order)

A partial order $(\mathbb{D}, \sqsubseteq)$ is a relation $\sqsubseteq$ on $\mathbb{D}$ that is reflexive, antisymmetric, and transitive, i.e., for all $a, b, c \in \mathbb{D}$:

$$\begin{aligned} a \sqsubseteq a & & \text{(reflexive)} \\ a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b & & \text{(antisymmetric)} \\ a \sqsubseteq b \sqsubseteq c \implies a \sqsubseteq c & & \text{(transitive)} \end{aligned}$$

Examples $\leq$ on $\mathbb{N}$, $\subseteq$. Also $\geq$, $\supseteq$

Lemma (Dual order)

We define $a \sqsupseteq b := b \sqsubseteq a$. Let $\sqsubseteq$ be a partial order on $\mathbb{D}$. Then $\sqsupseteq$ also is a partial order on $\mathbb{D}$.

More examples

$\mathbb{D} = 2^{\{a,b,c\}}$ with $\subseteq$

$$
\begin{tikzcd}[arrows=dash]
 & \{a,b,c\} \arrow[dl] \arrow[d] \arrow[dr] & \\
\{a,b\} \arrow[d] \arrow[dr] & \{a,c\} \arrow[dl] \arrow[dr] & \{b,c\} \arrow[dl] \arrow[d] \\
\{a\} \arrow[dr] & \{b\} \arrow[d] & \{c\} \arrow[dl] \\
 & \emptyset &
\end{tikzcd}
$$

More examples

$\mathbb{Z}$ with relation $=$

$$\cdots \qquad -2 \qquad -1 \qquad 0 \qquad 1 \qquad 2 \qquad \cdots$$

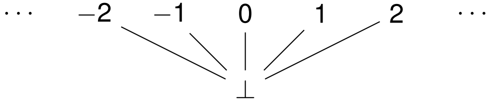
More examples

$\mathbb{Z}$ with relation $\leq$

$$
\begin{tikzcd}
\cdots \arrow[d, dash] \\
2 \arrow[d, dash] \\
1 \arrow[d, dash] \\
0 \arrow[d, dash] \\
-1 \arrow[d, dash] \\
-2 \arrow[d, dash] \\
\cdots
\end{tikzcd}
$$

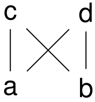
More examples

$\mathbb{Z}_\perp := \mathbb{Z} \cup \{\perp\}$ with relation $x \sqsubseteq y$ iff $x = \perp \vee x = y$



More examples

$\{a, b, c, d\}$ with $a \sqsubset c, a \sqsubset d, b \sqsubset c, b \sqsubset d$



Upper Bound

Definition (Upper bound)

$d \in \mathbb{D}$ is called upper bound of $X \subseteq \mathbb{D}$, iff

$$\forall x \in X.\ x \sqsubseteq d$$

Upper Bound

Definition (Upper bound)

$d \in \mathbb{D}$ is called upper bound of $X \subseteq \mathbb{D}$, iff

$$\forall x \in X.\ x \sqsubseteq d$$

Definition (Least Upper bound)

$d \in \mathbb{D}$ is called least upper bound of $X \subseteq \mathbb{D}$, iff

$$\begin{aligned} & d \text{ is upper bound of } X, \text{ and} \\ d \sqsubseteq y & \text{ for every upper bound } y \text{ of } X \end{aligned}$$

Upper Bound

Definition (Upper bound)

$d \in \mathbb{D}$ is called upper bound of $X \subseteq \mathbb{D}$, iff

$$\forall x \in X.\ x \sqsubseteq d$$

Definition (Least Upper bound)

$d \in \mathbb{D}$ is called least upper bound of $X \subseteq \mathbb{D}$, iff

$$\begin{array}{r} d \text{ is upper bound of } X \text{, and} \\ d \sqsubseteq y \text{ for every upper bound } y \text{ of } X \end{array}$$

Observation

Upper bound not always exists, e.g. $\{0, 2, 4, \ldots\} \subseteq \mathbb{Z}$

Upper Bound

Definition (Upper bound)

$d \in \mathbb{D}$ is called upper bound of $X \subseteq \mathbb{D}$, iff

$$\forall x \in X.\; x \sqsubseteq d$$

Definition (Least Upper bound)

$d \in \mathbb{D}$ is called least upper bound of $X \subseteq \mathbb{D}$, iff

$$\begin{aligned} &d \text{ is upper bound of } X\text{, and}\\ d \sqsubseteq y &\text{ for every upper bound } y \text{ of } X \end{aligned}$$

Observation

Upper bound not always exists, e.g. $\{0, 2, 4, \ldots\} \subseteq \mathbb{Z}$
Least upper bound not always exists, e.g. $\{a, b\} \subseteq \{a, b, c, d\}$ with $a \sqsubset c, a \sqsubset d, b \sqsubset c, b \sqsubset d$

Complete Lattice

Definition (Complete Lattice)

A complete lattice $(\mathbb{D}, \sqsubseteq)$ is a partial order where every subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$.

Complete Lattice

Definition (Complete Lattice)

A complete lattice $(\mathbb{D}, \sqsubseteq)$ is a partial order where every subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$.

Note Every complete lattice has

- A least element $\bot := \bigsqcup \emptyset \in \mathbb{D}$
- A greatest element $\top := \bigsqcup \mathbb{D} \in \mathbb{D}$

Complete Lattice

Definition (Complete Lattice)

A complete lattice $(\mathbb{D}, \sqsubseteq)$ is a partial order where every subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$.

Note Every complete lattice has

- A least element $\bot := \bigsqcup \emptyset \in \mathbb{D}$
- A greatest element $\top := \bigsqcup \mathbb{D} \in \mathbb{D}$

Moreover $a \sqcup b := \bigsqcup\{a, b\}$ and $a \sqcap b := \bigsqcap\{a, b\}$

Examples

- $(2^{\{a,b,c\}}, \subseteq)$ is complete lattice

Examples

- $(2^{\{a,b,c\}}, \subseteq)$ is complete lattice
- $(\mathbb{Z}, =)$ is not. Nor is $(\mathbb{Z}, \leq)$

Examples

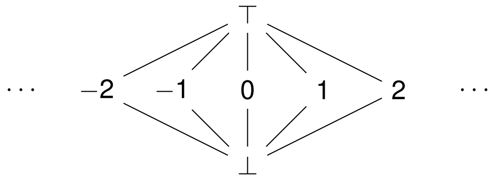
- $(2^{\{a,b,c\}}, \subseteq)$ is complete lattice
- $(\mathbb{Z}, =)$ is not. Nor is $(\mathbb{Z}, \leq)$
- $(\mathbb{Z}_\perp, \sqsubseteq)$ is also no complete lattice

Examples

- $(2^{\{a,b,c\}}, \subseteq)$ is complete lattice
- $(\mathbb{Z}, =)$ is not. Nor is $(\mathbb{Z}, \leq)$
- $(\mathbb{Z}_\perp, \sqsubseteq)$ is also no complete lattice
  - But we can define flat complete lattice

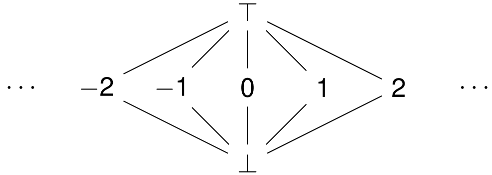
Flat complete lattice over $\mathbb{Z}$

$\mathbb{Z}_\perp^\top := \mathbb{Z} \cup \{\perp, \top\}$ with relation $x \sqsubseteq y$ iff $x = \perp \vee y = \top \vee x = y$



Flat complete lattice over $\mathbb{Z}$

$\mathbb{Z}_\perp^\top := \mathbb{Z} \cup \{\perp, \top\}$ with relation $x \sqsubseteq y$ iff $x = \perp \vee y = \top \vee x = y$



Note This construction works for every set, not only for $\mathbb{Z}$.

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

- Let $L = \{l \in \mathbb{D}.\ \forall x \in X.\ l \sqsubseteq x\}$
  - The set of all lower bounds of X

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

- Let $L = \{l \in \mathbb{D}.\ \forall x \in X.\ l \sqsubseteq x\}$
  - The set of all lower bounds of X
- Construct $\bigsqcap X := \bigsqcup L$

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

- Let $L = \{l \in \mathbb{D}.\ \forall x \in X.\ l \sqsubseteq x\}$
  - The set of all lower bounds of X
- Construct $\bigsqcap X := \bigsqcup L$
  - Show: $\bigsqcup L$ is lower bound

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

- Let $L = \{l \in \mathbb{D}.\ \forall x \in X.\ l \sqsubseteq x\}$
  - The set of all lower bounds of X
- Construct $\bigsqcap X := \bigsqcup L$
  - Show: $\bigsqcup L$ is lower bound
    - Assume $x \in X$.

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

- Let $L = \{l \in \mathbb{D}.\ \forall x \in X.\ l \sqsubseteq x\}$
  - The set of all lower bounds of X
- Construct $\bigsqcap X := \bigsqcup L$
  - Show: $\bigsqcup L$ is lower bound
    - Assume $x \in X$.
    - Then $\forall l \in L.\ l \sqsubseteq x$ (i.e., x is upper bound of L)

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

- Let $L = \{l \in \mathbb{D}.\ \forall x \in X.\ l \sqsubseteq x\}$
  - The set of all lower bounds of X
- Construct $\bigsqcap X := \bigsqcup L$
  - Show: $\bigsqcup L$ is lower bound
    - Assume $x \in X$.
    - Then $\forall l \in L.\ l \sqsubseteq x$ (i.e., x is upper bound of L)
    - Thus $\bigsqcup L \sqsubseteq x$ (b/c $\bigsqcup L$ is least upper bound)

Greatest Lower Bound

Theorem

Let $\mathbb{D}$ be a complete lattice. Then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

- Let $L = \{l \in \mathbb{D}.\ \forall x \in X.\ l \sqsubseteq x\}$
  - The set of all lower bounds of X
- Construct $\bigsqcap X := \bigsqcup L$
  - Show: $\bigsqcup L$ is lower bound
    - Assume $x \in X$.
    - Then $\forall l \in L.\ l \sqsubseteq x$ (i.e., x is upper bound of L)
    - Thus $\bigsqcup L \sqsubseteq x$ (b/c $\bigsqcup L$ is least upper bound)
  - Obvious: $\bigsqcup L$ is $\sqsupseteq$ than all lower bounds

Examples

- In $(2^{\{a,b,c\}}, \subseteq)$
  - Note, in lattices with $\subseteq$-ordering, we occasionally write $\bigcup$, $\bigcap$ instead of $\bigsqcup$, $\bigsqcap$
  - $\bigcup\{\{a,b\},\{a,c\}\} = \{a,b,c\}$, $\bigcap\{\{a,b\},\{a,c\}\} = \{a\}$

Examples

- In $(2^{\{a,b,c\}}, \subseteq)$
  - Note, in lattices with $\subseteq$-ordering, we occasionally write $\bigcup$, $\bigcap$ instead of $\bigsqcup$, $\bigsqcap$
  - $\bigcup\{\{a,b\},\{a,c\}\} = \{a,b,c\}$, $\bigcap\{\{a,b\},\{a,c\}\} = \{a\}$
- In $\mathbb{Z}_{-\infty}^{+\infty}$:
  - $\bigsqcup\{1,2,3,4\} = 4$, $\bigsqcap\{1,2,3,4\} = 1$
  - $\bigsqcup\{1,2,3,4,\ldots\} = +\infty$, $\bigsqcap\{1,2,3,4,\ldots\} = 1$

Last Lecture

- Syntactic criterion for available expressions
- Constraint system to express it
  - Yet to come: Link between CS and path-based criterion
- Naive fixpoint iteration to compute maximum solution of CS
- Partial orders, complete lattices

Monotonic function

Definition

Let $(\mathbb{D}_1, \sqsubseteq_1)$ and $(\mathbb{D}_2, \sqsubseteq_2)$ be partial orders. A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotonic, iff

$$\forall x, y \in \mathbb{D}_1 .\ x \sqsubseteq_1 y \implies f(x) \sqsubseteq_2 f(y)$$

Examples

- $f :: \mathbb{N} \to \mathbb{Z}$ with $f(x) := x - 10$

Examples

- $f :: \mathbb{N} \to \mathbb{Z}$ with $f(x) := x - 10$
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) := x + 10$

Examples

- $f :: \mathbb{N} \to \mathbb{Z}$ with $f(x) := x - 10$
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) := x + 10$
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) := (X \cup \{a, b\}) \setminus \{b, c\}$

Examples

- $f :: \mathbb{N} \to \mathbb{Z}$ with $f(x) := x - 10$
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) := x + 10$
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) := (X \cup \{a, b\}) \setminus \{b, c\}$
  - In general, functions of this form are monotonic wrt. $\subseteq$.

Examples

- $f :: \mathbb{N} \to \mathbb{Z}$ with $f(x) := x - 10$
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) := x + 10$
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) := (X \cup \{a, b\}) \setminus \{b, c\}$
  - In general, functions of this form are monotonic wrt. $\subseteq$.
- $f :: \mathbb{Z} \to \mathbb{Z}$ with $f(x) := -x$ (Not monotonic)

Examples

- $f :: \mathbb{N} \to \mathbb{Z}$ with $f(x) := x - 10$
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) := x + 10$
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) := (X \cup \{a, b\}) \setminus \{b, c\}$
  - In general, functions of this form are monotonic wrt. $\subseteq$.
- $f :: \mathbb{Z} \to \mathbb{Z}$ with $f(x) := -x$ (Not monotonic)
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) := \{x \mid x \notin X\}$ (Not monotonic)

Examples

- $f :: \mathbb{N} \to \mathbb{Z}$ with $f(x) := x - 10$
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) := x + 10$
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) := (X \cup \{a, b\}) \setminus \{b, c\}$
  - In general, functions of this form are monotonic wrt. $\subseteq$.
- $f :: \mathbb{Z} \to \mathbb{Z}$ with $f(x) := -x$ (Not monotonic)
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) := \{x \mid x \notin X\}$ (Not monotonic)
  - Functions involving negation/complement usually not monotonic.

Least fixed point

Definition

Let $f : \mathbb{D} \to \mathbb{D}$ be a function.
A value $d \in \mathbb{D}$ with $f(d) = d$ is called fixed point of f.

If $\mathbb{D}$ is a partial ordering, a fixed point $d_0 \in D$ with

$$\forall d.\ f(d) = d \implies d_0 \sqsubseteq d$$

is called least fixed point. If such a d_0 exists, it is uniquely determined, and we define

$$\mathrm{lfp}(f) := d_0$$

Examples

- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) = x + 1$ No fixed points

Examples

- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) = x + 1$ No fixed points
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) = x$. Every $x \in \mathbb{N}$ is fixed point.

Examples

- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) = x + 1$ No fixed points
- $f :: \mathbb{N} \to \mathbb{N}$ with $f(x) = x$. Every $x \in \mathbb{N}$ is fixed point.
- $f :: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ with $f(X) = X \cup \{a, b\}$. $\mathrm{lfp}(f) = \{a, b\}$.

Function composition

Theorem

If $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ *and* $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ *are monotonic, then also* $f_2 \circ f_1$ *is monotonic.*

Function composition

Theorem

If $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ and $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ are monotonic, then also $f_2 \circ f_1$ is monotonic.

Proof: $a \sqsubseteq b \implies f_1(a) \sqsubseteq f_1(b) \implies f_2(f_1(a)) \sqsubseteq f_2(f_1(b))$.

Function lattice

Definition

Let $(\mathbb{D}, \sqsubseteq)$ be a partial ordering. We overload $\sqsubseteq$ to functions from A to $\mathbb{D}$:

$$f \sqsubseteq g \text{ iff } \forall x.\ f(x) \sqsubseteq g(x)$$

$[A \to \mathbb{D}]$ is the set of functions from A to $\mathbb{D}$.

Function lattice

Definition

Let $(\mathbb{D}, \sqsubseteq)$ be a partial ordering. We overload $\sqsubseteq$ to functions from A to $\mathbb{D}$:

$$f \sqsubseteq g \text{ iff } \forall x.\ f(x) \sqsubseteq g(x)$$

$[A \to \mathbb{D}]$ is the set of functions from A to $\mathbb{D}$.

Theorem

If $(\mathbb{D}, \sqsubseteq)$ is a partial ordering/complete lattice, then also $([A \to \mathbb{D}], \sqsubseteq)$.
In particular, we have:

$$(\bigsqcup F)(x) = \bigsqcup\{f(x) \mid f \in F\}$$

Function lattice

Definition

Let $(\mathbb{D}, \sqsubseteq)$ be a partial ordering. We overload $\sqsubseteq$ to functions from A to $\mathbb{D}$:

$$f \sqsubseteq g \text{ iff } \forall x.\ f(x) \sqsubseteq g(x)$$

$[A \to \mathbb{D}]$ is the set of functions from A to $\mathbb{D}$.

Theorem

If $(\mathbb{D}, \sqsubseteq)$ is a partial ordering/complete lattice, then also $([A \to \mathbb{D}], \sqsubseteq)$.
In particular, we have:

$$(\bigsqcup F)(x) = \bigsqcup\{f(x) \mid f \in F\}$$

Proof: On whiteboard.

Component-wise ordering on tuples

- Tuples $\vec{x} \in \mathbb{D}^n$ can be seen as functions $\vec{x} : \{1, \ldots, n\} \to \mathbb{D}$

Component-wise ordering on tuples

- Tuples $\vec{x} \in \mathbb{D}^n$ can be seen as functions $\vec{x} : \{1, \dots, n\} \to \mathbb{D}$
- Yields component-wise ordering:

$$\vec{x} \sqsubseteq \vec{y} \text{ iff } \forall i : \{1, \dots, n\}.\ x_i \sqsubseteq y_i$$

Component-wise ordering on tuples

- Tuples $\vec{x} \in \mathbb{D}^n$ can be seen as functions $\vec{x} : \{1, \dots, n\} \to \mathbb{D}$
- Yields component-wise ordering:

$$\vec{x} \sqsubseteq \vec{y} \text{ iff } \forall i : \{1, \dots, n\}.\ x_i \sqsubseteq y_i$$

- $(\mathbb{D}^n, \sqsubseteq)$ is complete lattice if $(\mathbb{D}, \sqsubseteq)$ is complete lattice.

Application

- Idea: Encode constraint system as function. Solutions as fixed points.

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

$$x_i \sqsupseteq f_i(x_1, \dots, x_n)$$

where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n)$$

where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

- Observation: One constraint per x_i is enough.

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n)$$

where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

- Observation: One constraint per x_i is enough.
  - Assume we have $x_i \sqsupseteq rhs_1(x_1, \ldots, x_n)$, ..., $x_i \sqsupseteq rhs_m(x_1, \ldots, x_n)$

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

$$x_i \sqsupseteq f_i(x_1, \dots, x_n)$$

where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

- Observation: One constraint per x_i is enough.
  - Assume we have $x_i \sqsupseteq rhs_1(x_1, \dots, x_n)$, ..., $x_i \sqsupseteq rhs_m(x_1, \dots, x_n)$
  - Replace by $x_i \sqsupseteq (\bigsqcup\{rhs_j \mid 1 \le j \le m\})(x_1, \dots, x_n)$

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

  $$x_i \sqsupseteq f_i(x_1, \ldots, x_n)$$

  where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

- Observation: One constraint per x_i is enough.
  - Assume we have $x_i \sqsupseteq rhs_1(x_1, \ldots, x_n)$, ..., $x_i \sqsupseteq rhs_m(x_1, \ldots, x_n)$
  - Replace by $x_i \sqsupseteq (\bigsqcup\{rhs_j \mid 1 \leq j \leq m\})(x_1, \ldots, x_n)$
    - Does not change solutions.

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

$$x_i \sqsupseteq f_i(x_1, \dots, x_n)$$

where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

- Observation: One constraint per x_i is enough.
  - Assume we have $x_i \sqsupseteq rhs_1(x_1, \dots, x_n)$, ..., $x_i \sqsupseteq rhs_m(x_1, \dots, x_n)$
  - Replace by $x_i \sqsupseteq (\bigsqcup\{rhs_j \mid 1 \leq j \leq m\})(x_1, \dots, x_n)$
    - Does not change solutions.
- Define $F : \mathbb{D}^n \to \mathbb{D}^n$, with

$$F(x_1, \dots, x_n) := (f_1(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n))$$

Then, constraints expressed by $\vec{x} \sqsupseteq F(\vec{x})$.

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

$$x_i \sqsupseteq f_i(x_1, \dots, x_n)$$

  where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

- Observation: One constraint per x_i is enough.
  - Assume we have $x_i \sqsupseteq rhs_1(x_1, \dots, x_n)$, ..., $x_i \sqsupseteq rhs_m(x_1, \dots, x_n)$
  - Replace by $x_i \sqsupseteq (\bigsqcup\{rhs_j \mid 1 \leq j \leq m\})(x_1, \dots, x_n)$
    - Does not change solutions.
- Define $F : \mathbb{D}^n \to \mathbb{D}^n$, with

$$F(x_1, \dots, x_n) := (f_1(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n))$$

  Then, constraints expressed by $\vec{x} \sqsupseteq F(\vec{x})$.
  - Fixed-Points of F are solutions

Application

- Idea: Encode constraint system as function. Solutions as fixed points.
- Constraints have the form

$$x_i \sqsupseteq f_i(x_1, \dots, x_n)$$

where

x_i	variables	e.g., $A[u]$, for $u \in V$
$(\mathbb{D}, \sqsubseteq)$	complete lattice	e.g., $(2^{\mathrm{Expr}}, \supseteq)$
$f_i : \mathbb{D}^n \to \mathbb{D}$	RHS	e.g., $(A[u] \cup \{e\}) \setminus \mathrm{Expr}_R$

- Observation: One constraint per x_i is enough.
  - Assume we have $x_i \sqsupseteq rhs_1(x_1, \dots, x_n)$, ..., $x_i \sqsupseteq rhs_m(x_1, \dots, x_n)$
  - Replace by $x_i \sqsupseteq (\bigsqcup\{rhs_j \mid 1 \leq j \leq m\})(x_1, \dots, x_n)$
    - Does not change solutions.
- Define $F : \mathbb{D}^n \to \mathbb{D}^n$, with

$$F(x_1, \dots, x_n) := (f_1(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n))$$

Then, constraints expressed by $\vec{x} \sqsupseteq F(\vec{x})$.

- Fixed-Points of F are solutions
- Least solution = least fixed point (next!)

Least fixed points of monotonic functions

- Moreover, F is monotonic if the f_i are.
- Question: Does $\mathrm{lfp}(F)$ exist? Does fp-iteration compute it?

Knaster-Tarski fixed-point Theorem

Knaster-Tarski

Let $(\mathbb{D}, \sqsubseteq)$ *be a complete lattice, and* $f : \mathbb{D} \to \mathbb{D}$ *be a monotonic function. Then,* f *has a least and a greatest fixed point given by*

$$lfp(f) = \bigsqcap \{x \mid f(x) \sqsubseteq x\} \qquad gfp(f) = \bigsqcup \{x \mid x \sqsubseteq f(x)\}$$

Knaster-Tarski fixed-point Theorem

Knaster-Tarski

Let $(\mathbb{D}, \sqsubseteq)$ *be a complete lattice, and* $f : \mathbb{D} \to \mathbb{D}$ *be a monotonic function. Then,* f *has a least and a greatest fixed point given by*

$$lfp(f) = \bigsqcap \{x \mid f(x) \sqsubseteq x\} \qquad gfp(f) = \bigsqcup \{x \mid x \sqsubseteq f(x)\}$$

Proof Let $P = \{x \mid f(x) \sqsubseteq x\}$. ($P$ is set of *pre-fixpoints*)

Knaster-Tarski fixed-point Theorem

Knaster-Tarski

Let $(\mathbb{D}, \sqsubseteq)$ *be a complete lattice, and f* $: \mathbb{D} \to \mathbb{D}$ *be a monotonic function. Then, f has a least and a greatest fixed point given by*

$$lfp(f) = \bigsqcap\{x \mid f(x) \sqsubseteq x\} \qquad gfp(f) = \bigsqcup\{x \mid x \sqsubseteq f(x)\}$$

Proof Let $P = \{x \mid f(x) \sqsubseteq x\}$. ($P$ is set of *pre-fixpoints*)

- Show (1): $f(\bigsqcap P) \sqsubseteq \bigsqcap P$.
  - Have $\forall x \in P.\ f(\bigsqcap P) \sqsubseteq f(x) \sqsubseteq x$ (lower bound, mono, def.P)
  - I.e., $f(\bigsqcap P)$ is lower bound of P
  - Thus $f(\bigsqcap P) \sqsubseteq \bigsqcap P$ (greatest lower bound).

Knaster-Tarski fixed-point Theorem

Knaster-Tarski

Let $(\mathbb{D}, \sqsubseteq)$ *be a complete lattice, and* $f : \mathbb{D} \to \mathbb{D}$ *be a monotonic function. Then, f has a least and a greatest fixed point given by*

$$lfp(f) = \bigsqcap\{x \mid f(x) \sqsubseteq x\} \qquad\qquad gfp(f) = \bigsqcup\{x \mid x \sqsubseteq f(x)\}$$

Proof Let $P = \{x \mid f(x) \sqsubseteq x\}$. ($P$ is set of *pre-fixpoints*)

- Show (1): $f(\bigsqcap P) \sqsubseteq \bigsqcap P$.
  - Have $\forall x \in P.\ f(\bigsqcap P) \sqsubseteq f(x) \sqsubseteq x$ (lower bound, mono, def.P)
  - I.e., $f(\bigsqcap P)$ is lower bound of P
  - Thus $f(\bigsqcap P) \sqsubseteq \bigsqcap P$ (greatest lower bound).
- Show (2): $\bigsqcap P \sqsubseteq f(\bigsqcap P)$
  - From (1) have $f(f(\bigsqcap P)) \sqsubseteq f(\bigsqcap P)$ (mono)
  - Hence $f(\bigsqcap P) \in P$ (def.P)
  - Thus $\bigsqcap P \sqsubseteq f(\bigsqcap P)$ (lower bound).

Knaster-Tarski fixed-point Theorem

Knaster-Tarski

Let $(\mathbb{D}, \sqsubseteq)$ *be a complete lattice, and* $f : \mathbb{D} \to \mathbb{D}$ *be a monotonic function. Then, f has a least and a greatest fixed point given by*

$$lfp(f) = \bigsqcap \{x \mid f(x) \sqsubseteq x\} \qquad gfp(f) = \bigsqcup \{x \mid x \sqsubseteq f(x)\}$$

Proof Let $P = \{x \mid f(x) \sqsubseteq x\}$. ($P$ is set of *pre-fixpoints*)

- Show (1): $f(\sqcap P) \sqsubseteq \sqcap P$.
  - Have $\forall x \in P.\ f(\sqcap P) \sqsubseteq f(x) \sqsubseteq x$ (lower bound, mono, def.P)
  - I.e., $f(\sqcap P)$ is lower bound of P
  - Thus $f(\sqcap P) \sqsubseteq \sqcap P$ (greatest lower bound).
- Show (2): $\sqcap P \sqsubseteq f(\sqcap P)$
  - From (1) have $f(f(\sqcap P)) \sqsubseteq f(\sqcap P)$ (mono)
  - Hence $f(\sqcap P) \in P$ (def.P)
  - Thus $\sqcap P \sqsubseteq f(\sqcap P)$ (lower bound).
- Show (3): Least fixed point
  - Assume $d = f(d)$ is another fixed point
  - Hence $f(d) \sqsubseteq d$ (reflexive)
  - Hence $d \in P$ (def.P)
  - Thus $\sqcap P \sqsubseteq d$ (lower bound)

Knaster-Tarski fixed-point Theorem

Knaster-Tarski

Let $(\mathbb{D}, \sqsubseteq)$ be a complete lattice, and $f : \mathbb{D} \to \mathbb{D}$ be a monotonic function. Then, f has a least and a greatest fixed point given by

$$\mathit{lfp}(f) = \bigsqcap\{x \mid f(x) \sqsubseteq x\} \qquad \mathit{gfp}(f) = \bigsqcup\{x \mid x \sqsubseteq f(x)\}$$

Proof Let $P = \{x \mid f(x) \sqsubseteq x\}$. ($P$ is set of *pre-fixpoints*)

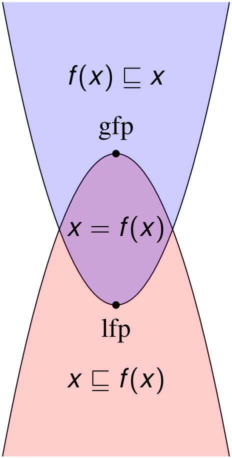
- Show (1): $f(\bigsqcap P) \sqsubseteq \bigsqcap P$.
  - Have $\forall x \in P.\ f(\bigsqcap P) \sqsubseteq f(x) \sqsubseteq x$ (lower bound, mono, def.P)
  - I.e., $f(\bigsqcap P)$ is lower bound of P
  - Thus $f(\bigsqcap P) \sqsubseteq \bigsqcap P$ (greatest lower bound).
- Show (2): $\bigsqcap P \sqsubseteq f(\bigsqcap P)$
  - From (1) have $f(f(\bigsqcap P)) \sqsubseteq f(\bigsqcap P)$ (mono)
  - Hence $f(\bigsqcap P) \in P$ (def.P)
  - Thus $\bigsqcap P \sqsubseteq f(\bigsqcap P)$ (lower bound).
- Show (3): Least fixed point
  - Assume $d = f(d)$ is another fixed point
  - Hence $f(d) \sqsubseteq d$ (reflexive)
  - Hence $d \in P$ (def.P)
  - Thus $\bigsqcap P \sqsubseteq d$ (lower bound)
- Greatest fixed point: Dually.

□

Used Facts

lower bound	$x \in X \implies \bigsqcap X \sqsubseteq x$
greatest lower bound	$(\forall x \in X.\, d \sqsubseteq X) \implies d \sqsubseteq \bigsqcap X$
mono	f monotonic: $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$
reflexive	$x \sqsubseteq x$

Knaster-Tarski Fixed-Point Theorem (Intuition)



pre-fixpoints

post-fixpoints

Least solution = lfp

Recall: Constraints where $\vec{x} \sqsupseteq F(\vec{x})$

Knaster-Tarski: $\mathrm{lfp}(F) = \bigsqcap\{\vec{x} \mid \vec{x} \sqsupseteq F(\vec{x})\}$

- I.e.: Least fixed point is lower bound of solutions

Kleene fixed-point theorem

Kleene fixed-point

Let $(\mathbb{D}, \sqsubseteq)$ *be a complete lattice, and* $f : \mathbb{D} \to \mathbb{D}$ *be a monotonic function. Then:*

$$\bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\} \sqsubseteq \mathit{lfp}(f)$$

If f *is distributive, we even have:*

$$\bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\} = \mathit{lfp}(f)$$

Kleene fixed-point theorem

Kleene fixed-point

Let $(\mathbb{D}, \sqsubseteq)$ *be a complete lattice, and* $f : \mathbb{D} \to \mathbb{D}$ *be a monotonic function. Then:*

$$\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\} \sqsubseteq \mathit{lfp}(f)$$

If f *is distributive, we even have:*

$$\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\} = \mathit{lfp}(f)$$

Definition

Distributivity A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ over complete lattices $(\mathbb{D}_1, \sqsubseteq_1)$ and $(\mathbb{D}_2, \sqsubseteq_2)$ is called distributive, iff

$$X \neq \emptyset \implies f(\bigsqcup_1 X) = \bigsqcup_2 \{f(x) \mid x \in X\}$$

Note: Distributivity implies monotonicity.

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.
  - $i = 0$: $f^0(\bot) = \bot \sqsubseteq \mathrm{lfp}(f)$ (def. f^0, bot least)

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.
  - $i = 0$: $f^0(\bot) = \bot \sqsubseteq \mathrm{lfp}(f)$ (def.f^0, bot least)
  - $i + 1$: IH: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$. To show: $f^{i+1}(\bot) \sqsubseteq \mathrm{lfp}(f)$

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.
  - $i = 0$: $f^0(\bot) = \bot \sqsubseteq \mathrm{lfp}(f)$ (def.f^0, bot least)
  - $i + 1$: IH: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$. To show: $f^{i+1}(\bot) \sqsubseteq \mathrm{lfp}(f)$
  - Have $f^{i+1}(\bot) = f(f^i(\bot))$ (def.f^{i+1})

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.
  - $i = 0$: $f^0(\bot) = \bot \sqsubseteq \mathrm{lfp}(f)$ (def.f^0, bot least)
  - $i + 1$: IH: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$. To show: $f^{i+1}(\bot) \sqsubseteq \mathrm{lfp}(f)$
  - Have $f^{i+1}(\bot) = f(f^i(\bot))$ (def.f^{i+1})
  - $\sqsubseteq f(\mathrm{lfp}(f))$ (IH, mono)

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.
  - $i = 0$: $f^0(\bot) = \bot \sqsubseteq \mathrm{lfp}(f)$ (def. f^0, bot least)
  - $i + 1$: IH: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$. To show: $f^{i+1}(\bot) \sqsubseteq \mathrm{lfp}(f)$
  - Have $f^{i+1}(\bot) = f(f^i(\bot))$ (def. f^{i+1})
  - $\sqsubseteq f(\mathrm{lfp}(f))$ (IH, mono)
  - $= \mathrm{lfp}(f)$ ($\mathrm{lfp}(f)$ is fixed point)

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.
  - $i = 0$: $f^0(\bot) = \bot \sqsubseteq \mathrm{lfp}(f)$ (def. f^0, bot least)
  - $i+1$: IH: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$. To show: $f^{i+1}(\bot) \sqsubseteq \mathrm{lfp}(f)$
  - Have $f^{i+1}(\bot) = f(f^i(\bot))$ (def. f^{i+1})
  - $\sqsubseteq f(\mathrm{lfp}(f))$ (IH, mono)
  - $= \mathrm{lfp}(f)$ ($\mathrm{lfp}(f)$ is fixed point)

I.e., $\mathrm{lfp}(f)$ is upper bound of $\{f^i(\bot) \mid i \in \mathbb{N}\}$

Kleene fixed-point theorem: Proof

By Knaster-Tarski theorem, $\mathrm{lfp}(f)$ exists.

Show that for all i: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$

- Induction on i.
  - $i = 0$: $f^0(\bot) = \bot \sqsubseteq \mathrm{lfp}(f)$ (def. f^0, bot least)
  - $i + 1$: IH: $f^i(\bot) \sqsubseteq \mathrm{lfp}(f)$. To show: $f^{i+1}(\bot) \sqsubseteq \mathrm{lfp}(f)$
  - Have $f^{i+1}(\bot) = f(f^i(\bot))$ (def. f^{i+1})
  - $\sqsubseteq f(\mathrm{lfp}(f))$ (IH, mono)
  - $= \mathrm{lfp}(f)$ ($\mathrm{lfp}(f)$ is fixed point)

I.e., $\mathrm{lfp}(f)$ is upper bound of $\{f^i(\bot) \mid i \in \mathbb{N}\}$

Thus, $\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\} \sqsubseteq \mathrm{lfp}(f)$ (least upper bound) $\square$

Kleene fixed-point theorem: Proof (ctd)

Assume f is distributive.

Kleene fixed-point theorem: Proof (ctd)

Assume f is distributive.
Hence $f(\bigsqcup\{f^i(\perp) \mid i \in \mathbb{N}\}) = \bigsqcup\{f^{i+1}(\perp) \mid i \in \mathbb{N}\}$ (def.distributive)

Kleene fixed-point theorem: Proof (ctd)

Assume f is distributive.

Hence $f(\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}) = \bigsqcup\{f^{i+1}(\bot) \mid i \in \mathbb{N}\}$ (def.distributive)

$= \bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ ($\bigsqcup(X \cup \{\bot\}) = \bigsqcup X$)

Kleene fixed-point theorem: Proof (ctd)

Assume f is distributive.

Hence $f(\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}) = \bigsqcup\{f^{i+1}(\bot) \mid i \in \mathbb{N}\}$ (def.distributive)

$= \bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ ($\bigsqcup(X \cup \{\bot\}) = \bigsqcup X$)

I.e., $\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ is fixed point

Kleene fixed-point theorem: Proof (ctd)

Assume f is distributive.

Hence $f(\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}) = \bigsqcup\{f^{i+1}(\bot) \mid i \in \mathbb{N}\}$ (def.distributive)

$= \bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ ($\bigsqcup(X \cup \{\bot\}) = \bigsqcup X$)

I.e., $\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ is fixed point

Hence $\mathrm{lfp}(f) \sqsubseteq \bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ (lfp is least fixed point)

Kleene fixed-point theorem: Proof (ctd)

Assume f is distributive.

Hence $f(\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}) = \bigsqcup\{f^{i+1}(\bot) \mid i \in \mathbb{N}\}$ (def.distributive)

$= \bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ ($\bigsqcup(X \cup \{\bot\}) = \bigsqcup X$)

I.e., $\bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ is fixed point

Hence $\mathrm{lfp}(f) \sqsubseteq \bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ (lfp is least fixed point)

With distributive implies mono, antisymmetry and first part, we get:

$$\mathrm{lfp}(f) = \bigsqcup\{f^i(\bot) \mid i \in \mathbb{N}\} \quad \square$$

Used Facts

bot least $\forall x.\ \bot \sqsubseteq x$

fixed point d is fixed point iff $f(d) = d$

least fixed point $f(d) = d \implies \mathrm{lfp}(f) \sqsubseteq d$

least upper bound $(\forall x \in X.\ x \sqsubseteq d) \implies \bigsqcup X \sqsubseteq d$

Summary

- Does $\mathrm{lfp}(F)$ exist?
  - Yes (Knaster-Tarski)

Summary

- Does $\mathrm{lfp}(F)$ exist?
  - Yes (Knaster-Tarski)
- Does fp-iteration compute it?
  - Fp-iteration computes the $F^i(\perp)$ for increasing i
    - By Kleene FP-Theorem, these are below $\mathrm{lfp}(F)$
  - It terminates only if a fixed-point has been reached
    - This fixed point is also below $\mathrm{lfp}(F)$ (and thus = $\mathrm{lfp}(F)$)

Note

- For any monotonic function f, we have

  $$f^{i}(\perp) \sqsubseteq f^{i+1}(\perp)$$

  - Straightforward induction on i

Table of Contents

Naive FP-iteration, again

Input Constraint system $x_i \sqsupseteq f_i(\vec{x})$

1. $\vec{x} := (\bot, \dots, \bot)$
2. $\vec{x} := F(\vec{x})$ (Recall $F(\vec{x}) = (f_1(\vec{x}), \dots, f_n(\vec{x}))$)
3. If $\neg(F(\vec{x}) \sqsubseteq \vec{x})$, goto 2
4. Return "$\vec{x}$ is least solution"

Naive FP-iteration, again

Input Constraint system $x_i \sqsupseteq f_i(\vec{x})$

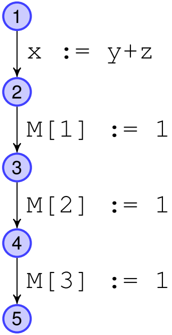
1. $\vec{x} := (\bot, \dots, \bot)$
2. $\vec{x} := F(\vec{x})$ (Recall $F(\vec{x}) = (f_1(\vec{x}), \dots, f_n(\vec{x}))$)
3. If $\neg(F(\vec{x}) \sqsubseteq \vec{x})$, goto 2
4. Return "$\vec{x}$ is least solution"

Note Originally, we had $\vec{x} := \vec{x} \sqcup F(\vec{x})$ in Step 2 and $F(\vec{x}) \neq \vec{x}$ in Step 3

- Also correct, as $F^i(\bot) \leq F^{i+1}(\bot)$, i.e., $\vec{x} \sqsubseteq F(\vec{x})$
- Saves $\sqcup$ operation.
- $\sqsubseteq$ may be more efficient than $=$.

Caveat

Naive fp-iteration may be rather inefficient

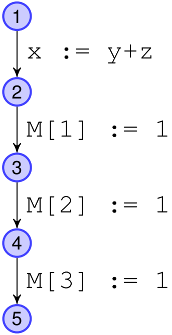


Let $S := (\text{Expr} \cup \{y + z\}) - \text{Expr}_x$

	0
$A[1]$	Expr
$A[2]$	Expr
$A[3]$	Expr
$A[4]$	Expr
$A[5]$	Expr

Caveat

Naive fp-iteration may be rather inefficient

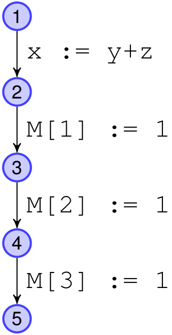


Let $S := (\mathrm{Expr} \cup \{y + z\}) - \mathrm{Expr}_x$

	0	1
$A[1]$	Expr	$\emptyset$
$A[2]$	Expr	S
$A[3]$	Expr	Expr
$A[4]$	Expr	Expr
$A[5]$	Expr	Expr

Caveat

Naive fp-iteration may be rather inefficient

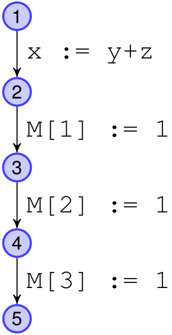


Let $S := (\text{Expr} \cup \{y + z\}) - \text{Expr}_x$

	0	1	2
$A[1]$	Expr	$\emptyset$	$\emptyset$
$A[2]$	Expr	S	$\{y + z\}$
$A[3]$	Expr	Expr	S
$A[4]$	Expr	Expr	Expr
$A[5]$	Expr	Expr	Expr

Caveat

Naive fp-iteration may be rather inefficient

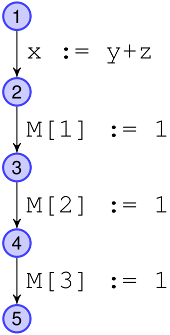


Let $S := (\text{Expr} \cup \{y + z\}) - \text{Expr}_x$

	0	1	2	3
$A[1]$	Expr	$\emptyset$	$\emptyset$	$\emptyset$
$A[2]$	Expr	S	$\{y + z\}$	$\{y + z\}$
$A[3]$	Expr	Expr	S	$\{y + z\}$
$A[4]$	Expr	Expr	Expr	S
$A[5]$	Expr	Expr	Expr	Expr

Caveat

Naive fp-iteration may be rather inefficient

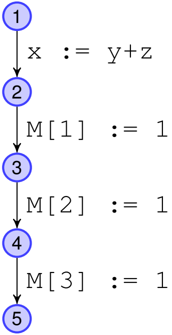


Let $S := (\mathrm{Expr} \cup \{y + z\}) - \mathrm{Expr}_x$

	0	1	2	3	4
$A[1]$	Expr	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$A[2]$	Expr	S	$\{y + z\}$	$\{y + z\}$	$\{y + z\}$
$A[3]$	Expr	Expr	S	$\{y + z\}$	$\{y + z\}$
$A[4]$	Expr	Expr	Expr	S	$\{y + z\}$
$A[5]$	Expr	Expr	Expr	Expr	S

Caveat

Naive fp-iteration may be rather inefficient

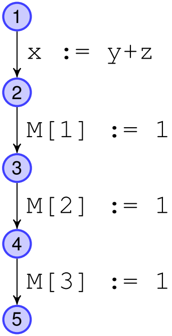


Let $S := (\mathrm{Expr} \cup \{y + z\}) - \mathrm{Expr}_x$

	0	1	2	3	4	5
$A[1]$	Expr	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$A[2]$	Expr	S	$\{y+z\}$	$\{y+z\}$	$\{y+z\}$	$\{y+z\}$
$A[3]$	Expr	Expr	S	$\{y+z\}$	$\{y+z\}$	$\{y+z\}$
$A[4]$	Expr	Expr	Expr	S	$\{y+z\}$	$\{y+z\}$
$A[5]$	Expr	Expr	Expr	Expr	S	$\{y+z\}$

Round-Robin iteration

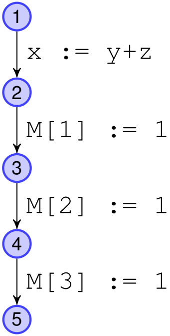
Idea: Instead of values from last iteration, use current values while computing RHSs.



	0
$A[1]$	Expr
$A[2]$	Expr
$A[3]$	Expr
$A[4]$	Expr
$A[5]$	Expr

Round-Robin iteration

Idea: Instead of values from last iteration, use current values while computing RHSs.



	0	1
$A[1]$	Expr	$\emptyset$
$A[2]$	Expr	$\{y + z\}$
$A[3]$	Expr	$\{y + z\}$
$A[4]$	Expr	$\{y + z\}$
$A[5]$	Expr	$\{y + z\}$

RR-Iteration: Pseudocode

```
x⃗ := (⊥, ..., ⊥)
do {
  finished := true
  for (i=1;i<=n;++i) {
    new := f_i(x⃗) // Evaluate RHS
    if (x_i ≠ new) { // If something changed
      finished = false    // No fp reached yet
      x_i := x_i ⊔ new // Update variable
    }
  }
} while (!finished)
return x⃗
```

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \ldots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\perp, \ldots, \perp) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \dots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \dots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def.$\sqsubseteq$ on $\mathbb{D}^n$)

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \ldots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def.$\sqsubseteq$ on $\mathbb{D}^n$)
  - From (IH) we get $F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (mono, fixed-point)

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \dots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def. $\sqsubseteq$ on $\mathbb{D}^n$)
  - From (IH) we get $F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (mono, fixed-point)
  - Hence $\vec{x} \sqcup F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (least-upper-bound, IH)

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \ldots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def.$\sqsubseteq$ on $\mathbb{D}^n$)
  - From (IH) we get $F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (mono, fixed-point)
  - Hence $\vec{x} \sqcup F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (least-upper-bound, IH)
  - Together: $\vec{x}' \sqsubseteq \mathrm{lfp}(F)$ (trans)

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \dots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def.$\sqsubseteq$ on $\mathbb{D}^n$)
  - From (IH) we get $F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (mono, fixed-point)
  - Hence $\vec{x} \sqcup F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (least-upper-bound, IH)
  - Together: $\vec{x}' \sqsubseteq \mathrm{lfp}(F)$ (trans)

Moreover, if algorithm terminates, we have $\vec{x} = F(\vec{x})$

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \dots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def.$\sqsubseteq$ on $\mathbb{D}^n$)
  - From (IH) we get $F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (mono, fixed-point)
  - Hence $\vec{x} \sqcup F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (least-upper-bound, IH)
  - Together: $\vec{x}' \sqsubseteq \mathrm{lfp}(F)$ (trans)

Moreover, if algorithm terminates, we have $\vec{x} = F(\vec{x})$

- I.e., $\vec{x}$ is a fixed-point.

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \dots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def.$\sqsubseteq$ on $\mathbb{D}^n$)
  - From (IH) we get $F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (mono, fixed-point)
  - Hence $\vec{x} \sqcup F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (least-upper-bound, IH)
  - Together: $\vec{x}' \sqsubseteq \mathrm{lfp}(F)$ (trans)

Moreover, if algorithm terminates, we have $\vec{x} = F(\vec{x})$

- I.e., $\vec{x}$ is a fixed-point.
- Invariant: $\vec{x} \sqsubseteq$ least fixed point

RR-Iteration: Correctness

Prove invariant: $\vec{x} \sqsubseteq \mathrm{lfp}(F)$

- Initially, $(\bot, \dots, \bot) \sqsubseteq \mathrm{lfp}(F)$ holds (bot-least)
- On update:
  - We have (1): $\vec{x}' = \vec{x}(i := x_i \sqcup f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq \mathrm{lfp}(F)$
  - From (1) we get $\vec{x}' \sqsubseteq \vec{x} \sqcup F(\vec{x})$ (def. $\sqsubseteq$ on $\mathbb{D}^n$)
  - From (IH) we get $F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (mono, fixed-point)
  - Hence $\vec{x} \sqcup F(\vec{x}) \sqsubseteq \mathrm{lfp}(F)$ (least-upper-bound, IH)
  - Together: $\vec{x}' \sqsubseteq \mathrm{lfp}(F)$ (trans)

Moreover, if algorithm terminates, we have $\vec{x} = F(\vec{x})$

- I.e., $\vec{x}$ is a fixed-point.
- Invariant: $\vec{x} \sqsubseteq$ least fixed point
- Thus: $\vec{x} = \mathrm{lfp}(F)$

Used Facts

trans $x \sqsubseteq y \sqsubseteq z \implies x \sqsubseteq z$

RR-Iteration: Improved Algorithm

We can save some operations

- Use $\sqsubseteq$ instead of $=$ in test
- No $\sqcup$ on update

$\vec{x} := (\bot, \dots, \bot)$
```
do {
  finished := true
  for (i=1;i<=n;++i) {
    new := f_i(x⃗) // Evaluate RHS
    if (¬(x_i ⊒ new)) { // If something changed
      finished = false    // No fp reached yet
      x_i := new // Update variable
    }
  }
} while (!finished)
return x⃗
```

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:
  - We have $\vec{x}' = \vec{x}(i := f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq F(\vec{x})$

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:
  - We have $\vec{x}' = \vec{x}(i := f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq F(\vec{x})$
  - Hence $\vec{x} \sqsubseteq \vec{x}' \sqsubseteq F(\vec{x})$ (Def.$\sqsubseteq$, IH)

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:
  - We have $\vec{x}' = \vec{x}(i := f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq F(\vec{x})$
  - Hence $\vec{x} \sqsubseteq \vec{x}' \sqsubseteq F(\vec{x})$ (Def.$\sqsubseteq$, IH)
  - Hence $F(\vec{x}) \sqsubseteq F(\vec{x}')$ (mono)

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:
  - We have $\vec{x}' = \vec{x}(i := f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq F(\vec{x})$
  - Hence $\vec{x} \sqsubseteq \vec{x}' \sqsubseteq F(\vec{x})$ (Def.$\sqsubseteq$, IH)
  - Hence $F(\vec{x}) \sqsubseteq F(\vec{x}')$ (mono)
  - Together $\vec{x}' \sqsubseteq F(\vec{x}')$ (trans)

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:
  - We have $\vec{x}' = \vec{x}(i := f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq F(\vec{x})$
  - Hence $\vec{x} \sqsubseteq \vec{x}' \sqsubseteq F(\vec{x})$ (Def.$\sqsubseteq$, IH)
  - Hence $F(\vec{x}) \sqsubseteq F(\vec{x}')$ (mono)
  - Together $\vec{x}' \sqsubseteq F(\vec{x}')$ (trans)

With this invariant, we have

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:
  - We have $\vec{x}' = \vec{x}(i := f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq F(\vec{x})$
  - Hence $\vec{x} \sqsubseteq \vec{x}' \sqsubseteq F(\vec{x})$ (Def.$\sqsubseteq$, IH)
  - Hence $F(\vec{x}) \sqsubseteq F(\vec{x}')$ (mono)
  - Together $\vec{x}' \sqsubseteq F(\vec{x}')$ (trans)

With this invariant, we have

- $x_i = f_i(\vec{x})$ iff $x_i \sqsupseteq f_i(\vec{x})$ (antisym)

RR-Iteration: Improved Algorithm: Correctness

Justification: Invariant $\vec{x} \sqsubseteq F(\vec{x})$

- Holds initially: Obvious
- On update:
  - We have $\vec{x}' = \vec{x}(i := f_i(\vec{x}))$. We assume (IH): $\vec{x} \sqsubseteq F(\vec{x})$
  - Hence $\vec{x} \sqsubseteq \vec{x}' \sqsubseteq F(\vec{x})$ (Def.$\sqsubseteq$, IH)
  - Hence $F(\vec{x}) \sqsubseteq F(\vec{x}')$ (mono)
  - Together $\vec{x}' \sqsubseteq F(\vec{x}')$ (trans)

With this invariant, we have

- $x_i = f_i(\vec{x})$ iff $x_i \sqsupseteq f_i(\vec{x})$ (antisym)
- $x_i \sqcup f_i(\vec{x}) = f_i(\vec{x})$ (sup-absorb)
  - sup-absorb: $x \sqsubseteq y \implies x \sqcup y = y$

RR-Iteration: Termination

Definition (Chain)

A set $C \subseteq \mathbb{D}$ is called chain, iff all elements are mutually comparable:

$$\forall c_1, c_2 \in C.\ c_1 \sqsubseteq c_2 \vee c_2 \sqsubseteq c_1$$

A partial order has finite height, iff every chain is finite. Then, the height $h \in \mathbb{N}$ is the maximum cardinality of any chain.

RR-Iteration: Termination

Definition (Chain)

A set $C \subseteq \mathbb{D}$ is called chain, iff all elements are mutually comparable:

$$\forall c_1, c_2 \in C.\ c_1 \sqsubseteq c_2 \vee c_2 \sqsubseteq c_1$$

A partial order has finite height, iff every chain is finite. Then, the height $h \in \mathbb{N}$ is the maximum cardinality of any chain.

For a domain with finite chain height h, RR-iteration terminates within $O(n^2 h)$ RHS-evaluations.

- In each iteration of the outer loop, at least one variable increases, or the algorithm terminates. A variable may only increase $h - 1$ times.

Last Lecture

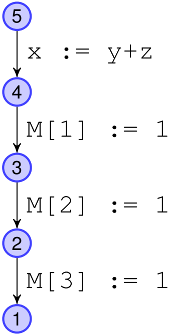
- Monotonic functions
  - Constraint system modeled as function
  - Least solution is least fixed point
- Knaster-Tarski fp-thm:
  - lfp of monotonic function exists
- Kleene fp theorem:
  - Iterative characterization of lfp for distributive functions
  - Justifies naive fp-iteration
- Round-Robin iteration
  - Improves on naive iteration by using values of current round
  - Still depends on variable ordering

Problem:

The efficiency of RR depends on variable ordering

Problem:

The efficiency of RR depends on variable ordering

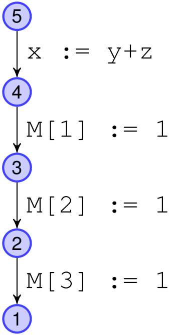


Let $S := (\mathrm{Expr} \cup \{y+z\}) - \mathrm{Expr}_x$

	0	1
$A[1]$	Expr	Expr
$A[2]$	Expr	Expr
$A[3]$	Expr	Expr
$A[4]$	Expr	S
$A[5]$	Expr	$\emptyset$

Problem:

The efficiency of RR depends on variable ordering

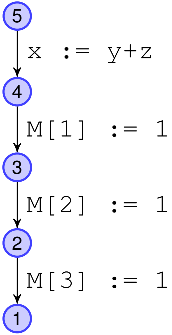


Let $S := (\mathrm{Expr} \cup \{y+z\}) - \mathrm{Expr}_x$

	0	1	2
$A[1]$	Expr	Expr	Expr
$A[2]$	Expr	Expr	Expr
$A[3]$	Expr	Expr	S
$A[4]$	Expr	S	$\{y+z\}$
$A[5]$	Expr	$\emptyset$	$\emptyset$

Problem:

The efficiency of RR depends on variable ordering

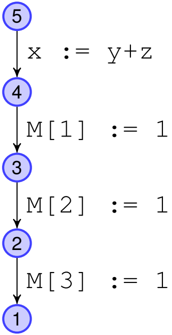


Let $S := (\mathrm{Expr} \cup \{y+z\}) - \mathrm{Expr}_x$

	0	1	2	3
$A[1]$	Expr	Expr	Expr	Expr
$A[2]$	Expr	Expr	Expr	S
$A[3]$	Expr	Expr	S	$\{y+z\}$
$A[4]$	Expr	S	$\{y+z\}$	$\{y+z\}$
$A[5]$	Expr	$\emptyset$	$\emptyset$	$\emptyset$

Problem:

The efficiency of RR depends on variable ordering

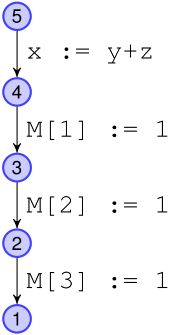


Let $S := (\text{Expr} \cup \{y + z\}) - \text{Expr}_x$

	0	1	2	3	4
$A[1]$	Expr	Expr	Expr	Expr	S
$A[2]$	Expr	Expr	Expr	S	$\{y + z\}$
$A[3]$	Expr	Expr	S	$\{y + z\}$	$\{y + z\}$
$A[4]$	Expr	S	$\{y + z\}$	$\{y + z\}$	$\{y + z\}$
$A[5]$	Expr	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Problem:

The efficiency of RR depends on variable ordering



Let $S := (\text{Expr} \cup \{y+z\}) - \text{Expr}_x$

	0	1	2	3	4	5
$A[1]$	Expr	Expr	Expr	Expr	S	$\{y+z\}$
$A[2]$	Expr	Expr	Expr	S	$\{y+z\}$	$\{y+z\}$
$A[3]$	Expr	Expr	S	$\{y+z\}$	$\{y+z\}$	$\{y+z\}$
$A[4]$	Expr	S	$\{y+z\}$	$\{y+z\}$	$\{y+z\}$	$\{y+z\}$
$A[5]$	Expr	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Problem:

The efficiency of RR depends on variable ordering

Rule of thumb

- u before v, if $u \to^{*} v$
- Entry condition before loop body

Worklist algorithm

Problems of RR (remaining)

- Complete round required to detect termination
- If only one variable changes, everything is re-computed
- Depends on variable ordering.

Worklist algorithm

Problems of RR (remaining)

Complete round required to detect termination
If only one variable changes, everything is re-computed
Depends on variable ordering.

Idea of worklist algorithm

- Store constraints whose RHS may have changed in a list

Worklist Algorithm: Pseudocode

```
W = {1...n}
x⃗ = (⊥,...,⊥)

while (W != ε) {
  get an i ∈ W, W = W - {i}

  t = f_i(x⃗)
  if (¬(t ⊑ x_i)) {
    x_i = t
    W = W ∪ {j | f_j depends on variable i}
  }
}
```

Worklist Algorithm: Example

- On whiteboard

Worklist Algorithm: Correctness

Invariants

1. $\vec{x} \sqsubseteq F(\vec{x})$ and $\vec{x} \sqsubseteq \mathrm{lfp} F$
   - Same argument as for RR-iteration
2. $\neg(x_i \sqsupseteq f_i(\vec{x})) \implies i \in W$
   - Intuitively: Constraints that are not satisfied are on worklist
   - Initially, all i in W
   - On update: Only RHS that depend on updated variable may change. Exactly these are added to W.
     If f_i does not depend on variable i, the constraint i holds for the new $\vec{x}$, so its removal from W is OK.

- If loop terminates: Due to Inv. 2, we have solution. Due to Inv. 1, it is least solution.

Worklist Algorithm: Termination

Theorem

For a monotonic CS and a domain with finite height h, the worklist algorithm returns the least solution and terminates within $O(hN)$ iterations, where N is the size of the constraint system:

$$N := \sum_{i=1}^{n} 1 + |f_i| \textit{ where } |f_i| := |\{i \mid f_i \textit{ depends on variable } i\}|$$

Worklist Algorithm: Termination

Theorem

For a monotonic CS and a domain with finite height h, the worklist algorithm returns the least solution and terminates within $O(hN)$ iterations, where N is the size of the constraint system:

$$N := \sum_{i=1}^{n} 1 + |f_i| \textit{ where } |f_i| := |\{i \mid f_i \textit{ depends on variable } i\}|$$

Proof (Sketch):

Worklist Algorithm: Termination

Theorem

For a monotonic CS and a domain with finite height h, the worklist algorithm returns the least solution and terminates within $O(hN)$ iterations, where N is the size of the constraint system:

$$N := \sum_{i=1}^{n} 1 + |f_i| \text{ where } |f_i| := |\{i \mid f_i \text{ depends on variable } i\}|$$

Proof (Sketch):

- Number of iterations = Number of elements added to W.

Worklist Algorithm: Termination

Theorem

For a monotonic CS and a domain with finite height h, the worklist algorithm returns the least solution and terminates within $O(hN)$ iterations, where N is the size of the constraint system:

$$N := \sum_{i=1}^{n} 1 + |f_i| \text{ where } |f_i| := |\{i \mid f_i \text{ depends on variable } i\}|$$

Proof (Sketch):

- Number of iterations = Number of elements added to W.
- Initially: n elements

Worklist Algorithm: Termination

Theorem

For a monotonic CS and a domain with finite height h, the worklist algorithm returns the least solution and terminates within $O(hN)$ iterations, where N is the size of the constraint system:

$$N := \sum_{i=1}^{n} 1 + |f_i| \textit{ where } |f_i| := |\{i \mid f_i \textit{ depends on variable } i\}|$$

Proof (Sketch):

- Number of iterations = Number of elements added to W.
- Initially: n elements
- Constraint i added if variable its RHS depends on is changed
  - Variable may not change more than h times. Constraint depends on $|f_i|$ variables.

Worklist Algorithm: Termination

Theorem

For a monotonic CS and a domain with finite height h, the worklist algorithm returns the least solution and terminates within $O(hN)$ iterations, where N is the size of the constraint system:

$$N := \sum_{i=1}^{n} 1 + |f_i| \text{ where } |f_i| := |\{i \mid f_i \text{ depends on variable } i\}|$$

Proof (Sketch):

- Number of iterations = Number of elements added to W.
- Initially: n elements
- Constraint i added if variable its RHS depends on is changed
  - Variable may not change more than h times. Constraint depends on $|f_i|$ variables.
- Thus, no more than

  $$n + \sum_{i=1}^{n} h|f_i| = hN$$

  elements added to worklist. $\square$

Worklist Algorithm: Problems

- Dependencies of RHS need to be known.
  - No problem for our application

Worklist Algorithm: Problems

- Dependencies of RHS need to be known.
    - No problem for our application
- Which constraint to select next from worklist?
    - Requires strategy.

Worklist Algorithm: Problems

- Dependencies of RHS need to be known.
  - No problem for our application
- Which constraint to select next from worklist?
  - Requires strategy.
- Various more advanced algorithms exists
  - Determine dependencies dynamically (Generic solvers)
  - Only compute solution for subset of the variables (Local solvers)
  - Even: Local generic solvers

Summary:

- Constraint systems (over complete lattice, monotonic RHSs)
  - Encode as monotonic function $F : \mathbb{D}^n \to \mathbb{D}^n$
  - (Least) Solution = (least) fixed point
- Knaster-Tarski theorem: A least solution always exists
- Solve by fixpoint-iteration (naive, RR, WL)
  - Kleene-Theorem justifies naive fixpoint iteration
  - Similar ideas to justify RR, WL

Summary:

- Constraint systems (over complete lattice, monotonic RHSs)
  - Encode as monotonic function $F : \mathbb{D}^n \to \mathbb{D}^n$
  - (Least) Solution = (least) fixed point
- Knaster-Tarski theorem: A least solution always exists
- Solve by fixpoint-iteration (naive, RR, WL)
  - Kleene-Theorem justifies naive fixpoint iteration
  - Similar ideas to justify RR, WL
- Still Missing:
  - Link between least solution of constraint system, and
    Available at u: $A[u] = \bigcap\{[\![\pi]\!]^\# \emptyset \mid \pi.\ v_0 \xrightarrow{\pi} u\}$

Table of Contents

Monotonic Analysis Framework

Given

Flowgraph

A complete lattice $(\mathbb{D}, \sqsubseteq)$.

An initialization value $d_0 \in \mathbb{D}$

An abstract effect $[\![k]\!]^\# : \mathbb{D} \to \mathbb{D}$ for edges k

- Such that $[\![k]\!]^\#$ is monotonic.

Monotonic Analysis Framework

Given: Flowgraph
A complete lattice $(\mathbb{D}, \sqsubseteq)$.
An initialization value $d_0 \in \mathbb{D}$
An abstract effect $[\![k]\!]^\# : \mathbb{D} \to \mathbb{D}$ for edges k

- Such that $[\![k]\!]^\#$ is monotonic.

Wanted: $\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\#(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$
MOP = Merge over all paths

Monotonic Analysis Framework

Given Flowgraph
A complete lattice $(\mathbb{D}, \sqsubseteq)$.
An initialization value $d_0 \in \mathbb{D}$
An abstract effect $[\![k]\!]^\# : \mathbb{D} \to \mathbb{D}$ for edges k

- Such that $[\![k]\!]^\#$ is monotonic.

Wanted $\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\#(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$
MOP = Merge over all paths

Method Compute least solution MFP of constraint system

$$\mathrm{MFP}[v_0] \sqsupseteq d_0 \qquad \text{(init)}$$

$$\mathrm{MFP}[v] \sqsupseteq [\![k]\!]^\#(\mathrm{MFP}[u]) \quad \text{for edges } k = (u, a, v) \qquad \text{(edge)}$$

MFP = Minimal fixed point

Kam, Ullmann

Kam, Ullman, 1975

In a monotonic analysis framework, we have

$$MOP \sqsubseteq MFP$$

Kam, Ullmann

Kam, Ullman, 1975

In a monotonic analysis framework, we have

$$MOP \sqsubseteq MFP$$

- Intuitively: The constraint system's least solution (MFP) is a correct approximation to the value defined over all paths reaching the program point (MOP).

Kam, Ullmann

Kam, Ullman, 1975

In a monotonic analysis framework, we have

$$MOP \sqsubseteq MFP$$

- Intuitively: The constraint system's least solution (MFP) is a correct approximation to the value defined over all paths reaching the program point (MOP).
- In particular: $[\![\pi]\!]^{\#}(d_0) \sqsubseteq \mathrm{MFP}[u]$ for $v_0 \xrightarrow{\pi} u$

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.$\mathrm{MOP}$, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup \{ [\![\pi]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u \} \sqsubseteq \mathrm{MFP}[u]$$

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.MOP, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^{\#} d_0 \sqsubseteq \mathrm{MFP}[u]$$

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.MOP, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.$\mathrm{MOP}$, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup \{ [\![\pi]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u \} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^{\#} d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.$\mathrm{MOP}$, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^{\#} d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.
  - We have $u = v_0$ (empty-path) and $[\![\varepsilon]\!]^{\#} d_0 = d_0$ (empty-eff)

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.$\mathrm{MOP}$, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.
  - We have $u = v_0$ (empty-path) and $[\![\varepsilon]\!]^\# d_0 = d_0$ (empty-eff)
  - As MFP is solution, the (init)-constraint yields $d_0 \sqsubseteq \mathrm{MFP}[v_0]$.

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.MOP, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.
  - We have $u = v_0$ (empty-path) and $[\![\varepsilon]\!]^\# d_0 = d_0$ (empty-eff)
  - As MFP is solution, the (init)-constraint yields $d_0 \sqsubseteq \mathrm{MFP}[v_0]$.
- Step case: $\pi = \pi' k$ for edge $k = (u, a, v)$

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.$\mathrm{MOP}$, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u. \bigsqcup\{[\![\pi]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^{\#} d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.
  - We have $u = v_0$ (empty-path) and $[\![\varepsilon]\!]^{\#} d_0 = d_0$ (empty-eff)
  - As MFP is solution, the (init)-constraint yields $d_0 \sqsubseteq \mathrm{MFP}[v_0]$.
- Step case: $\pi = \pi' k$ for edge $k = (u, a, v)$
  - Assume $v_0 \xrightarrow{\pi'} u \xrightarrow{a} v$ and (IH): $[\![\pi']\!]^{\#} d_0 \sqsubseteq \mathrm{MFP}[u]$.
    To show: $[\![\pi' k]\!]^{\#} d_0 \sqsubseteq \mathrm{MFP}[v]$

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.$\mathrm{MOP}$, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.
  - We have $u = v_0$ (empty-path) and $[\![\varepsilon]\!]^\# d_0 = d_0$ (empty-eff)
  - As MFP is solution, the (init)-constraint yields $d_0 \sqsubseteq \mathrm{MFP}[v_0]$.
- Step case: $\pi = \pi' k$ for edge $k = (u, a, v)$
  - Assume $v_0 \xrightarrow{\pi'} u \xrightarrow{a} v$ and (IH): $[\![\pi']\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$.
    To show: $[\![\pi' k]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[v]$
  - Have $[\![\pi' k]\!]^\# = [\![k]\!]^\#([\![\pi']\!]^\# d_0)$ (eff-comp)

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.$\mathrm{MOP}$, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.
  - We have $u = v_0$ (empty-path) and $[\![\varepsilon]\!]^\# d_0 = d_0$ (empty-eff)
  - As MFP is solution, the (init)-constraint yields $d_0 \sqsubseteq \mathrm{MFP}[v_0]$.
- Step case: $\pi = \pi' k$ for edge $k = (u, a, v)$
  - Assume $v_0 \xrightarrow{\pi'} u \xrightarrow{a} v$ and (IH): $[\![\pi']\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$.
    To show: $[\![\pi' k]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[v]$
  - Have $[\![\pi' k]\!]^\# = [\![k]\!]^\#([\![\pi']\!]^\# d_0)$ (eff-comp)
  - $\sqsubseteq [\![k]\!]^\#(\mathrm{MFP}[u])$ (IH,mono)

Kam, Ullman: Proof

To show $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$, i.e. (def.MOP, def.$\sqsubseteq$ on $\mathbb{D}^n$)

$$\forall u.\ \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\} \sqsubseteq \mathrm{MFP}[u]$$

It suffices to show that $\mathrm{MFP}[u]$ is an upper bound. (least-upper-bound)

$$\forall \pi, u.\ v_0 \xrightarrow{\pi} u \implies [\![\pi]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$$

Induction on π.

- Base case: $\pi = \varepsilon$.
  - We have $u = v_0$ (empty-path) and $[\![\varepsilon]\!]^\# d_0 = d_0$ (empty-eff)
  - As MFP is solution, the (init)-constraint yields $d_0 \sqsubseteq \mathrm{MFP}[v_0]$.
- Step case: $\pi = \pi' k$ for edge $k = (u, a, v)$
  - Assume $v_0 \xrightarrow{\pi'} u \xrightarrow{a} v$ and (IH): $[\![\pi']\!]^\# d_0 \sqsubseteq \mathrm{MFP}[u]$.
    To show: $[\![\pi' k]\!]^\# d_0 \sqsubseteq \mathrm{MFP}[v]$
  - Have $[\![\pi' k]\!]^\# = [\![k]\!]^\#([\![\pi']\!]^\# d_0)$ (eff-comp)
  - $\sqsubseteq [\![k]\!]^\#(\mathrm{MFP}[u])$ (IH,mono)
  - $\sqsubseteq \mathrm{MFP}[v]$ ((edge)-constraint, MFP is solution) $\square$

Facts

empty-path	$u \xrightarrow{\varepsilon} v \iff u = v$
empty-eff	$[\![\varepsilon]\!]^{\#} d = d$
eff-comp	$[\![\pi_1 \pi_2]\!]^{\#} = [\![\pi_2]\!]^{\#} \circ [\![\pi_1]\!]^{\#}$

Problem

- Yet another approximation :(
  - Recall: Abstract effect was already approximation

Problem

- Yet another approximation :(
  - Recall: Abstract effect was already approximation
- Good news:
  - If the right-hand sides are distributive, we can compute MOP exactly

Theorem of Kildal

Kildal, 1972

In a distributive analysis framework (i.e., a monotonic analysis framework where the $[\![k]\!]^\#$ are distributive), where all nodes are reachable, we have

$$MOP = MFP$$

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^{\#}(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^{\#}(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

(init) To show: $\mathrm{MOP}[v_0] \sqsupseteq d_0$

- Straightforward (upper-bound, empty-path, empty-eff)

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^{\#}(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

(init) To show: $\mathrm{MOP}[v_0] \sqsupseteq d_0$

- Straightforward (upper-bound, empty-path, empty-eff)

(edge) To show: $\mathrm{MOP}[v] \sqsupseteq [\![k]\!]^{\#}\mathrm{MOP}[u]$ for edge $k = (u, a, v)$

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^\#(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

(init) To show: $\mathrm{MOP}[v_0] \sqsupseteq d_0$

- Straightforward (upper-bound, empty-path, empty-eff)

(edge) To show: $\mathrm{MOP}[v] \sqsupseteq [\![k]\!]^\#\mathrm{MOP}[u]$ for edge $k = (u, a, v)$

- Note (*): $P[u]$ not empty, as all nodes reachable

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^{\#}(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

(init) To show: $\mathrm{MOP}[v_0] \sqsupseteq d_0$

- Straightforward (upper-bound, empty-path, empty-eff)

(edge) To show: $\mathrm{MOP}[v] \sqsupseteq [\![k]\!]^{\#}\mathrm{MOP}[u]$ for edge $k = (u, a, v)$

- Note (*): $P[u]$ not empty, as all nodes reachable
- $[\![k]\!]^{\#}\mathrm{MOP}[u] = \bigsqcup\{[\![k]\!]^{\#}([\![\pi]\!]^{\#} d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$ (def.MOP, distrib,*)

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^{\#}(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

(init) To show: $\mathrm{MOP}[v_0] \sqsupseteq d_0$

- Straightforward (upper-bound, empty-path, empty-eff)

(edge) To show: $\mathrm{MOP}[v] \sqsupseteq [\![k]\!]^{\#}\mathrm{MOP}[u]$ for edge $k = (u, a, v)$

- Note (*): $P[u]$ not empty, as all nodes reachable
- $[\![k]\!]^{\#}\mathrm{MOP}[u] = \bigsqcup\{[\![k]\!]^{\#}([\![\pi]\!]^{\#} d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$ (def.MOP, distrib,*)
- $= \bigsqcup\{[\![\pi k]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi k} v\}$ (def.$[\![\cdot]\!]^{\#}$ on paths. k is edge, path-append)

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^{\#}(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

(init) To show: $\mathrm{MOP}[v_0] \sqsupseteq d_0$

- Straightforward (upper-bound, empty-path, empty-eff)

(edge) To show: $\mathrm{MOP}[v] \sqsupseteq [\![k]\!]^{\#}\mathrm{MOP}[u]$ for edge $k = (u, a, v)$

- Note (*): $P[u]$ not empty, as all nodes reachable
- $[\![k]\!]^{\#}\mathrm{MOP}[u] = \bigsqcup\{[\![k]\!]^{\#}([\![\pi]\!]^{\#} d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$ (def.MOP, distrib,*)
- $= \bigsqcup\{[\![\pi k]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi k} v\}$ (def.$[\![\cdot]\!]^{\#}$ on paths. k is edge, path-append)
- $\sqsubseteq \bigsqcup\{[\![\pi]\!]^{\#} d_0 \mid \pi.\ v_0 \xrightarrow{\pi} v\}$ (sup-subset)

Proof

We already know $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$. To show that also $\mathrm{MFP} \sqsubseteq \mathrm{MOP}$, it suffices to show that MOP is a solution of the constraint system.

- As MFP is least solution, the proposition follows.

- Recall:

$$\mathrm{MOP}[u] := \bigsqcup P[u], \text{ where } P[u] := \{[\![\pi]\!]^\#(d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$$

(init) To show: $\mathrm{MOP}[v_0] \sqsupseteq d_0$

- Straightforward (upper-bound, empty-path, empty-eff)

(edge) To show: $\mathrm{MOP}[v] \sqsupseteq [\![k]\!]^\# \mathrm{MOP}[u]$ for edge $k = (u, a, v)$

- Note (*): $P[u]$ not empty, as all nodes reachable
- $[\![k]\!]^\# \mathrm{MOP}[u] = \bigsqcup\{[\![k]\!]^\#([\![\pi]\!]^\# d_0) \mid \pi.\ v_0 \xrightarrow{\pi} u\}$ (def.MOP, distrib,*)
- $= \bigsqcup\{[\![\pi k]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi k} v\}$ (def.$[\![\cdot]\!]^\#$ on paths. k is edge, path-append)
- $\sqsubseteq \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} v\}$ (sup-subset)
- $= \mathrm{MOP}[v]$ (def.MOP) $\square$

Facts

path-append $k = (u, a, v) \in E \wedge v_0 \xrightarrow{\pi} u \iff v_0 \xrightarrow{\pi k} v$

- Append edge to path

sup-subset $X \subseteq Y \implies \bigsqcup X \sqsubseteq \bigsqcup Y$

Note

Reachability of all nodes is essential

- No paths to unreachable node u, i.e., $\mathrm{MOP}[u] = \bot$
- But edges from other unreachable nodes possible

$\Longrightarrow$ Constraint of form $\mathrm{MFP}[u] \sqsupseteq \ldots$

Note

Reachability of all nodes is essential

- No paths to unreachable node u, i.e., $\mathrm{MOP}[u] = \bot$
- But edges from other unreachable nodes possible

$\Longrightarrow$ Constraint of form $\mathrm{MFP}[u] \sqsupseteq \ldots$

Eliminate unreachable nodes before creating CS

- E.g. by DFS from start node.

Depth first search (pseudocode)

```
void dfs (node u) {
  if u ∉ R {
    R := R ∪ {u}
    for all v with (u,a,v) ∈ E {dfs v}
  }
}

void find_reachable () {
  R = {}
  dfs(v0)
  // R contains reachable nodes now
}
```

Summary

Input CFG, distributive/(monotonic) analysis framework

- Framework defines domain $(\mathbb{D}, \sqsubseteq)$, initial value $d_0 \in \mathbb{D}$ and abstract effects $[\![\cdot]\!]^\# : E \to \mathbb{D} \to \mathbb{D}$
- For each edge k, $[\![k]\!]^\#$ is distributive/(monotonic)

1. Eliminate unreachable nodes
2. Put up constraint system
3. Solve by worklist-algo, RR-iteration, ...

Output (Safe approximation of) MOP - solution

Summary

Input CFG, distributive/(monotonic) analysis framework

- Framework defines domain $(\mathbb{D}, \sqsubseteq)$, initial value $d_0 \in \mathbb{D}$ and abstract effects $[\![\cdot]\!]^\# : E \to \mathbb{D} \to \mathbb{D}$
- For each edge k, $[\![k]\!]^\#$ is distributive/(monotonic)

1. Eliminate unreachable nodes
2. Put up constraint system
3. Solve by worklist-algo, RR-iteration, ...

Output (Safe approximation of) MOP - solution

Note Abstract effects of available expressions are distributive

- As all functions of the form: $x \mapsto (a \cup x) \setminus b$

Last lecture

- Worklist algorithm: Find least solution with $O(hN)$ RHS-evaluations
  - h height of domain, N size of constraint system
- Monotonic analysis framework: $(\mathbb{D}, \subseteq)$, $d_0 \in D$, $[\![\cdot]\!]^\#$ (monotonic)
  - Yields $\mathrm{MOP}[u] = \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \pi.\ v_0 \xrightarrow{\pi} u\}$
- Theorems of Kam/Ullman and Kildal
  - $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$,
  - Distributive framework and all nodes reachable: $\mathrm{MOP} = \mathrm{MFP}$
- Started with dead-assignment elimination

Summary (II) – How to develop a program optimization

- Optimization = Analysis + Transformation

Summary (II) – How to develop a program optimization

- Optimization = Analysis + Transformation
- Create semantic description of analysis result
  - Result for each program point
  - Depends on states reachable at this program point
  - In general, not computable
  - Prove transformation correct for (approximations of) this result

Summary (II) – How to develop a program optimization

- Optimization = Analysis + Transformation
- Create semantic description of analysis result
  - Result for each program point
  - Depends on states reachable at this program point
  - In general, not computable
  - Prove transformation correct for (approximations of) this result
- Create syntactic approximation of analysis result
  - Abstract effect of edges
  - Yields monotonic/distributive analysis framework

Summary (II) – How to develop a program optimization

- Optimization = Analysis + Transformation
- Create semantic description of analysis result
  - Result for each program point
  - Depends on states reachable at this program point
  - In general, not computable
  - Prove transformation correct for (approximations of) this result
- Create syntactic approximation of analysis result
  - Abstract effect of edges
  - Yields monotonic/distributive analysis framework
- Compute MFP.
  - Approximation of semantic result

Summary (II) – How to develop a program optimization

- Optimization = Analysis + Transformation
- Create semantic description of analysis result
  - Result for each program point
  - Depends on states reachable at this program point
  - In general, not computable
  - Prove transformation correct for (approximations of) this result
- Create syntactic approximation of analysis result
  - Abstract effect of edges
  - Yields monotonic/distributive analysis framework
- Compute MFP.
  - Approximation of semantic result
- Perform transformation based on MFP

Table of Contents

Now: Dead-Assignment Elimination

Example

```
1: x = y + 2;
2: y = 4;
3: x = y + 3
```

Now: Dead-Assignment Elimination

Example

```
1: x = y + 2;
2: y = 4;
3: x = y + 3
```

Value of *x* computed in line 1 never used

Now: Dead-Assignment Elimination

Example

```
1: x = y + 2;
2: y = 4;
3: x = y + 3
```

Value of *x* computed in line 1 never used

Equivalent program:

```
1: nop;
2: y = 4;
3: x = y + 3
```

Now: Dead-Assignment Elimination

Example

```
1: x = y + 2;
2: y = 4;
3: x = y + 3
```

Value of *x* computed in line 1 never used

Equivalent program:

```
1: nop;
2: y = 4;
3: x = y + 3
```

- `x` is called dead at 1.

Live registers (semantically)

Register x is semantically live at program point u, iff there is an execution to an end node, that depends on the value of x at u:

$$
\begin{aligned}
x \in \mathrm{Live}[u] \iff \exists \pi, v, \rho, \mu, a.\ & u \xrightarrow{\pi} v \wedge v \in V_{\mathrm{end}} \\
& \wedge (\rho, \mu) \in [\![u]\!] \\
& \wedge [\![\pi]\!](\rho(x := a), \mu) \neq_X [\![\pi]\!](\rho, \mu)
\end{aligned}
$$

Where $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\ v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$

- Intuition: All states reachable at u
- Collecting semantics

Live registers (semantically)

Register x is semantically live at program point u, iff there is an execution to an end node, that depends on the value of x at u:

$$
\begin{aligned}
x \in \mathrm{Live}[u] \iff & \exists \pi, v, \rho, \mu, a. \\
& u \xrightarrow{\pi} v \wedge v \in V_{\mathrm{end}} \\
& \wedge (\rho, \mu) \in [\![u]\!] \\
& \wedge [\![\pi]\!](\rho(x := a), \mu) \neq_X [\![\pi]\!](\rho, \mu)
\end{aligned}
$$

Where $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\ v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$

- Intuition: All states reachable at u
- Collecting semantics

- $(\rho, \mu) =_X (\rho', \mu')$ iff $\mu = \mu'$ and $\forall x \in X.\ \rho(x) = \rho'(x)$
  - Equal on memory and "interesting" registers X

Live registers (semantically)

Register x is semantically live at program point u, iff there is an execution to an end node, that depends on the value of x at u:

$$\begin{aligned} x \in \mathrm{Live}[u] \iff \exists \pi, v, \rho, \mu, a.\; & u \xrightarrow{\pi} v \wedge v \in V_{\mathrm{end}} \\ & \wedge (\rho, \mu) \in [\![u]\!] \\ & \wedge [\![\pi]\!](\rho(x := a), \mu) \neq_X [\![\pi]\!](\rho, \mu) \end{aligned}$$

Where $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\; v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$

- Intuition: All states reachable at u
- Collecting semantics

- $(\rho, \mu) =_X (\rho', \mu')$ iff $\mu = \mu'$ and $\forall x \in X.\; \rho(x) = \rho'(x)$
  - Equal on memory and "interesting" registers X
- x is semantically dead at u, iff it is not live.
  - No execution depends on the value of x at u.

Transformation: Dead-Assignment Elimination

- Replace assignments/loads to dead registers by `Nop`
- $(u, x := *, v) \mapsto (u, \mathrm{Nop}, v)$ if x dead at v
- Obviously correct
  - States reachable at end nodes are preserved
- Correct approximation: Less dead variables (= More live variables)

Live registers (syntactic approximation)

Register x is live at u ($x \in L[u]$), iff there is a path $u \xrightarrow{\pi} v$, $v \in V_{\mathrm{end}}$, such that

- π does not contain writes to x, and $x \in X$
- or π contains a read of x before the first write to x

Live registers (syntactic approximation)

Register x is live at u ($x \in L[u]$), iff there is a path $u \xrightarrow{\pi} v$, $v \in V_{\mathrm{end}}$, such that

- π does not contain writes to x, and $x \in X$
- or π contains a read of x before the first write to x

Abstract effects, propagating live variables backwards over edge

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^\# L &= L \\
[\![\mathrm{Pos}(e)]\!]^\# L &= L \cup \mathrm{regs}(e) \\
[\![\mathrm{Neg}(e)]\!]^\# L &= L \cup \mathrm{regs}(e) \\
[\![x := e]\!]^\# L &= L \setminus \{x\} \cup \mathrm{regs}(e) \\
[\![x := M(e)]\!]^\# L &= L \setminus \{x\} \cup \mathrm{regs}(e) \\
[\![M(e_1) := M(e_2)]\!]^\# L &= L \cup \mathrm{regs}(e_1) \cup \mathrm{regs}(e_2)
\end{aligned}
$$

Note: distributive.

Live registers (syntactic approximation)

Register x is live at u ($x \in L[u]$), iff there is a path $u \xrightarrow{\pi} v$, $v \in V_{\mathrm{end}}$, such that

- π does not contain writes to x, and $x \in X$
- or π contains a read of x before the first write to x

Abstract effects, propagating live variables backwards over edge

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^\# L &= L \\
[\![\mathrm{Pos}(e)]\!]^\# L &= L \cup \mathrm{regs}(e) \\
[\![\mathrm{Neg}(e)]\!]^\# L &= L \cup \mathrm{regs}(e) \\
[\![x := e]\!]^\# L &= L \setminus \{x\} \cup \mathrm{regs}(e) \\
[\![x := M(e)]\!]^\# L &= L \setminus \{x\} \cup \mathrm{regs}(e) \\
[\![M(e_1) := M(e_2)]\!]^\# L &= L \cup \mathrm{regs}(e_1) \cup \mathrm{regs}(e_2)
\end{aligned}
$$

Note: distributive.

Lift to path (backwards!): $[\![k_1 \dots k_n]\!]^\# := [\![k_1]\!]^\# \circ \dots \circ [\![k_n]\!]^\#$

Live registers (syntactic approximation)

Register x is live at u ($x \in L[u]$), iff there is a path $u \xrightarrow{\pi} v$, $v \in V_{\mathrm{end}}$, such that

- π does not contain writes to x, and $x \in X$
- or π contains a read of x before the first write to x

Abstract effects, propagating live variables backwards over edge

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#} L &= L \\
[\![\mathrm{Pos}(e)]\!]^{\#} L &= L \cup \mathrm{regs}(e) \\
[\![\mathrm{Neg}(e)]\!]^{\#} L &= L \cup \mathrm{regs}(e) \\
[\![x := e]\!]^{\#} L &= L \setminus \{x\} \cup \mathrm{regs}(e) \\
[\![x := M(e)]\!]^{\#} L &= L \setminus \{x\} \cup \mathrm{regs}(e) \\
[\![M(e_1) := M(e_2)]\!]^{\#} L &= L \cup \mathrm{regs}(e_1) \cup \mathrm{regs}(e_2)
\end{aligned}
$$

Note: distributive.

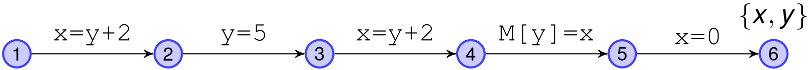
Lift to path (backwards!): $[\![k_1 \dots k_n]\!]^{\#} := [\![k_1]\!]^{\#} \circ \dots \circ [\![k_n]\!]^{\#}$

Live at u (MOP): $L[u] = \bigcup \{ [\![\pi]\!]^{\#} X \mid \exists v \in V_{\mathrm{end}}.\ u \xrightarrow{\pi} v \}$

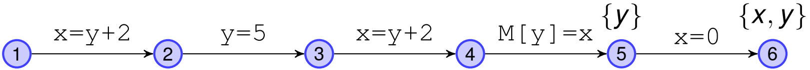
Example

$$1 \xrightarrow{\texttt{x=y+2}} 2 \xrightarrow{\texttt{y=5}} 3 \xrightarrow{\texttt{x=y+2}} 4 \xrightarrow{\texttt{M[y]=x}} 5 \xrightarrow{\texttt{x=0}} 6$$

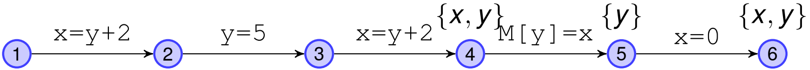
Example



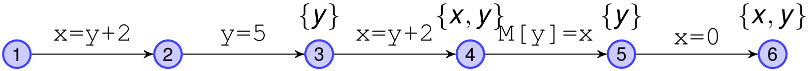
Example



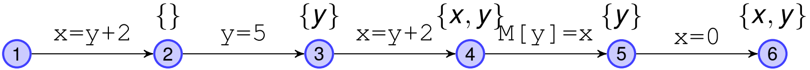
Example



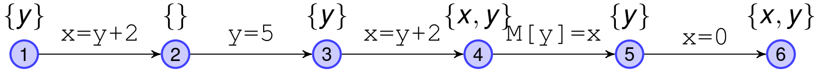
Example



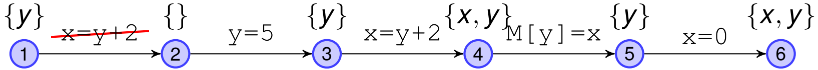
Example



Example



Example



Liveness: Correct approximation

Theorem

(Syntactic) liveness is a correct approximation of semantic liveness
$\mathit{Live}[u] \subseteq L[u]$

Liveness: Correct approximation

Theorem

(Syntactic) liveness is a correct approximation of semantic liveness
$\mathit{Live}[u] \subseteq L[u]$

- Proof: On whiteboard.

Computing L

Use constraint system

$$L[u] \supseteq X \qquad \text{for } u \in V_{\text{end}}$$

$$L[u] \supseteq [\![k]\!]^{\#} L[v] \qquad \text{for edges } k = (u, a, v)$$

Computing L

Use constraint system

$$L[u] \supseteq X \qquad \text{for } u \in V_{\text{end}}$$
$$L[u] \supseteq [\![k]\!]^{\#} L[v] \qquad \text{for edges } k = (u, a, v)$$

Information propagated backwards

Computing L

Use constraint system

$$L[u] \supseteq X \qquad \text{for } u \in V_{\text{end}}$$
$$L[u] \supseteq [\![k]\!]^{\#} L[v] \qquad \text{for edges } k = (u, a, v)$$

Information propagated backwards

Domain: $(\mathrm{Reg}, \subseteq)$

- Reg: The finitely many registers occurring in program.
  $\Longrightarrow$ Finite height
- Moreover, the $[\![k]\!]^{\#}$ are distributive

Computing L

Use constraint system

$$\begin{aligned} L[u] &\supseteq X && \text{for } u \in V_{\mathrm{end}} \\ L[u] &\supseteq [\![k]\!]^{\#} L[v] && \text{for edges } k = (u, a, v) \end{aligned}$$

Information propagated backwards

Domain: $(\mathrm{Reg}, \subseteq)$

- Reg: The finitely many registers occurring in program.
  $\Longrightarrow$ Finite height
- Moreover, the $[\![k]\!]^{\#}$ are distributive

Can compute least solution (MFP)

- Worklist algo, RR-iteration, naive fp-iteration

Backwards Analysis Framework

Given CFG, Domain: $(\mathbb{D}, \sqsubseteq)$, init. value: $d_0 \in \mathbb{D}$, abstract effects: $[\![\cdot]\!]^\# : \mathbb{D} \to \mathbb{D}$, monotonic

$\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \exists v \in V_{\mathrm{end}}.\ u \xrightarrow{\pi} v\}$

MFP is least solution of

$$\begin{aligned} \mathrm{MFP}[u] &\sqsupseteq d_0 && \text{for } u \in V_{\mathrm{end}} \\ \mathrm{MFP}[u] &\sqsupseteq [\![k]\!]^\# \mathrm{MFP}[v] && \text{for edges } k = (u, a, v) \end{aligned}$$

Backwards Analysis Framework

Given CFG, Domain: $(\mathbb{D}, \sqsubseteq)$, init. value: $d_0 \in \mathbb{D}$, abstract effects: $[\![\cdot]\!]^\# : \mathbb{D} \to \mathbb{D}$, monotonic

$\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \exists v \in V_{\mathrm{end}}.\ u \xrightarrow{\pi} v\}$
MFP is least solution of

$$\begin{aligned} \mathrm{MFP}[u] &\sqsupseteq d_0 && \text{for } u \in V_{\mathrm{end}} \\ \mathrm{MFP}[u] &\sqsupseteq [\![k]\!]^\# \mathrm{MFP}[v] && \text{for edges } k = (u, a, v) \end{aligned}$$

- We have:

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

Backwards Analysis Framework

Given CFG, Domain: $(\mathbb{D}, \sqsubseteq)$, init. value: $d_0 \in \mathbb{D}$, abstract effects: $[\![\cdot]\!]^\# : \mathbb{D} \to \mathbb{D}$, monotonic

$\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \exists v \in V_{\mathrm{end}}.\ u \xrightarrow{\pi} v\}$
MFP is least solution of

$$\begin{aligned} \mathrm{MFP}[u] &\sqsupseteq d_0 && \text{for } u \in V_{\mathrm{end}} \\ \mathrm{MFP}[u] &\sqsupseteq [\![k]\!]^\# \mathrm{MFP}[v] && \text{for edges } k = (u, a, v) \end{aligned}$$

- We have:

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

- If the $[\![k]\!]^\#$ are distributive, and from every node an end node can be reached:

$$\mathrm{MOP} = \mathrm{MFP}$$

Backwards Analysis Framework

Given CFG, Domain: $(\mathbb{D}, \sqsubseteq)$, init. value: $d_0 \in \mathbb{D}$, abstract effects: $[\![\cdot]\!]^\# : \mathbb{D} \to \mathbb{D}$, monotonic

$\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \exists v \in V_{\mathrm{end}}.\ u \xrightarrow{\pi} v\}$
MFP is least solution of

$$\begin{aligned} \mathrm{MFP}[u] &\sqsupseteq d_0 && \text{for } u \in V_{\mathrm{end}} \\ \mathrm{MFP}[u] &\sqsupseteq [\![k]\!]^\# \mathrm{MFP}[v] && \text{for edges } k = (u, a, v) \end{aligned}$$

- We have:

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

- If the $[\![k]\!]^\#$ are distributive, and from every node an end node can be reached:

$$\mathrm{MOP} = \mathrm{MFP}$$

- Proofs:

Backwards Analysis Framework

Given CFG, Domain: $(\mathbb{D}, \sqsubseteq)$, init. value: $d_0 \in \mathbb{D}$, abstract effects: $[\![\cdot]\!]^\# : \mathbb{D} \to \mathbb{D}$, monotonic

$\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\# d_0 \mid \exists v \in V_{\mathrm{end}}.\ u \xrightarrow{\pi} v\}$
MFP is least solution of

$$\begin{aligned} \mathrm{MFP}[u] &\sqsupseteq d_0 && \text{for } u \in V_{\mathrm{end}} \\ \mathrm{MFP}[u] &\sqsupseteq [\![k]\!]^\# \mathrm{MFP}[v] && \text{for edges } k = (u, a, v) \end{aligned}$$

- We have:

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

- If the $[\![k]\!]^\#$ are distributive, and from every node an end node can be reached:

$$\mathrm{MOP} = \mathrm{MFP}$$

- Proofs:
  - Analogously to forward case :) $\square$

Example: Dead Assignment elimination

```
while (x>0) {
  y = y + 1
  x = x + y
  x = 1
}
```

On whiteboard.

Last Lecture

- Monotonic forward/backward framework
- Live variables, dead assignment elimination
  - x live at u
  - Semantically: $x \in \mathrm{Live}[u]$: Exists execution that depends on value of x at u
  - Syntactic approximation: $x \in L[u]$: x read before it is overwritten
  - Correctness proof
    - Induction on path, case distinction over edges

Analysis: Classifications

- Forward vs. backward

Forward	Considers executions reaching a program point
Backwards	Considers executions from program point to end

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point

  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization

  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

  May Something may hold, and thus prevents (correct) optimization

  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point

  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization

  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

  May Something may hold, and thus prevents (correct) optimization

  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

- Kill/Gen analysis
  - Effects have form $[\![k]\!]^{\#} X = X \sqcap \mathrm{kill}_k \sqcup \mathrm{gen}_k$
  - Particular simple class. Distributive by construction.

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point

  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization

  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

  May Something may hold, and thus prevents (correct) optimization

  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

- Kill/Gen analysis
  - Effects have form $[\![k]\!]^\# X = X \sqcap \mathrm{kill}_k \sqcup \mathrm{gen}_k$
  - Particular simple class. Distributive by construction.
  - Bitvector analysis: Kill/Gen on finite set domain.

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point
  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization
  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

  May Something may hold, and thus prevents (correct) optimization
  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

- Kill/Gen analysis
  - Effects have form $[\![k]\!]^{\#} X = X \sqcap \mathrm{kill}_k \sqcup \mathrm{gen}_k$
  - Particular simple class. Distributive by construction.
  - Bitvector analysis: Kill/Gen on finite set domain.
- Examples:

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point

  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization

  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

  May Something may hold, and thus prevents (correct) optimization

  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

- Kill/Gen analysis
  - Effects have form $[\![k]\!]^{\#} X = X \sqcap \mathrm{kill}_k \sqcup \mathrm{gen}_k$
  - Particular simple class. Distributive by construction.
  - Bitvector analysis: Kill/Gen on finite set domain.
- Examples:
  - Available expressions:

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point

  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization

  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

  May Something may hold, and thus prevents (correct) optimization

  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

- Kill/Gen analysis
  - Effects have form $[\![k]\!]^\# X = X \sqcap \mathrm{kill}_k \sqcup \mathrm{gen}_k$
  - Particular simple class. Distributive by construction.
  - Bitvector analysis: Kill/Gen on finite set domain.
- Examples:
  - Available expressions: forward,must,kill-gen

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point

  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization

  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

  May Something may hold, and thus prevents (correct) optimization

  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

- Kill/Gen analysis
  - Effects have form $[\![k]\!]^{\#}X = X \sqcap \mathrm{kill}_k \sqcup \mathrm{gen}_k$
  - Particular simple class. Distributive by construction.
  - Bitvector analysis: Kill/Gen on finite set domain.
- Examples:
  - Available expressions: forward,must,kill-gen
  - Live variables:

Analysis: Classifications

- Forward vs. backward

  Forward Considers executions reaching a program point

  Backwards Considers executions from program point to end

- Must vs. May

  Must Something is guaranteed to hold, and thus allows optimization

  - On set domain: $\sqsubseteq = \supseteq$, i.e. $\sqcup = \cap$

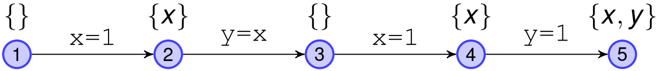
  May Something may hold, and thus prevents (correct) optimization

  - On set domain: $\sqsubseteq = \subseteq$, i.e. $\sqcup = \cup$

- Kill/Gen analysis
  - Effects have form $[\![k]\!]^{\#} X = X \sqcap \mathrm{kill}_k \sqcup \mathrm{gen}_k$
  - Particular simple class. Distributive by construction.
  - Bitvector analysis: Kill/Gen on finite set domain.
- Examples:
  - Available expressions: forward,must,kill-gen
  - Live variables: backward,may,kill-gen

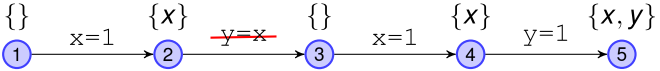
Dead Assignment Elimination: Problems

Eliminating dead assignments may lead to new dead assignments



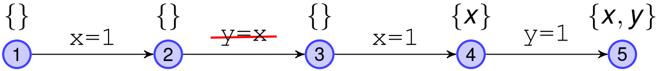
Dead Assignment Elimination: Problems

Eliminating dead assignments may lead to new dead assignments



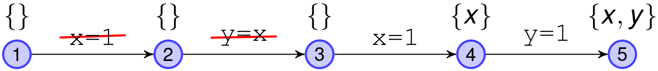
Dead Assignment Elimination: Problems

Eliminating dead assignments may lead to new dead assignments



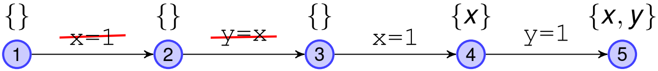
Dead Assignment Elimination: Problems

Eliminating dead assignments may lead to new dead assignments

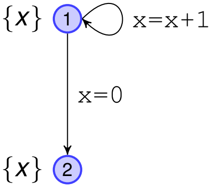


Dead Assignment Elimination: Problems

Eliminating dead assignments may lead to new dead assignments



In a loop, a variable may keep itself alive

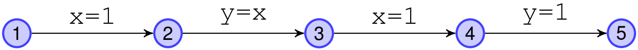


Truly live registers

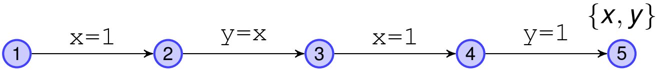
Idea: Consider assignment edge $(u, x = e, v)$.

- If x is not semantically live at v, the registers in e need not become live at u
- There values influence a register that is dead anyway.

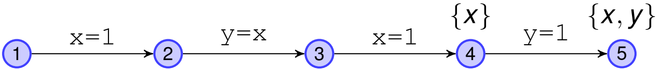
Example



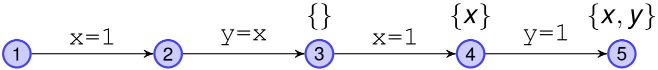
Example



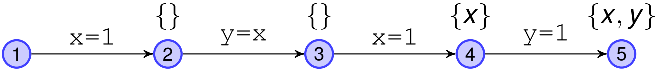
Example



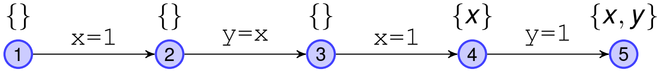
Example



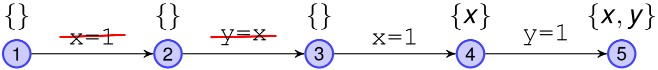
Example



Example



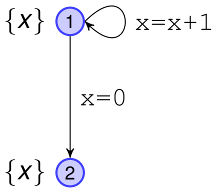
Example



True Liveness vs. repeated liveness

- True liveness detects more dead variables than repeated liveness

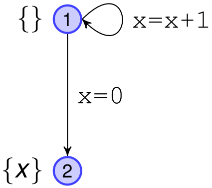
Repeated liveness:



True Liveness vs. repeated liveness

- True liveness detects more dead variables than repeated liveness

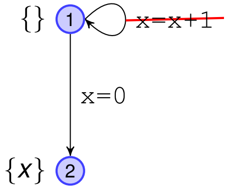
True liveness:



True Liveness vs. repeated liveness

- True liveness detects more dead variables than repeated liveness

True liveness:



Live registers: Abstract effects

$$
\begin{array}{rll}
[\![\mathrm{Nop}]\!]^\# & L = & L \\
[\![\mathrm{Pos}(e)]\!]^\# & L = & L \cup \mathrm{regs}(e) \\
[\![\mathrm{Neg}(e)]\!]^\# & L = & L \cup \mathrm{regs}(e) \\
[\![x := e]\!]^\# & L = & L \setminus \{x\} \cup (\qquad\qquad \mathrm{regs}(e) \quad) \\
[\![x := M(e)]\!]^\# & L = & L \setminus \{x\} \cup (\qquad\qquad \mathrm{regs}(e) \quad) \\
[\![M(e_1) := e_2]\!]^\# & L = & L \cup \mathrm{regs}(e_1) \cup \mathrm{regs}(e_2)
\end{array}
$$

Truly live registers: Abstract effects

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#}\, TL &= TL \\
[\![\mathrm{Pos}(e)]\!]^{\#}\, TL &= TL \cup \mathrm{regs}(e) \\
[\![\mathrm{Neg}(e)]\!]^{\#}\, TL &= TL \cup \mathrm{regs}(e) \\
[\![x := e]\!]^{\#}\, TL &= TL \setminus \{x\} \cup (x \in TL ? \mathrm{regs}(e) : \emptyset) \\
[\![x := M(e)]\!]^{\#}\, TL &= TL \setminus \{x\} \cup (x \in TL ? \mathrm{regs}(e) : \emptyset) \\
[\![M(e_1) := e_2]\!]^{\#}\, TL &= TL \cup \mathrm{regs}(e_1) \cup \mathrm{regs}(e_2)
\end{aligned}
$$

Truly live registers: Abstract effects

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^\# TL &= TL \\
[\![\mathrm{Pos}(e)]\!]^\# TL &= TL \cup \mathrm{regs}(e) \\
[\![\mathrm{Neg}(e)]\!]^\# TL &= TL \cup \mathrm{regs}(e) \\
[\![x := e]\!]^\# TL &= TL \setminus \{x\} \cup (x \in TL ? \mathrm{regs}(e) : \emptyset) \\
[\![x := M(e)]\!]^\# TL &= TL \setminus \{x\} \cup (x \in TL ? \mathrm{regs}(e) : \emptyset) \\
[\![M(e_1) := e_2]\!]^\# TL &= TL \cup \mathrm{regs}(e_1) \cup \mathrm{regs}(e_2)
\end{aligned}
$$

Effects are more complicated. No kill/gen, but still distributive.

Truly live registers: Abstract effects

$$
\begin{aligned}
[\![\text{Nop}]\!]^\# \mathit{TL} &= \mathit{TL} \\
[\![\text{Pos}(e)]\!]^\# \mathit{TL} &= \mathit{TL} \cup \text{regs}(e) \\
[\![\text{Neg}(e)]\!]^\# \mathit{TL} &= \mathit{TL} \cup \text{regs}(e) \\
[\![x := e]\!]^\# \mathit{TL} &= \mathit{TL} \setminus \{x\} \cup (x \in \mathit{TL} ? \text{regs}(e) : \emptyset) \\
[\![x := M(e)]\!]^\# \mathit{TL} &= \mathit{TL} \setminus \{x\} \cup (x \in \mathit{TL} ? \text{regs}(e) : \emptyset) \\
[\![M(e_1) := e_2]\!]^\# \mathit{TL} &= \mathit{TL} \cup \text{regs}(e_1) \cup \text{regs}(e_2)
\end{aligned}
$$

Effects are more complicated. No kill/gen, but still distributive.

We have $\text{MFP} = \text{MOP}$:)

True Liveness: Correct approximation

Theorem

True liveness is a correct approximation of semantic liveness $\mathit{Live}[u] \subseteq TL[u]$

True Liveness: Correct approximation

Theorem

True liveness is a correct approximation of semantic liveness $\mathit{Live}[u] \subseteq TL[u]$

- Proof: On whiteboard.

Table of Contents

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination
- In many cases, we can, instead, replace r_1 by r_2 in subsequent code

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination
- In many cases, we can, instead, replace r_1 by r_2 in subsequent code

$\Longrightarrow$ r_1 becomes dead, and assignment can be eliminated

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination
- In many cases, we can, instead, replace r_1 by r_2 in subsequent code

$\Longrightarrow$ r_1 becomes dead, and assignment can be eliminated

`r`$_1$`=`$\boldsymbol{T_e}$`; M[0] = r`$_1$` + 3`

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination
- In many cases, we can, instead, replace r_1 by r_2 in subsequent code

$\Longrightarrow$ r_1 becomes dead, and assignment can be eliminated

`r`$_1$`=`$\boldsymbol{T_e}$`; M[0] = `$\boldsymbol{T_e}$` + 3`

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination
- In many cases, we can, instead, replace r_1 by r_2 in subsequent code

$\Longrightarrow$ r_1 becomes dead, and assignment can be eliminated

`Nop; M[0] =` T_e `+ 3`

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination
- In many cases, we can, instead, replace r_1 by r_2 in subsequent code

$\Longrightarrow$ r_1 becomes dead, and assignment can be eliminated

Analysis: Maintain an acyclic graph between registers

- Edge $x \to y$ implies $\rho(x) = \rho(y)$ for every state reachable at u
- Assignment $x = y$ creates edge $x \to y$.

Copy propagation

Idea: Often have assignments of form $r_1 = r_2$.

- E.g., $R = T_e$ after redundancy elimination
- In many cases, we can, instead, replace r_1 by r_2 in subsequent code

$\Longrightarrow$ r_1 becomes dead, and assignment can be eliminated

Analysis: Maintain an acyclic graph between registers

- Edge $x \to y$ implies $\rho(x) = \rho(y)$ for every state reachable at u
- Assignment $x = y$ creates edge $x \to y$.

Transformation: Replace variables in expressions according to graph

Example

On Whiteboard

Abstract Effects

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#} C &= C \\
[\![\mathrm{Pos}(e)]\!]^{\#} C &= C \\
[\![\mathrm{Neg}(e)]\!]^{\#} C &= C \\
[\![x = y]\!]^{\#} C &= C \setminus \{x \to *, * \to x\} \cup \{x \to y\} && \text{for } y \in \mathrm{Reg}, y \neq x \\
[\![x = e]\!]^{\#} C &= C \setminus \{x \to *, * \to x\} && \text{for } e \in \mathrm{Expr} \setminus \mathrm{Reg} \text{ or } e = x \\
[\![x = M[e]]\!]^{\#} C &= C \setminus \{x \to *, * \to x\} \\
[\![M[e_1] = e_2]\!]^{\#} C &= C
\end{aligned}
$$

where $\{x \to *, * \to x\}$ is the set of edges from/to x

Obviously, abstract effects preserve acyclicity of C

Moreover, out-degree of nodes is ≤ 1

Abstract effects are distributive

Last Lecture

- Classification of analysis
  - Forward vs. backward, must vs. may, kill/gen, bitvector
- Truly live variables
  - Better approximation of „semantically life”
  - Idea: Don’t care about values of variables that only affect dead variables anyway.
- Copy propagation
  - Replace registers by registers with equal value, to create dead assignments
- Whole procedure: Simple redundancy elimination, then CP and DAE to clean up

Analysis Framework

- Domain: $(\mathbb{D} = 2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
  - I.e.: More precise means more edges (Safe approximation: less edges)
  - Join: $\cap$ (Must analysis)
  - Forward analysis, initial value $d_0 = \emptyset$

$\Longrightarrow \quad \mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^{\#}\emptyset \mid v_0 \xrightarrow{\pi} u\}$

Analysis Framework

- Domain: $(\mathbb{D} = 2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
  - I.e.: More precise means more edges (Safe approximation: less edges)
  - Join: $\cap$ (Must analysis)
  - Forward analysis, initial value $d_0 = \emptyset$

$\Longrightarrow$ $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^{\#}\emptyset \mid v_0 \xrightarrow{\pi} u\}$

- Correctness: $x \to y \in \mathrm{MOP}[u] \implies \forall(\rho,\mu) \in [\![u]\!].\ \rho(x) = \rho(y)$
  - Justifies correctness of transformation wrt. MOP
  - Proof: Later!

Analysis Framework

- Domain: $(\mathbb{D} = 2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
  - I.e.: More precise means more edges (Safe approximation: less edges)
  - Join: $\cap$ (Must analysis)
  - Forward analysis, initial value $d_0 = \emptyset$

$\Longrightarrow$ $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^{\#}\emptyset \mid v_0 \xrightarrow{\pi} u\}$

- Correctness: $x \to y \in \mathrm{MOP}[u] \implies \forall(\rho,\mu) \in [\![u]\!].\ \rho(x) = \rho(y)$
  - Justifies correctness of transformation wrt. MOP
  - Proof: Later!
- Note: Formally, domain contains all graphs.

Analysis Framework

- Domain: $(\mathbb{D} = 2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
  - I.e.: More precise means more edges (Safe approximation: less edges)
  - Join: $\cap$ (Must analysis)
  - Forward analysis, initial value $d_0 = \emptyset$

$\Longrightarrow$ $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^{\#}\emptyset \mid v_0 \xrightarrow{\pi} u\}$

- Correctness: $x \to y \in \mathrm{MOP}[u] \implies \forall(\rho,\mu) \in [\![u]\!].\ \rho(x) = \rho(y)$
  - Justifies correctness of transformation wrt. MOP
  - Proof: Later!
- Note: Formally, domain contains all graphs.
  - Required for complete lattice property!

Analysis Framework

- Domain: $(\mathbb{D} = 2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
  - I.e.: More precise means more edges (Safe approximation: less edges)
  - Join: $\cap$ (Must analysis)
  - Forward analysis, initial value $d_0 = \emptyset$

$\implies$ $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^\# \emptyset \mid v_0 \xrightarrow{\pi} u\}$

- Correctness: $x \to y \in \mathrm{MOP}[u] \implies \forall(\rho, \mu) \in [\![u]\!].\ \rho(x) = \rho(y)$
  - Justifies correctness of transformation wrt. MOP
  - Proof: Later!
- Note: Formally, domain contains all graphs.
  - Required for complete lattice property!
  - But not suited for implementation (Set of all pairs of registers)

Analysis Framework

- Domain: $(\mathbb{D} = 2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
  - I.e.: More precise means more edges (Safe approximation: less edges)
  - Join: $\cap$ (Must analysis)
  - Forward analysis, initial value $d_0 = \emptyset$

$\implies$ $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^{\#}\emptyset \mid v_0 \xrightarrow{\pi} u\}$

- Correctness: $x \to y \in \mathrm{MOP}[u] \implies \forall(\rho,\mu) \in [\![u]\!].\ \rho(x) = \rho(y)$
  - Justifies correctness of transformation wrt. MOP
  - Proof: Later!
- Note: Formally, domain contains all graphs.
  - Required for complete lattice property!
  - But not suited for implementation (Set of all pairs of registers)
  - Add $\bot$-element to domain. $[\![k]\!]^{\#}\bot := \bot$.

Analysis Framework

- Domain: $(\mathbb{D} = 2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
  - I.e.: More precise means more edges (Safe approximation: less edges)
  - Join: $\cap$ (Must analysis)
  - Forward analysis, initial value $d_0 = \emptyset$

$\Longrightarrow$ $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^\#\emptyset \mid v_0 \xrightarrow{\pi} u\}$

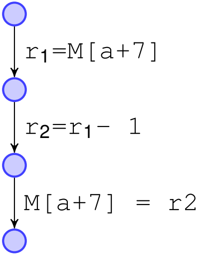
- Correctness: $x \to y \in \mathrm{MOP}[u] \implies \forall(\rho,\mu) \in [\![u]\!].\ \rho(x) = \rho(y)$
  - Justifies correctness of transformation wrt. MOP
  - Proof: Later!
- Note: Formally, domain contains all graphs.
  - Required for complete lattice property!
  - But not suited for implementation (Set of all pairs of registers)
  - Add $\bot$-element to domain. $[\![k]\!]^\#\bot := \bot$.
  - Intuition: $\bot$ means unreachable.

Table of Contents

Procedure as a whole

1. Simple redundancy elimination
   - Replaces re-computation by memorization
   - Inserts superfluous moves
2. Copy propagation
   - Removes superfluous moves
   - Creates dead assignments
3. Dead assignment elimination

Example: $a[7] - -$



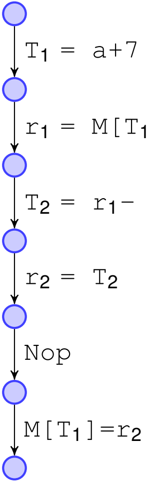
Example: $a[7] - -$

Introduced memorization registers

```
T1 = a+7
r1 = M[T1]
T2 = r1- 1
r2 = T2
T1 = a+7
M[T1]=r2
```

Example: $a[7]--$

Eliminated redundant computations



Example: $a[7]--$

Copy propagation done

$\mathtt{T}_1 = \mathtt{a+7}$

$\mathtt{r}_1 = \mathtt{M[T}_1\mathtt{]}$

$\mathtt{T}_2 = \mathtt{r}_1\mathtt{-\ 1}$

$\mathtt{r}_2 = \mathtt{T}_2$

Nop

$\mathtt{M[T}_1\mathtt{]=T}_2$

Example: $a[7]--$

Eliminated dead assignments

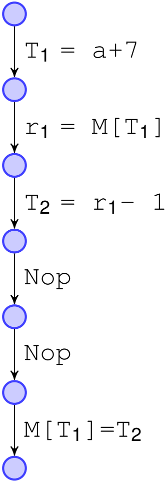


Table of Contents

Background: Simulation

- Given:

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$
  - Concrete effects $[\![a]\!] : \mathbb{C} \to \mathbb{C}$, abstract effects $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
    - With forward-generalization to paths: $[\![k_1 \dots k_n]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$ and $[\![k_1 \dots k_n]\!]^\# = [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$
  - Concrete effects $[\![a]\!] : \mathbb{C} \to \mathbb{C}$, abstract effects $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
    - With forward-generalization to paths: $[\![k_1 \dots k_n]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$ and $[\![k_1 \dots k_n]\!]^\# = [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$
  - Relation $\Delta \subseteq \mathbb{C} \times \mathbb{D}$

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$
  - Concrete effects $[\![a]\!] : \mathbb{C} \to \mathbb{C}$, abstract effects $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
    - With forward-generalization to paths: $[\![k_1 \dots k_n]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$ and $[\![k_1 \dots k_n]\!]^\# = [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$
  - Relation $\Delta \subseteq \mathbb{C} \times \mathbb{D}$
- Assume:

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$
  - Concrete effects $[\![a]\!] : \mathbb{C} \to \mathbb{C}$, abstract effects $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
    - With forward-generalization to paths: $[\![k_1 \dots k_n]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$ and $[\![k_1 \dots k_n]\!]^\# = [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$
  - Relation $\Delta \subseteq \mathbb{C} \times \mathbb{D}$
- Assume:
  - Initial values in relation: $c_0 \; \Delta \; d_0$

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$
  - Concrete effects $[\![a]\!] : \mathbb{C} \to \mathbb{C}$, abstract effects $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
    - With forward-generalization to paths: $[\![k_1 \dots k_n]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$ and $[\![k_1 \dots k_n]\!]^\# = [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$
  - Relation $\Delta \subseteq \mathbb{C} \times \mathbb{D}$
- Assume:
  - Initial values in relation: $c_0 \;\Delta\; d_0$
  - Relation preserved by effects: $c \;\Delta\; d \implies [\![k]\!]c \;\Delta\; [\![k]\!]^\# d$

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$
  - Concrete effects $[\![a]\!] : \mathbb{C} \to \mathbb{C}$, abstract effects $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
    - With forward-generalization to paths: $[\![k_1 \dots k_n]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$ and $[\![k_1 \dots k_n]\!]^\# = [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$
  - Relation $\Delta \subseteq \mathbb{C} \times \mathbb{D}$
- Assume:
  - Initial values in relation: $c_0 \; \Delta \; d_0$
  - Relation preserved by effects: $c \; \Delta \; d \implies [\![k]\!]c \; \Delta \; [\![k]\!]^\# d$
- Get: Relation preserved by paths from initial values: $[\![\pi]\!]c_0 \; \Delta \; [\![\pi]\!]^\# d_0$

Background: Simulation

- Given:
  - Concrete values $\mathbb{C}$, abstract values $\mathbb{D}$, actions A
  - Initial values $c_0 \in \mathbb{C}$, $d_0 \in \mathbb{D}$
  - Concrete effects $[\![a]\!] : \mathbb{C} \to \mathbb{C}$, abstract effects $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
    - With forward-generalization to paths: $[\![k_1 \dots k_n]\!] = [\![k_n]\!] \circ \dots \circ [\![k_1]\!]$ and $[\![k_1 \dots k_n]\!]^\# = [\![k_n]\!]^\# \circ \dots \circ [\![k_1]\!]^\#$
  - Relation $\Delta \subseteq \mathbb{C} \times \mathbb{D}$
- Assume:
  - Initial values in relation: $c_0 \; \Delta \; d_0$
  - Relation preserved by effects: $c \; \Delta \; d \implies [\![k]\!]c \; \Delta \; [\![k]\!]^\# d$
- Get: Relation preserved by paths from initial values: $[\![\pi]\!]c_0 \; \Delta \; [\![\pi]\!]^\# d_0$
- Proof: Straightforward induction on paths. On whiteboard!

Background: Description relation

- Now: $c \mathrel{\Delta} d$ — Concrete value c described by abstract value d

Background: Description relation

- Now: $c \; \Delta \; d$ — Concrete value c described by abstract value d
- Moreover, assume complete lattices on $\mathbb{C}$ and $\mathbb{D}$.
  - Intuition: $x \sqsubseteq x'$ — x is more precise than x'

Background: Description relation

- Now: $c \mathrel{\Delta} d$ — Concrete value c described by abstract value d
- Moreover, assume complete lattices on $\mathbb{C}$ and $\mathbb{D}$.
  - Intuition: $x \sqsubseteq x'$ — x is more precise than x'
- Assume Δ to be monotonic on abstract values:

$$c \mathrel{\Delta} d \wedge d \sqsubseteq d' \implies c \mathrel{\Delta} d'$$

  - Intuition: Less precise abstract value still describes concrete value

Background: Description relation

- Now: $c \mathrel{\Delta} d$ — Concrete value c described by abstract value d
- Moreover, assume complete lattices on $\mathbb{C}$ and $\mathbb{D}$.
  - Intuition: $x \sqsubseteq x'$ — x is more precise than x'
- Assume Δ to be monotonic on abstract values:

$$c \mathrel{\Delta} d \wedge d \sqsubseteq d' \implies c \mathrel{\Delta} d'$$

  - Intuition: Less precise abstract value still describes concrete value
- Assume Δ to be distributive on concrete values:

$$(\forall c \in C.\ c \mathrel{\Delta} d) \iff (\bigsqcup C) \mathrel{\Delta} d$$

  - Note: Implies anti-monotonicity: $c' \sqsubseteq c \wedge c \mathrel{\Delta} d \implies c' \mathrel{\Delta} d$
  - Intuition: More precise concrete values still described by abstract value

Background: Description relation

- Now: $c \,\Delta\, d$ — Concrete value c described by abstract value d
- Moreover, assume complete lattices on $\mathbb{C}$ and $\mathbb{D}$.
  - Intuition: $x \sqsubseteq x'$ — x is more precise than x'
- Assume Δ to be monotonic on abstract values:

$$c \,\Delta\, d \wedge d \sqsubseteq d' \implies c \,\Delta\, d'$$

  - Intuition: Less precise abstract value still describes concrete value
- Assume Δ to be distributive on concrete values:

$$(\forall c \in C.\, c \,\Delta\, d) \iff (\bigsqcup C) \,\Delta\, d$$

  - Note: Implies anti-monotonicity: $c' \sqsubseteq c \wedge c \,\Delta\, d \implies c' \,\Delta\, d$
  - Intuition: More precise concrete values still described by abstract value
- We get for all sets of paths P:

$$(\forall \pi \in P.\, [\![\pi]\!] c_0 \,\Delta\, [\![\pi]\!]^{\#} d_0) \implies (\bigsqcup_{\pi \in P} [\![\pi]\!] c_0) \,\Delta\, (\bigsqcup_{\pi \in P} [\![\pi]\!]^{\#} d_0)$$

  - Intuition: Concrete values due to paths P described by abstract values

Application to Program Analysis

- Concrete values: Sets of states with $\subseteq$
  - Intuition: Less states = more precise information

Application to Program Analysis

- Concrete values: Sets of states with $\subseteq$
  - Intuition: Less states = more precise information
- Concrete effects: Effects of edges (generalized to sets of states)
  - $[\![k]\!]C := \bigcup_{(\rho,\mu)\in C\cap\mathrm{dom}[\![k]\!]} [\![k]\!](\rho,\mu)$, i.e., don't include undefined effects

Application to Program Analysis

- Concrete values: Sets of states with $\subseteq$
  - Intuition: Less states = more precise information
- Concrete effects: Effects of edges (generalized to sets of states)
  - $[\![k]\!]C := \bigcup_{(\rho,\mu)\in C\cap \mathrm{dom}[\![k]\!]} [\![k]\!](\rho,\mu)$, i.e., don't include undefined effects
- Concrete initial values: All states: $c_0 = \mathrm{State}$

Application to Program Analysis

- Concrete values: Sets of states with $\subseteq$
  - Intuition: Less states = more precise information
- Concrete effects: Effects of edges (generalized to sets of states)
  - $[\![k]\!]C := \bigcup_{(\rho,\mu)\in C\cap\mathrm{dom}[\![k]\!]} [\![k]\!](\rho,\mu)$, i.e., don't include undefined effects
- Concrete initial values: All states: $c_0 = \mathrm{State}$
- Abstract values: Domain of analysis, abstract effects: $[\![k]\!]^{\#}$, d_0

Application to Program Analysis

- Concrete values: Sets of states with $\subseteq$
  - Intuition: Less states = more precise information
- Concrete effects: Effects of edges (generalized to sets of states)
  - $[\![k]\!]C := \bigcup_{(\rho,\mu)\in C\cap\mathrm{dom}[\![k]\!]} [\![k]\!](\rho,\mu)$, i.e., don't include undefined effects
- Concrete initial values: All states: $c_0 = \mathrm{State}$
- Abstract values: Domain of analysis, abstract effects: $[\![k]\!]^\#$, d_0
- Description relation: States described by abstract value
  - Usually: Define Δ on single states, and lift to set of states:

$$S \;\Delta\; A \text{ iff } \forall(\rho,\mu) \in S.\ (\rho,\mu) \;\Delta\; A$$

  - This guarantees distributivity in concrete states

Application to Program Analysis

- Concrete values: Sets of states with $\subseteq$
  - Intuition: Less states = more precise information
- Concrete effects: Effects of edges (generalized to sets of states)
  - $[\![k]\!]C := \bigcup_{(\rho,\mu)\in C\cap \mathrm{dom}[\![k]\!]} [\![k]\!](\rho,\mu)$, i.e., don't include undefined effects
- Concrete initial values: All states: $c_0 = \mathrm{State}$
- Abstract values: Domain of analysis, abstract effects: $[\![k]\!]^\#$, d_0
- Description relation: States described by abstract value
  - Usually: Define Δ on single states, and lift to set of states:

$$S \;\Delta\; A \text{ iff } \forall(\rho,\mu)\in S.\ (\rho,\mu) \;\Delta\; A$$

  - This guarantees distributivity in concrete states
- We get: $[\![u]\!] \;\Delta\; \mathrm{MOP}[u]$
  - All states reachable at u described by analysis result at u.

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\mathrm{Expr}}, \supseteq)$

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\mathrm{Expr}}, \supseteq)$
- Define: $(\rho, \mu) \,\Delta\, A$ iff $\forall e \in A.\ [\![e]\!]\rho = \rho(T_e)$

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\text{Expr}}, \supseteq)$
- Define: $(\rho, \mu) \, \Delta \, A$ iff $\forall e \in A. \, [\![e]\!]\rho = \rho(T_e)$
- Prove: $A \supseteq A' \wedge (\rho, \mu) \, \Delta \, A \implies (\rho, \mu) \, \Delta \, A'$

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\mathrm{Expr}}, \supseteq)$
- Define: $(\rho, \mu) \;\Delta\; A$ iff $\forall e \in A.\ [\![e]\!]\rho = \rho(T_e)$
- Prove: $A \supseteq A' \wedge (\rho, \mu) \;\Delta\; A \implies (\rho, \mu) \;\Delta\; A'$
- Prove: $(\rho, \mu) \;\Delta\; A \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, A)]\!](\rho, \mu)$
  - where $\begin{aligned} \mathrm{tr}(T_e = e, A) &= \text{if } e \in A \text{ then } \mathrm{Nop} \text{ else } T_e = e \mid \\ \mathrm{tr}(a, A) &= a \end{aligned}$

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\text{Expr}}, \supseteq)$
- Define: $(\rho, \mu)\ \Delta\ A$ iff $\forall e \in A.\ [\![e]\!]\rho = \rho(T_e)$
- Prove: $A \supseteq A' \wedge (\rho, \mu)\ \Delta\ A \implies (\rho, \mu)\ \Delta\ A'$
- Prove: $(\rho, \mu)\ \Delta\ A \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, A)]\!](\rho, \mu)$
  - where $\begin{array}{rl} \mathrm{tr}(T_e = e, A) & = \text{if } e \in A \text{ then } \mathrm{Nop} \text{ else } T_e = e \mid \\ \mathrm{tr}(a, A) & = a \end{array}$
  - Transformation in CFG: $(u, a, v) \mapsto (u, \mathrm{tr}(a, A[u]), v)$

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\mathrm{Expr}}, \supseteq)$
- Define: $(\rho, \mu) \Delta A$ iff $\forall e \in A.\ [\![e]\!]\rho = \rho(T_e)$
- Prove: $A \supseteq A' \wedge (\rho, \mu) \Delta A \implies (\rho, \mu) \Delta A'$
- Prove: $(\rho, \mu) \Delta A \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, A)]\!](\rho, \mu)$
  - where $\begin{aligned} \mathrm{tr}(T_e = e, A) &= \text{if } e \in A \text{ then } \mathrm{Nop} \text{ else } T_e = e \mid \\ \mathrm{tr}(a, A) &= a \end{aligned}$
  - Transformation in CFG: $(u, a, v) \mapsto (u, \mathrm{tr}(a, A[u]), v)$
- Prove: $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \Delta d_0$
  - For AE, we have $d_0 = \emptyset$, which implies the above.

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\mathrm{Expr}}, \supseteq)$
- Define: $(\rho, \mu) \;\Delta\; A$ iff $\forall e \in A.\ [\![e]\!]\rho = \rho(T_e)$
- Prove: $A \supseteq A' \wedge (\rho, \mu) \;\Delta\; A \implies (\rho, \mu) \;\Delta\; A'$
- Prove: $(\rho, \mu) \;\Delta\; A \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, A)]\!](\rho, \mu)$
  - where $\begin{aligned} \mathrm{tr}(T_e = e, A) &= \text{if } e \in A \text{ then } \mathrm{Nop} \text{ else } T_e = e \mid \\ \mathrm{tr}(a, A) &= a \end{aligned}$
  - Transformation in CFG: $(u, a, v) \mapsto (u, \mathrm{tr}(a, A[u]), v)$
- Prove: $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \;\Delta\; d_0$
  - For AE, we have $d_0 = \emptyset$, which implies the above.
- Prove: $(\rho, \mu) \in \mathrm{dom}[\![k]\!] \wedge (\rho, \mu) \;\Delta\; D \implies [\![k]\!](\rho, \mu) \;\Delta\; [\![k]\!]^{\#} D$

Example: Available expressions

- Recall: $\mathbb{D} = (2^{\mathrm{Expr}}, \supseteq)$
- Define: $(\rho, \mu) \,\Delta\, A$ iff $\forall e \in A.\ [\![e]\!]\rho = \rho(T_e)$
- Prove: $A \supseteq A' \wedge (\rho, \mu) \,\Delta\, A \implies (\rho, \mu) \,\Delta\, A'$
- Prove: $(\rho, \mu) \,\Delta\, A \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, A)]\!](\rho, \mu)$
  - where $\begin{aligned} \mathrm{tr}(T_e = e, A) &= \text{if } e \in A \text{ then } \mathrm{Nop} \text{ else } T_e = e \mid \\ \mathrm{tr}(a, A) &= a \end{aligned}$
  - Transformation in CFG: $(u, a, v) \mapsto (u, \mathrm{tr}(a, A[u]), v)$
- Prove: $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \,\Delta\, d_0$
  - For AE, we have $d_0 = \emptyset$, which implies the above.
- Prove: $(\rho, \mu) \in \mathrm{dom}[\![k]\!] \wedge (\rho, \mu) \,\Delta\, D \implies [\![k]\!](\rho, \mu) \,\Delta\, [\![k]\!]^{\#} D$
- Get: $[\![u]\!] \,\Delta\, \mathrm{MOP}[u]$, thus $[\![u]\!] \,\Delta\, \mathrm{MFP}[u]$
  - Which justifies correctness of transformation wrt. MFP

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\text{Reg}\times\text{Reg}}, \supseteq)$

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\text{Reg} \times \text{Reg}}, \supseteq)$
- $(\rho, \mu) \ \Delta \ C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg} \times \mathrm{Reg}}, \supseteq)$
- $(\rho, \mu) \, \Delta \, C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Monotonic for abstract values.

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
- $(\rho, \mu) \mathrel{\Delta} C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - tr(a, C): Replace variables in expressions due to edges in C

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg} \times \mathrm{Reg}}, \supseteq)$
- $(\rho, \mu) \, \Delta \, C$ iff $\forall (x \to y) \in C. \, \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - $\mathrm{tr}(a, C)$: Replace variables in expressions due to edges in C
  - $(\rho, \mu) \, \Delta \, C \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, C)]\!](\rho, \mu)$
    - Replace variables by equal variables

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg} \times \mathrm{Reg}}, \supseteq)$
- $(\rho, \mu) \, \Delta \, C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - $\mathrm{tr}(a, C)$: Replace variables in expressions due to edges in C
  - $(\rho, \mu) \, \Delta \, C \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, C)]\!](\rho, \mu)$
    - Replace variables by equal variables
- $d_0 = \emptyset$. Obviously $(\rho_0, \mu_0) \, \Delta \, \emptyset$ for all ρ_0, μ_0.

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
- $(\rho, \mu)\ \Delta\ C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - $\mathrm{tr}(a, C)$: Replace variables in expressions due to edges in C
  - $(\rho, \mu)\ \Delta\ C \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, C)]\!](\rho, \mu)$
    - Replace variables by equal variables
- $d_0 = \emptyset$. Obviously $(\rho_0, \mu_0)\ \Delta\ \emptyset$ for all ρ_0, μ_0.
- Show $(\rho, \mu) \in \mathrm{dom}[\![k]\!] \wedge (\rho, \mu)\ \Delta\ C \implies [\![k]\!](\rho, \mu)\ \Delta\ [\![k]\!]^{\#} C$

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
- $(\rho, \mu)\ \Delta\ C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - $\mathrm{tr}(a, C)$: Replace variables in expressions due to edges in C
  - $(\rho, \mu)\ \Delta\ C \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, C)]\!](\rho, \mu)$
    - Replace variables by equal variables
- $d_0 = \emptyset$. Obviously $(\rho_0, \mu_0)\ \Delta\ \emptyset$ for all ρ_0, μ_0.
- Show $(\rho, \mu) \in \mathrm{dom}[\![k]\!] \wedge (\rho, \mu)\ \Delta\ C \implies [\![k]\!](\rho, \mu)\ \Delta\ [\![k]\!]^{\#} C$
  - Assume (IH) $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg} \times \mathrm{Reg}}, \supseteq)$
- $(\rho, \mu) \; \Delta \; C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - $\mathrm{tr}(a, C)$: Replace variables in expressions due to edges in C
  - $(\rho, \mu) \; \Delta \; C \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, C)]\!](\rho, \mu)$
    - Replace variables by equal variables
- $d_0 = \emptyset$. Obviously $(\rho_0, \mu_0) \; \Delta \; \emptyset$ for all ρ_0, μ_0.
- Show $(\rho, \mu) \in \mathrm{dom}[\![k]\!] \wedge (\rho, \mu) \; \Delta \; C \implies [\![k]\!](\rho, \mu) \; \Delta \; [\![k]\!]^{\#} C$
  - Assume (IH) $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Assume (1) $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and (2) $x \to y \in [\![k]\!]^{\#} C$

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg}\times\mathrm{Reg}}, \supseteq)$
- $(\rho, \mu) \; \Delta \; C$ iff $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - $\mathrm{tr}(a, C)$: Replace variables in expressions due to edges in C
  - $(\rho, \mu) \; \Delta \; C \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, C)]\!](\rho, \mu)$
    - Replace variables by equal variables
- $d_0 = \emptyset$. Obviously $(\rho_0, \mu_0) \; \Delta \; \emptyset$ for all ρ_0, μ_0.
- Show $(\rho, \mu) \in \mathrm{dom}[\![k]\!] \wedge (\rho, \mu) \; \Delta \; C \implies [\![k]\!](\rho, \mu) \; \Delta \; [\![k]\!]^{\#} C$
  - Assume (IH) $\forall (x \to y) \in C.\ \rho(x) = \rho(y)$
  - Assume (1) $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and (2) $x \to y \in [\![k]\!]^{\#} C$
  - Show $\rho'(x) = \rho'(y)$

Example: Copy propagation

- $(\mathbb{D}, \sqsubseteq) = (2^{\mathrm{Reg} \times \mathrm{Reg}}, \supseteq)$
- $(\rho, \mu) \; \Delta \; C$ iff $\forall (x \to y) \in C. \; \rho(x) = \rho(y)$
  - Monotonic for abstract values.
  - $\mathrm{tr}(a, C)$: Replace variables in expressions due to edges in C
  - $(\rho, \mu) \; \Delta \; C \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, C)]\!](\rho, \mu)$
    - Replace variables by equal variables
- $d_0 = \emptyset$. Obviously $(\rho_0, \mu_0) \; \Delta \; \emptyset$ for all ρ_0, μ_0.
- Show $(\rho, \mu) \in \mathrm{dom}[\![k]\!] \wedge (\rho, \mu) \; \Delta \; C \implies [\![k]\!](\rho, \mu) \; \Delta \; [\![k]\!]^{\#} C$
  - Assume (IH) $\forall (x \to y) \in C. \; \rho(x) = \rho(y)$
  - Assume (1) $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and (2) $x \to y \in [\![k]\!]^{\#} C$
  - Show $\rho'(x) = \rho'(y)$
  - By case distinction on k. On whiteboard.

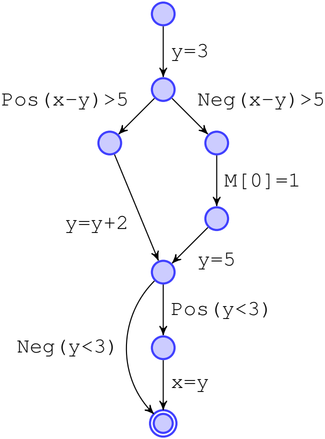
Table of Contents

Constant Propagation: Idea

- Compute constant values at compile time
- Eliminate unreachable code

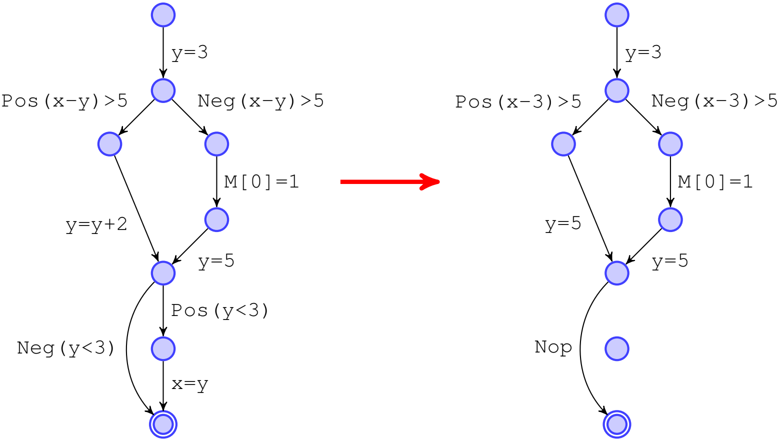
Constant Propagation: Idea

- Compute constant values at compile time
- Eliminate unreachable code



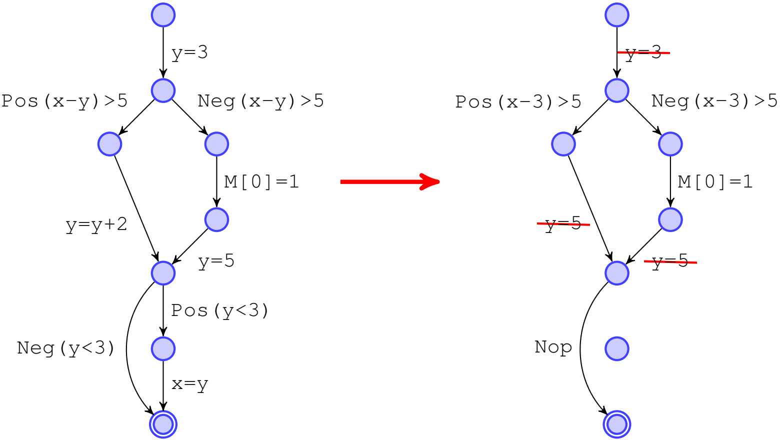
Constant Propagation: Idea

- Compute constant values at compile time
- Eliminate unreachable code



Constant Propagation: Idea

- Compute constant values at compile time
- Eliminate unreachable code



- Dead-code elimination afterwards to clean up (assume *y* not interesting)

Approach

- Idea: Store, for each register, whether it is definitely constant at u

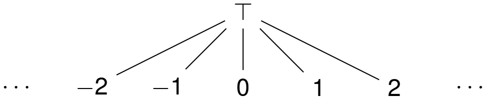
Approach

- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^\top$

$$
\begin{tikzcd}
 & & & \top & & & \\
\cdots & -2 \arrow[urr, no head] & -1 \arrow[ur, no head] & 0 \arrow[u, no head] & 1 \arrow[ul, no head] & 2 \arrow[ull, no head] & \cdots
\end{tikzcd}
$$

Approach

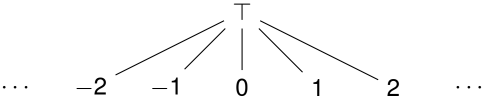
- Idea: Store, for each register, whether it is definitely constant at *u*
  - Assign each register a value from $\mathbb{Z}^{\top}$



    - Intuition: $\top$ — don't know value of register

Approach

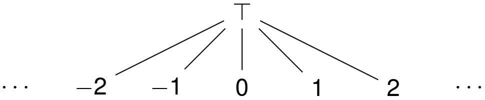
- Idea: Store, for each register, whether it is definitely constant at *u*
  - Assign each register a value from $\mathbb{Z}^\top$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^\top)$

Approach

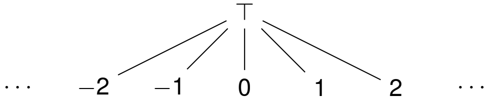
- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^\top$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^\top) \cup \{\bot\}$
  - Add a bottom-element

Approach

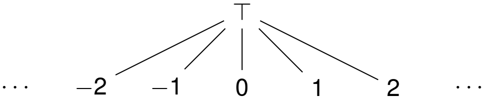
- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^{\top}$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^{\top}) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable

Approach

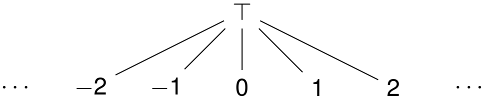
- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^{\top}$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^{\top}) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable
  - Ordering: Pointwise ordering on functions, $\bot$ being the least element.

Approach

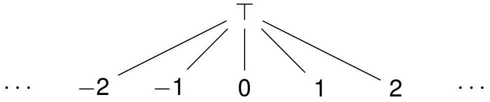
- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^{\top}$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^{\top}) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable
  - Ordering: Pointwise ordering on functions, $\bot$ being the least element.
  - $(\mathbb{D}, \sqsubseteq)$ is complete lattice

Approach

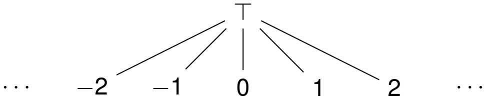
- Idea: Store, for each register, whether it is definitely constant at *u*
  - Assign each register a value from $\mathbb{Z}^{\top}$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^{\top}) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable
  - Ordering: Pointwise ordering on functions, $\bot$ being the least element.
  - $(\mathbb{D}, \sqsubseteq)$ is complete lattice
- Examples

Approach

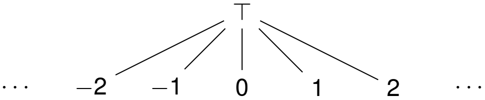
- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^{\top}$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^{\top}) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable
  - Ordering: Pointwise ordering on functions, $\bot$ being the least element.
  - $(\mathbb{D}, \sqsubseteq)$ is complete lattice
- Examples
  - $D[u] = \bot$:

Approach

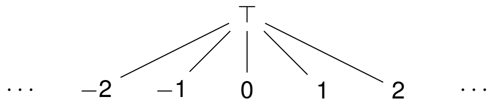
- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^\top$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \rightarrow \mathbb{Z}^\top) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable
  - Ordering: Pointwise ordering on functions, $\bot$ being the least element.
  - $(\mathbb{D}, \sqsubseteq)$ is complete lattice
- Examples
  - $D[u] = \bot$: u not reachable

Approach

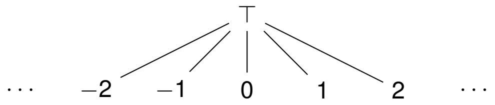
- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^\top$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^\top) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable
  - Ordering: Pointwise ordering on functions, $\bot$ being the least element.
  - $(\mathbb{D}, \sqsubseteq)$ is complete lattice
- Examples
  - $D[u] = \bot$: u not reachable
  - $D[u] = \{x \mapsto \top, y \mapsto 5\}$:

Approach

- Idea: Store, for each register, whether it is definitely constant at u
  - Assign each register a value from $\mathbb{Z}^\top$



    - Intuition: $\top$ — don't know value of register
  - $\mathbb{D} = (\mathrm{Reg} \to \mathbb{Z}^\top) \cup \{\bot\}$
  - Add a bottom-element
    - Intuition: $\bot$ — program point not reachable
  - Ordering: Pointwise ordering on functions, $\bot$ being the least element.
  - $(\mathbb{D}, \sqsubseteq)$ is complete lattice
- Examples
  - $D[u] = \bot$: u not reachable
  - $D[u] = \{x \mapsto \top, y \mapsto 5\}$: y is always 5 at u, nothing known about x

Abstract evaluation of expressions

- For concrete operator $\Box : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, we define abstract operator $\Box^\# : \mathbb{Z}^\top \times \mathbb{Z}^\top \to \mathbb{Z}^\top$:

$$\top \,\Box^\#\, x := \top$$
$$x \,\Box^\#\, \top := \top$$
$$x \,\Box^\#\, y := x \,\Box\, y$$

Abstract evaluation of expressions

- For concrete operator $\square : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, we define abstract operator $\square^\# : \mathbb{Z}^\top \times \mathbb{Z}^\top \to \mathbb{Z}^\top$:

$$\top \,\square^\#\, x := \top$$
$$x \,\square^\#\, \top := \top$$
$$x \,\square^\#\, y := x \,\square\, y$$

- Evaluate expression wrt. abstract values and operators: $[\![e]\!]^\# : (\mathrm{Reg} \to \mathbb{Z}^\top) \to \mathbb{Z}^\top$

$[\![c]\!]^\# D := c$	for constant c
$[\![r]\!]^\# D := D(r)$	for register r
$[\![e_1 \,\square\, e_2]\!]^\# D := [\![e_1]\!]^\# D \,\square^\#\, [\![e_2]\!]^\# D$	for operator $\square$

Abstract evaluation of expressions

- For concrete operator $\Box : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, we define abstract operator $\Box^{\#} : \mathbb{Z}^{\top} \times \mathbb{Z}^{\top} \to \mathbb{Z}^{\top}$:

$$\begin{aligned} \top \,\Box^{\#}\, x &:= \top \\ x \,\Box^{\#}\, \top &:= \top \\ x \,\Box^{\#}\, y &:= x \,\Box\, y \end{aligned}$$

- Evaluate expression wrt. abstract values and operators: $[\![e]\!]^{\#} : (\mathrm{Reg} \to \mathbb{Z}^{\top}) \to \mathbb{Z}^{\top}$

$$\begin{aligned} [\![c]\!]^{\#} D &:= c && \text{for constant } c \\ [\![r]\!]^{\#} D &:= D(r) && \text{for register } r \\ [\![e_1 \,\Box\, e_2]\!]^{\#} D &:= [\![e_1]\!]^{\#} D \,\Box^{\#}\, [\![e_2]\!]^{\#} D && \text{for operator } \Box \end{aligned}$$

Analogously for unary, ternary, etc. operators

Example

- Example: $D = \{x \mapsto \top, y \mapsto 5\}$

Example

- Example: $D = \{x \mapsto \top, y \mapsto 5\}$

$$\begin{aligned}[\![y - 3]\!]^\# D &= [\![y]\!]^\# D -^\# [\![3]\!]^\# D \\ &= 5 -^\# 3 \\ &= 2\end{aligned}$$

Example

- Example: $D = \{x \mapsto \top, y \mapsto 5\}$

$$\begin{aligned} [\![y - 3]\!]^{\#} D &= [\![y]\!]^{\#} D -^{\#} [\![3]\!]^{\#} D \\ &= 5 -^{\#} 3 \\ &= 2 \end{aligned}$$

$$\begin{aligned} [\![x + y]\!]^{\#} D &= [\![x]\!]^{\#} D +^{\#} [\![y]\!]^{\#} D \\ &= \top +^{\#} 5 \\ &= \top \end{aligned}$$

Abstract effects (forward)

$$\llbracket k \rrbracket^\# \bot := \bot \qquad \text{for any edge } k$$

$$\llbracket \mathrm{Nop} \rrbracket^\# D := D$$

$$\llbracket \mathrm{Pos}(e) \rrbracket^\# := \begin{cases} \bot & \text{if } \llbracket e \rrbracket^\# D = 0 \\ D & \text{otherwise} \end{cases}$$

$$\llbracket \mathrm{Neg}(e) \rrbracket^\# := \begin{cases} \bot & \text{if } \llbracket e \rrbracket^\# D = v, v \in \mathbb{Z} \setminus \{0\} \\ D & \text{otherwise} \end{cases}$$

$$\llbracket r = e \rrbracket^\# D := D(r \mapsto \llbracket e \rrbracket^\# D)$$

$$\llbracket r = M[e] \rrbracket^\# D := D(r \mapsto \top)$$

$$\llbracket M[e_1] = e_2 \rrbracket^\# D := D$$

For $D \neq \bot$.

Abstract effects (forward)

$$
\begin{aligned}
[\![k]\!]^\# \bot &:= \bot && \text{for any edge } k \\
[\![\mathrm{Nop}]\!]^\# D &:= D \\
[\![\mathrm{Pos}(e)]\!]^\# &:= \begin{cases} \bot & \text{if } [\![e]\!]^\# D = 0 \\ D & \text{otherwise} \end{cases} \\
[\![\mathrm{Neg}(e)]\!]^\# &:= \begin{cases} \bot & \text{if } [\![e]\!]^\# D = v, v \in \mathbb{Z} \setminus \{0\} \\ D & \text{otherwise} \end{cases} \\
[\![r = e]\!]^\# D &:= D(r \mapsto [\![e]\!]^\# D) \\
[\![r = M[e]]\!]^\# D &:= D(r \mapsto \top) \\
[\![M[e_1] = e_2]\!]^\# D &:= D
\end{aligned}
$$

For $D \neq \bot$.

Initial value at start: $d_0 := \lambda x.\ \top$.

Abstract effects (forward)

$$
\begin{aligned}
[\![k]\!]^\# \bot &:= \bot && \text{for any edge } k \\
[\![\mathrm{Nop}]\!]^\# D &:= D \\
[\![\mathrm{Pos}(e)]\!]^\# &:= \begin{cases} \bot & \text{if } [\![e]\!]^\# D = 0 \\ D & \text{otherwise} \end{cases} \\
[\![\mathrm{Neg}(e)]\!]^\# &:= \begin{cases} \bot & \text{if } [\![e]\!]^\# D = v, v \in \mathbb{Z} \setminus \{0\} \\ D & \text{otherwise} \end{cases} \\
[\![r = e]\!]^\# D &:= D(r \mapsto [\![e]\!]^\# D) \\
[\![r = M[e]]\!]^\# D &:= D(r \mapsto \top) \\
[\![M[e_1] = e_2]\!]^\# D &:= D
\end{aligned}
$$

For $D \neq \bot$.

Initial value at start: $d_0 := \lambda x.\ \top$.

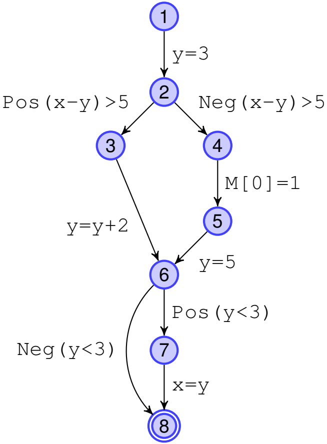
(Reachable, all variables have unknown value)

Last lecture

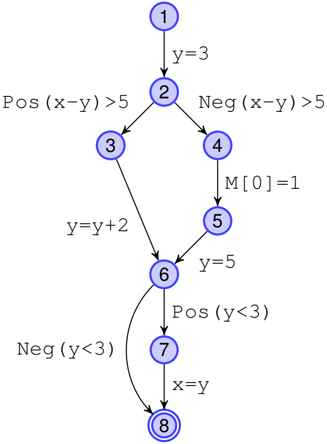
- Simulation based framework for program analysis
- Abstract setting:
  - Actions preserve relation Δ between concrete and abstract state.

    $\Longrightarrow$ States after executing path are related
  - Approximation: Complete lattice structure
    - Δ monotonic
    - Distributive $\Longrightarrow$ generalization to sets of path
- For program analysis:
  - Concrete state: Sets of program states
    - All states reachable via path.
- Constant propagation

Example

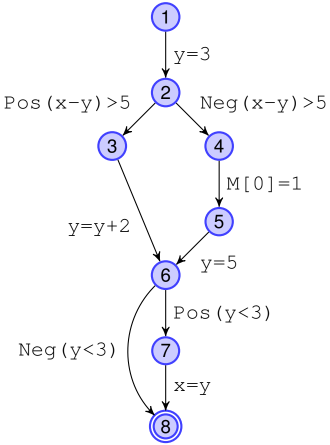


Example



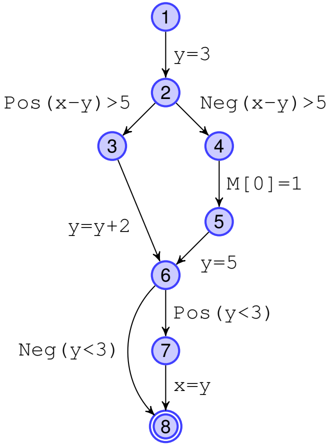
$$\mathrm{D}[1] \;=\; x \mapsto \top, y \mapsto \top$$

Example



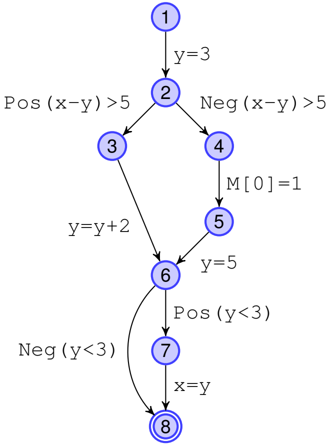
$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3
\end{aligned}
$$

Example



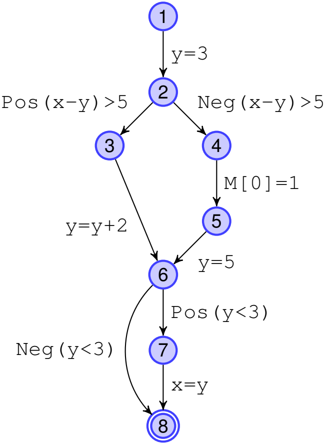
$$\begin{aligned} D[1] &= x \mapsto \top, y \mapsto \top \\ D[2] &= x \mapsto \top, y \mapsto 3 \\ D[3] &= x \mapsto \top, y \mapsto 3 \end{aligned}$$

Example



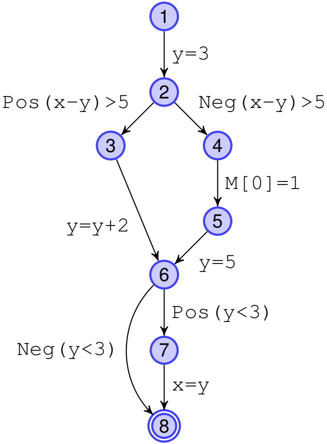
$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3
\end{aligned}
$$

Example



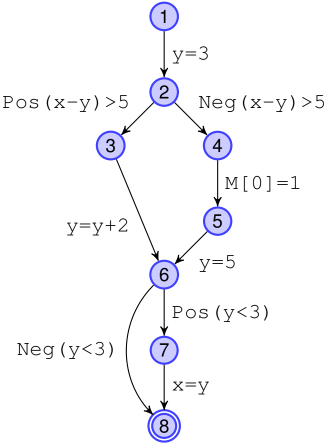
$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3 \\
D[5] &= x \mapsto \top, y \mapsto 3
\end{aligned}
$$

Example



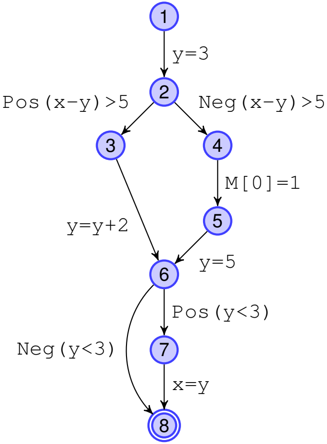
$$\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3 \\
D[5] &= x \mapsto \top, y \mapsto 3 \\
D[6] &= x \mapsto \top, y \mapsto 5
\end{aligned}$$

Example



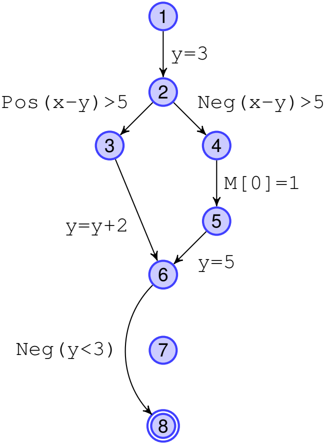
$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3 \\
D[5] &= x \mapsto \top, y \mapsto 3 \\
D[6] &= x \mapsto \top, y \mapsto 5 \\
D[7] &= \bot
\end{aligned}
$$

Example



$$\begin{array}{rcl}
D[1] &=& x \mapsto \top, y \mapsto \top \\
D[2] &=& x \mapsto \top, y \mapsto 3 \\
D[3] &=& x \mapsto \top, y \mapsto 3 \\
D[4] &=& x \mapsto \top, y \mapsto 3 \\
D[5] &=& x \mapsto \top, y \mapsto 3 \\
D[6] &=& x \mapsto \top, y \mapsto 5 \\
D[7] &=& \bot \\
D[8] &=& x \mapsto \top, y \mapsto 5
\end{array}$$

Example

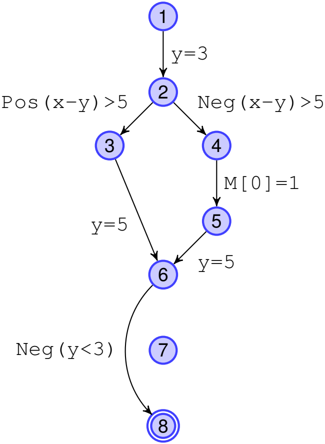


$$
\begin{array}{rcl}
D[1] & = & x \mapsto \top, y \mapsto \top \\
D[2] & = & x \mapsto \top, y \mapsto 3 \\
D[3] & = & x \mapsto \top, y \mapsto 3 \\
D[4] & = & x \mapsto \top, y \mapsto 3 \\
D[5] & = & x \mapsto \top, y \mapsto 3 \\
D[6] & = & x \mapsto \top, y \mapsto 5 \\
D[7] & = & \bot \\
D[8] & = & x \mapsto \top, y \mapsto 5
\end{array}
$$

Transformations:

Remove (u, a, v) if $D[u] = \bot$ or $D[v] = \bot$

Example



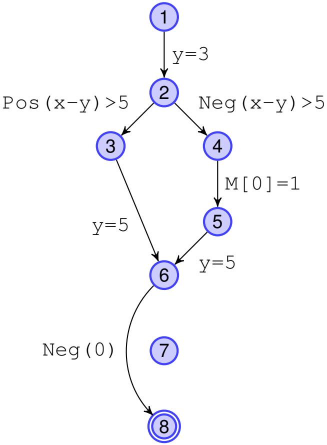
$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3 \\
D[5] &= x \mapsto \top, y \mapsto 3 \\
D[6] &= x \mapsto \top, y \mapsto 5 \\
D[7] &= \perp \\
D[8] &= x \mapsto \top, y \mapsto 5
\end{aligned}
$$

Transformations:

Remove (u, a, v) if $D[u] = \perp$ or $D[v] = \perp$

$(u, r = e, v) \mapsto (u, r = c, v)$ if $[\![e]\!]^{\#}(D[u]) = c \in \mathbb{Z}$

Example



$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3 \\
D[5] &= x \mapsto \top, y \mapsto 3 \\
D[6] &= x \mapsto \top, y \mapsto 5 \\
D[7] &= \perp \\
D[8] &= x \mapsto \top, y \mapsto 5
\end{aligned}
$$

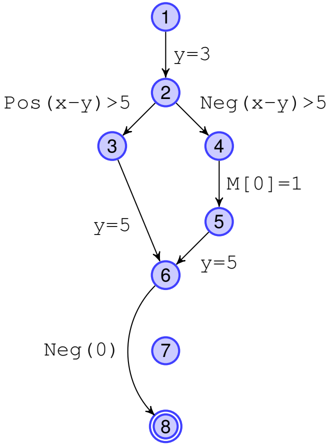
Transformations:

Remove (u, a, v) if $D[u] = \perp$ or $D[v] = \perp$

$(u, r = e, v) \mapsto (u, r = c, v)$ if $[\![e]\!]^\#(D[u]) = c \in \mathbb{Z}$

Analogously for test, load, store

Example



$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3 \\
D[5] &= x \mapsto \top, y \mapsto 3 \\
D[6] &= x \mapsto \top, y \mapsto 5 \\
D[7] &= \bot \\
D[8] &= x \mapsto \top, y \mapsto 5
\end{aligned}
$$

Transformations:

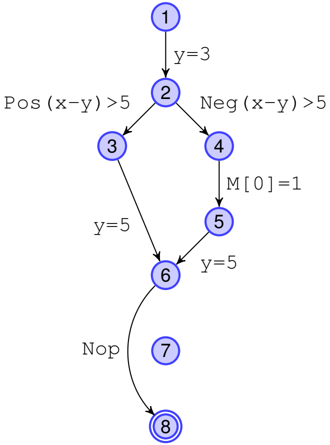
Remove (u, a, v) if $D[u] = \bot$ or $D[v] = \bot$

$(u, r = e, v) \mapsto (u, r = c, v)$ if $[\![e]\!]^\#(D[u]) = c \in \mathbb{Z}$

Analogously for test, load, store

$(u, \mathrm{Pos}(c), v) \mapsto \mathrm{Nop}$ if $c \in \mathbb{Z} \setminus \{0\}$

Example



$$
\begin{aligned}
D[1] &= x \mapsto \top, y \mapsto \top \\
D[2] &= x \mapsto \top, y \mapsto 3 \\
D[3] &= x \mapsto \top, y \mapsto 3 \\
D[4] &= x \mapsto \top, y \mapsto 3 \\
D[5] &= x \mapsto \top, y \mapsto 3 \\
D[6] &= x \mapsto \top, y \mapsto 5 \\
D[7] &= \bot \\
D[8] &= x \mapsto \top, y \mapsto 5
\end{aligned}
$$

Transformations:

Remove (u, a, v) if $D[u] = \bot$ or $D[v] = \bot$

$(u, r = e, v) \mapsto (u, r = c, v)$ if $[\![e]\!]^{\#}(D[u]) = c \in \mathbb{Z}$

Analogously for test, load, store

$(u, \mathrm{Pos}(c), v) \mapsto \mathrm{Nop}$ if $c \in \mathbb{Z} \setminus \{0\}$

$(u, \mathrm{Neg}(0), v) \mapsto \mathrm{Nop}$

Correctness (Description Relation)

- Establish description relation
  - Between values, valuations, states

Correctness (Description Relation)

- Establish description relation
  - Between values, valuations, states
- Values: for $v \in \mathbb{Z}$: $v \mathrel{\Delta} v$ and $v \mathrel{\Delta} \top$
  - Value described by same value, all values described by $\top$

Correctness (Description Relation)

- Establish description relation
  - Between values, valuations, states
- Values: for $v \in \mathbb{Z}$: $v \; \Delta \; v$ and $v \; \Delta \; \top$
  - Value described by same value, all values described by $\top$
  - Note: Monotonic, i.e. $v \; \Delta \; d \wedge d \sqsubseteq d' \implies v \; \Delta \; d'$
    - Only cases: $d = d'$ or $d' = \top$ (flat ordering).

Correctness (Description Relation)

- Establish description relation
  - Between values, valuations, states
- Values: for $v \in \mathbb{Z}$: $v \ \Delta \ v$ and $v \ \Delta \ \top$
  - Value described by same value, all values described by $\top$
  - Note: Monotonic, i.e. $v \ \Delta \ d \wedge d \sqsubseteq d' \implies v \ \Delta \ d'$
    - Only cases: $d = d'$ or $d' = \top$ (flat ordering).
- Valuations: For $\rho : \mathrm{Reg} \to \mathbb{Z}, \rho^{\#} : \mathrm{Reg} \to \mathbb{Z}^{\top}$: $\rho \ \Delta \ \rho^{\#}$ iff $\forall x.\ \rho(x) \ \Delta \ \rho^{\#}(x)$
  - Value of each variable must be described.
  - Note: Monotonic. (Same point-wise definition as for $\sqsubseteq$)

Correctness (Description Relation)

- Establish description relation
  - Between values, valuations, states
- Values: for $v \in \mathbb{Z}$: $v \,\Delta\, v$ and $v \,\Delta\, \top$
  - Value described by same value, all values described by $\top$
  - Note: Monotonic, i.e. $v \,\Delta\, d \wedge d \sqsubseteq d' \implies v \,\Delta\, d'$
    - Only cases: $d = d'$ or $d' = \top$ (flat ordering).
- Valuations: For $\rho : \mathrm{Reg} \to \mathbb{Z}, \rho^{\#} : \mathrm{Reg} \to \mathbb{Z}^{\top}$: $\rho \,\Delta\, \rho^{\#}$ iff $\forall x.\ \rho(x) \,\Delta\, \rho^{\#}(x)$
  - Value of each variable must be described.
  - Note: Monotonic. (Same point-wise definition as for $\sqsubseteq$)
- States: $(\rho, \mu) \,\Delta\, \rho^{\#}$ if $\rho \,\Delta\, \rho^{\#}$ and $\forall s.\ \neg(s \,\Delta\, \bot)$
  - Bottom describes no states (i.e., empty set of states)
  - Note: Monotonic. (Only new case: $s \,\Delta\, \bot \wedge \bot \sqsubseteq d \implies s \,\Delta\, d$)

Correctness (Abstract values)

- Show: For every constant c and operator $\Box$, we have

$$c \,\Delta\, c^{\#}$$

$$v_1 \,\Delta\, d_1 \wedge v_2 \,\Delta\, d_2 \implies (v_1 \Box v_2) \,\Delta\, (d_1 \Box^{\#} d_2)$$

Correctness (Abstract values)

- Show: For every constant c and operator $\Box$, we have

$$c \,\Delta\, c^{\#}$$

$$v_1 \,\Delta\, d_1 \wedge v_2 \,\Delta\, d_2 \implies (v_1 \Box v_2) \,\Delta\, (d_1 \Box^{\#} d_2)$$

- We get (by induction on expression)

$$\rho \,\Delta\, \rho^{\#} \implies [\![e]\!]\rho \,\Delta\, [\![e]\!]^{\#} \rho^{\#}$$

Correctness (Abstract values)

- Show: For every constant c and operator $\Box$, we have

$$c \,\Delta\, c^{\#}$$

$$v_1 \,\Delta\, d_1 \wedge v_2 \,\Delta\, d_2 \implies (v_1 \Box v_2) \,\Delta\, (d_1 \Box^{\#} d_2)$$

- We get (by induction on expression)

$$\rho \,\Delta\, \rho^{\#} \implies [\![e]\!]\rho \,\Delta\, [\![e]\!]^{\#} \rho^{\#}$$

- Moreover, show $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \,\Delta\, d_0$
  - Here: $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \,\Delta\, \lambda x.\ \top$

Correctness (Abstract values)

- Show: For every constant c and operator $\Box$, we have

$$c \;\Delta\; c^{\#}$$

$$v_1 \;\Delta\; d_1 \wedge v_2 \;\Delta\; d_2 \implies (v_1 \Box v_2) \;\Delta\; (d_1 \Box^{\#} d_2)$$

- We get (by induction on expression)

$$\rho \;\Delta\; \rho^{\#} \implies [\![e]\!]\rho \;\Delta\; [\![e]\!]^{\#}\rho^{\#}$$

- Moreover, show $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \;\Delta\; d_0$
    - Here: $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \;\Delta\; \lambda x.\ \top$

$\Longleftarrow\ \rho_0 \;\Delta\; \lambda x.\ \top$

Correctness (Abstract values)

- Show: For every constant c and operator $\Box$, we have

$$c \,\Delta\, c^{\#}$$

$$v_1 \,\Delta\, d_1 \wedge v_2 \,\Delta\, d_2 \implies (v_1 \Box v_2) \,\Delta\, (d_1 \Box^{\#} d_2)$$

- We get (by induction on expression)

$$\rho \,\Delta\, \rho^{\#} \implies [\![e]\!]\rho \,\Delta\, [\![e]\!]^{\#}\rho^{\#}$$

- Moreover, show $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \,\Delta\, d_0$
  - Here: $\forall \rho_0, \mu_0.\ (\rho_0, \mu_0) \,\Delta\, \lambda x.\ \top$

$\Longleftarrow$ $\rho_0 \,\Delta\, \lambda x.\ \top$

$\Longleftarrow$ $\forall x.\ \rho_0(x) \,\Delta\, \top$. Holds by definition.

Correctness (Of Transformations)

- Assume $(\rho, \mu)\ \Delta\ \rho^{\#}$. Show $[\![a]\!](\rho, \mu) = [\![\mathsf{tr}(a, \rho^{\#})]\!](\rho, \mu)$

Correctness (Of Transformations)

- Assume $(\rho, \mu) \, \Delta \, \rho^{\#}$. Show $[\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, \rho^{\#})]\!](\rho, \mu)$
  - Remove edge if $\rho^{\#} = \bot$. Trivial.

Correctness (Of Transformations)

- Assume $(\rho, \mu) \,\Delta\, \rho^{\#}$. Show $[\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, \rho^{\#})]\!](\rho, \mu)$
  - Remove edge if $\rho^{\#} = \bot$. Trivial.
  - Replace $r = e$ by $r = [\![e]\!]^{\#} \rho^{\#}$ if $[\![e]\!]^{\#} \rho^{\#} \neq \top$
    - From $\rho \,\Delta\, \rho^{\#} \implies [\![e]\!]\rho \,\Delta\, [\![e]\!]^{\#} \rho^{\#} \implies [\![e]\!]\rho = [\![e]\!]^{\#} \rho^{\#}$

Correctness (Of Transformations)

- Assume $(\rho, \mu) \Delta \rho^\#$. Show $[\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, \rho^\#)]\!](\rho, \mu)$
  - Remove edge if $\rho^\# = \bot$. Trivial.
  - Replace $r = e$ by $r = [\![e]\!]^\# \rho^\#$ if $[\![e]\!]^\# \rho^\# \neq \top$
    - From $\rho \Delta \rho^\# \implies [\![e]\!]\rho \Delta [\![e]\!]^\# \rho^\# \implies [\![e]\!]\rho = [\![e]\!]^\# \rho^\#$
    - Analogously for expressions in load, store, Neg, Pos.

Correctness (Of Transformations)

- Assume $(\rho, \mu) \; \Delta \; \rho^{\#}$. Show $[\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, \rho^{\#})]\!](\rho, \mu)$
  - Remove edge if $\rho^{\#} = \bot$. Trivial.
  - Replace $r = e$ by $r = [\![e]\!]^{\#}\rho^{\#}$ if $[\![e]\!]^{\#}\rho^{\#} \neq \top$
    - From $\rho \; \Delta \; \rho^{\#} \implies [\![e]\!]\rho \; \Delta \; [\![e]\!]^{\#}\rho^{\#} \implies [\![e]\!]\rho = [\![e]\!]^{\#}\rho^{\#}$
    - Analogously for expressions in load, store, Neg, Pos.
  - Replace tests on constants by `Nop`: Obviously correct.
    - Does not depend on analysis result.

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \; \Delta \; C$. Show $(\rho', \mu') \; \Delta \; [\![k]\!]^{\#} C$.

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu)\ \Delta\ C$. Show $(\rho', \mu')\ \Delta\ [\![k]\!]^{\#} C$.
  - By case distinction on k. Assume $\rho^{\#} := C \neq \bot$.
    - Note: We have $\rho\ \Delta\ \rho^{\#}$

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \,\Delta\, C$. Show $(\rho', \mu') \,\Delta\, [\![k]\!]^{\#} C$.
  - By case distinction on k. Assume $\rho^{\#} := C \neq \bot$.
    - Note: We have $\rho \,\Delta\, \rho^{\#}$
  - Case $k = (u, x = e, v)$: To show $\rho(x := [\![e]\!]\rho) \,\Delta\, \rho^{\#}(x := [\![e]\!]^{\#} \rho^{\#})$

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \Delta C$. Show $(\rho', \mu') \Delta [\![k]\!]^{\#} C$.
  - By case distinction on k. Assume $\rho^{\#} := C \neq \bot$.
    - Note: We have $\rho \Delta \rho^{\#}$
  - Case $k = (u, x = e, v)$: To show $\rho(x := [\![e]\!]\rho) \Delta \rho^{\#}(x := [\![e]\!]^{\#}\rho^{\#})$
    $\Longleftarrow$ $[\![e]\!]\rho \Delta [\![e]\!]^{\#}\rho^{\#}$. Already proved.

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \; \Delta \; C$. Show $(\rho', \mu') \; \Delta \; [\![k]\!]^\# C$.
  - By case distinction on k. Assume $\rho^\# := C \neq \bot$.
    - Note: We have $\rho \; \Delta \; \rho^\#$
  - Case $k = (u, x = e, v)$: To show $\rho(x := [\![e]\!]\rho) \; \Delta \; \rho^\#(x := [\![e]\!]^\# \rho^\#)$

    $\Longleftarrow$ $[\![e]\!]\rho \; \Delta \; [\![e]\!]^\# \rho^\#$. Already proved.
  - Case $k = (u, \mathrm{Pos}(e), v)$ and $[\![e]\!]^\# \rho^\# = 0$:

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \; \Delta \; C$. Show $(\rho', \mu') \; \Delta \; [\![k]\!]^{\#} C$.
  - By case distinction on k. Assume $\rho^{\#} := C \neq \bot$.
    - Note: We have $\rho \; \Delta \; \rho^{\#}$
  - Case $k = (u, x = e, v)$: To show $\rho(x := [\![e]\!]\rho) \; \Delta \; \rho^{\#}(x := [\![e]\!]^{\#} \rho^{\#})$

    $\Longleftarrow$ $[\![e]\!]\rho \; \Delta \; [\![e]\!]^{\#} \rho^{\#}$. Already proved.
  - Case $k = (u, \mathrm{Pos}(e), v)$ and $[\![e]\!]^{\#} \rho^{\#} = 0$:
    - From $[\![e]\!]\rho \; \Delta \; [\![e]\!]^{\#} \rho^{\#}$, we have $[\![e]\!]\rho = 0$

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \;\Delta\; C$. Show $(\rho', \mu') \;\Delta\; [\![k]\!]^{\#} C$.
  - By case distinction on k. Assume $\rho^{\#} := C \neq \bot$.
    - Note: We have $\rho \;\Delta\; \rho^{\#}$
  - Case $k = (u, x = e, v)$: To show $\rho(x := [\![e]\!]\rho) \;\Delta\; \rho^{\#}(x := [\![e]\!]^{\#}\rho^{\#})$

    $\Longleftarrow$ $[\![e]\!]\rho \;\Delta\; [\![e]\!]^{\#}\rho^{\#}$. Already proved.
  - Case $k = (u, \mathrm{Pos}(e), v)$ and $[\![e]\!]^{\#}\rho^{\#} = 0$:
    - From $[\![e]\!]\rho \;\Delta\; [\![e]\!]^{\#}\rho^{\#}$, we have $[\![e]\!]\rho = 0$
    - Hence, $[\![\mathrm{Pos}(e)]\!](\rho, \mu) =$ undefined. Contradiction to assumption.

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \ \Delta \ C$. Show $(\rho', \mu') \ \Delta \ [\![k]\!]^\# C$.
  - By case distinction on k. Assume $\rho^\# := C \neq \bot$.
    - Note: We have $\rho \ \Delta \ \rho^\#$
  - Case $k = (u, x = e, v)$: To show $\rho(x := [\![e]\!]\rho) \ \Delta \ \rho^\#(x := [\![e]\!]^\# \rho^\#)$

    $\Longleftarrow$ $[\![e]\!]\rho \ \Delta \ [\![e]\!]^\# \rho^\#$. Already proved.
  - Case $k = (u, \mathrm{Pos}(e), v)$ and $[\![e]\!]^\# \rho^\# = 0$:
    - From $[\![e]\!]\rho \ \Delta \ [\![e]\!]^\# \rho^\#$, we have $[\![e]\!]\rho = 0$
    - Hence, $[\![\mathrm{Pos}(e)]\!](\rho, \mu) =$ undefined. Contradiction to assumption.
  - Other cases: Analogously.

Correctness (Steps)

- Assume $(\rho', \mu') = [\![k]\!](\rho, \mu)$ and $(\rho, \mu) \,\Delta\, C$. Show $(\rho', \mu') \,\Delta\, [\![k]\!]^{\#} C$.
  - By case distinction on k. Assume $\rho^{\#} := C \neq \bot$.
    - Note: We have $\rho \,\Delta\, \rho^{\#}$
  - Case $k = (u, x = e, v)$: To show $\rho(x := [\![e]\!]\rho) \,\Delta\, \rho^{\#}(x := [\![e]\!]^{\#}\rho^{\#})$

    $\Longleftarrow$ $[\![e]\!]\rho \,\Delta\, [\![e]\!]^{\#}\rho^{\#}$. Already proved.
  - Case $k = (u, \mathrm{Pos}(e), v)$ and $[\![e]\!]^{\#}\rho^{\#} = 0$:
    - From $[\![e]\!]\rho \,\Delta\, [\![e]\!]^{\#}\rho^{\#}$, we have $[\![e]\!]\rho = 0$
    - Hence, $[\![\mathrm{Pos}(e)]\!](\rho, \mu) =$ undefined. Contradiction to assumption.
  - Other cases: Analogously.
- Our general theory gives us: $[\![u]\!] \,\Delta\, \mathrm{MFP}[u]$
  - Thus, transformation wrt. MFP is correct.

Constant propagation: Caveat

- Abstract effects are monotonic

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# =$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) =$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) = \{x \mapsto \top, y \mapsto \top\}$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) = \{x \mapsto \top, y \mapsto \top\}$
  - However: $[\![x = x + y]\!]^\#(\rho_1^\#) =$ and $[\![x = x + y]\!]^\#(\rho_2^\#) =$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) = \{x \mapsto \top, y \mapsto \top\}$
  - However: $[\![x = x + y]\!]^\#(\rho_1^\#) = \{x \mapsto 5, y \mapsto 2\}$ and $[\![x = x + y]\!]^\#(\rho_2^\#) =$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) = \{x \mapsto \top, y \mapsto \top\}$
  - However: $[\![x = x + y]\!]^\#(\rho_1^\#) = \{x \mapsto 5, y \mapsto 2\}$ and $[\![x = x + y]\!]^\#(\rho_2^\#) = \{x \mapsto 5, y \mapsto 3\}$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) = \{x \mapsto \top, y \mapsto \top\}$
  - However: $[\![x = x + y]\!]^\#(\rho_1^\#) = \{x \mapsto 5, y \mapsto 2\}$ and $[\![x = x + y]\!]^\#(\rho_2^\#) = \{x \mapsto 5, y \mapsto 3\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\#) \sqcup [\![x = x + y]\!]^\#(\rho_2^\#) =$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) = \{x \mapsto \top, y \mapsto \top\}$
  - However: $[\![x = x + y]\!]^\#(\rho_1^\#) = \{x \mapsto 5, y \mapsto 2\}$ and $[\![x = x + y]\!]^\#(\rho_2^\#) = \{x \mapsto 5, y \mapsto 3\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\#) \sqcup [\![x = x + y]\!]^\#(\rho_2^\#) = \{x \mapsto 5, y \mapsto \top\}$

Constant propagation: Caveat

- Abstract effects are monotonic
- Unfortunately: Not distributive
  - Consider $\rho_1^\# = \{x \mapsto 3, y \mapsto 2\}$ and $\rho_2^\# = \{x \mapsto 2, y \mapsto 3\}$
  - Have: $\rho_1^\# \sqcup \rho_2^\# = \{x \mapsto \top, y \mapsto \top\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\# \sqcup \rho_2^\#) = \{x \mapsto \top, y \mapsto \top\}$
  - However: $[\![x = x + y]\!]^\#(\rho_1^\#) = \{x \mapsto 5, y \mapsto 2\}$ and $[\![x = x + y]\!]^\#(\rho_2^\#) = \{x \mapsto 5, y \mapsto 3\}$
  - I.e.: $[\![x = x + y]\!]^\#(\rho_1^\#) \sqcup [\![x = x + y]\!]^\#(\rho_2^\#) = \{x \mapsto 5, y \mapsto \top\}$
- Thus, MFP only approximation of MOP in general.

Undecidability of MOP

- MFP only approximation of MOP

Undecidability of MOP

- MFP only approximation of MOP
- And there is nothing we can do about :(

Theorem

For constant propagation, it is undecidable whether $MOP[u](x) = \top$.

Undecidability of MOP

- MFP only approximation of MOP
- And there is nothing we can do about :(

Theorem

For constant propagation, it is undecidable whether $\mathit{MOP}[u](x) = \top$.

- Proof: By undecidability of Hilbert's 10th problem

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \ldots, x_n) = 0$$

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \ldots, x_n) = 0$$

- Where p is a polynomial with integer coefficients.
  - E.g. $p(x_1, x_2) = x_1^2 + 2x_1 - x_2^2 + 2$

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \ldots, x_n) = 0$$

- Where p is a polynomial with integer coefficients.
  - E.g. $p(x_1, x_2) = x_1^2 + 2x_1 - x_2^2 + 2$
  - Solution:

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \ldots, x_n) = 0$$

- Where p is a polynomial with integer coefficients.
  - E.g. $p(x_1, x_2) = x_1^2 + 2x_1 - x_2^2 + 2$
  - Solution: (-1,1)

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \ldots, x_n) = 0$$

- Where p is a polynomial with integer coefficients.
  - E.g. $p(x_1, x_2) = x_1^2 + 2x_1 - x_2^2 + 2$
  - Solution: (-1,1)
- Hard problem. E.g. $x^n + y^n = z^n$ for $n > 2$.

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \dots, x_n) = 0$$

- Where p is a polynomial with integer coefficients.
  - E.g. $p(x_1, x_2) = x_1^2 + 2x_1 - x_2^2 + 2$
  - Solution: (-1,1)
- Hard problem. E.g. $x^n + y^n = z^n$ for $n > 2$. (Fermat's last Theorem)

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \dots, x_n) = 0$$

- Where p is a polynomial with integer coefficients.
  - E.g. $p(x_1, x_2) = x_1^2 + 2x_1 - x_2^2 + 2$
  - Solution: (-1,1)
- Hard problem. E.g. $x^n + y^n = z^n$ for $n > 2$. (Fermat's last Theorem)
  - Wiles,Taylor: No solutions.

Hilbert's 10th problem (1900)

- Find an integer solution of a Diophantine equation

$$p(x_1, \ldots, x_n) = 0$$

- Where p is a polynomial with integer coefficients.
  - E.g. $p(x_1, x_2) = x_1^2 + 2x_1 - x_2^2 + 2$
  - Solution: (-1,1)
- Hard problem. E.g. $x^n + y^n = z^n$ for $n > 2$. (Fermat's last Theorem)
  - Wiles,Taylor: No solutions.

Theorem (Matiyasevich, 1970)

(Based on work of David, Putnam, Robinson)
It is undecidable whether a Diophantine equation has an integer solution.

Regard the following program

```
x₁=x₂=...xₙ=0
while (*) { x₁ = x₁ + 1 }
...
while (*) { xₙ = xₙ + 1 }
r=0
if (p(x₁,...,xₙ) == 0) then r=1
u: Nop
```

- For any valuation of the variables, there is a path through the program

Regard the following program

```
x₁=x₂=...xₙ=0
while (*) { x₁ = x₁ + 1 }
...
while (*) { xₙ = xₙ + 1 }
r=0
if (p(x₁,...,xₙ) == 0) then r=1
u: Nop
```

- For any valuation of the variables, there is a path through the program
- For every path, constant propagation computes the values of the x_i

Regard the following program

```
x1=x2=...xn=0
while (*) { x1 = x1 + 1 }
...
while (*) { xn = xn + 1 }
r=0
if (p(x1,...,xn) == 0) then r=1
u: Nop
```

- For any valuation of the variables, there is a path through the program
- For every path, constant propagation computes the values of the x_i
- And gets a precise value for $p(x_1, \ldots, x_n)$

Regard the following program

```
x₁=x₂=...xₙ=0
while (*) { x₁ = x₁ + 1 }
...
while (*) { xₙ = xₙ + 1 }
r=0
if (p(x₁,...,xₙ) == 0) then r=1
u: Nop
```

- For any valuation of the variables, there is a path through the program
- For every path, constant propagation computes the values of the x_i
- And gets a precise value for $p(x_1, \ldots, x_n)$
- r is only found to be non-constant, if $p(x_1, \ldots, x_n) = 0$

Regard the following program

```
x₁=x₂=... xₙ=0
while (*) { x₁ = x₁ + 1 }
...
while (*) { xₙ = xₙ + 1 }
r=0
if (p(x₁,...,xₙ) == 0) then r=1
u: Nop
```

- For any valuation of the variables, there is a path through the program
- For every path, constant propagation computes the values of the x_i
- And gets a precise value for $p(x_1, \ldots, x_n)$
- r is only found to be non-constant, if $p(x_1, \ldots, x_n) = 0$
- Thus, $\mathrm{MOP}[u](r) = \top$ if, and only if $p(x_1, \ldots, x_n) = 0$ has a solution $\square$

Extensions

- Also simplify subexpressions:
  - For $\{x \mapsto \top, y \mapsto 3\}$, replace $x + 2 * y$ by $x + 6$.

Extensions

- Also simplify subexpressions:
  - For $\{x \mapsto \top, y \mapsto 3\}$, replace $x + 2 * y$ by $x + 6$.
- Apply further arithmetic simplifications
  - E.g. $x * 0 \to 0$, $x * 1 \to x$, $\ldots$

Extensions

- Also simplify subexpressions:
  - For $\{x \mapsto \top, y \mapsto 3\}$, replace $x + 2 * y$ by $x + 6$.
- Apply further arithmetic simplifications
  - E.g. $x * 0 \to 0$, $x * 1 \to x$, ...
- Exploit equalities in conditions
  - `if (x==4) M[0]=x+1 else M[0]=x` $\to$

Extensions

- Also simplify subexpressions:
  - For $\{x \mapsto \top, y \mapsto 3\}$, replace $x + 2 * y$ by $x + 6$.
- Apply further arithmetic simplifications
  - E.g. $x * 0 \to 0$, $x * 1 \to x$, $\ldots$
- Exploit equalities in conditions
  - **`if`** `(x==4) M[0]=x+1` **`else`** `M[0]=x` $\to$
    `if` `(x==4) M[0]=5` **`else`** `M[0]=x`

Extensions

- Also simplify subexpressions:
  - For $\{x \mapsto \top, y \mapsto 3\}$, replace $x + 2 * y$ by $x + 6$.
- Apply further arithmetic simplifications
  - E.g. $x * 0 \to 0$, $x * 1 \to x$, ...
- Exploit equalities in conditions
  - `if (x==4) M[0]=x+1 else M[0]=x` $\to$ `if (x==4) M[0]=5 else M[0]=x`
  - Use

$$[\![\mathrm{Pos}(x == e)]\!]^{\#} = \begin{cases} D & \text{if } [\![x == e]\!]^{\#} D = 1 \\ \bot & \text{if } [\![x == e]\!]^{\#} D = 0 \\ D_1 & \text{otherwise} \end{cases}$$

where $D_1 := D(x := D(x) \sqcap [\![e]\!]^{\#} D)$

Extensions

- Also simplify subexpressions:
  - For $\{x \mapsto \top, y \mapsto 3\}$, replace $x + 2 * y$ by $x + 6$.
- Apply further arithmetic simplifications
  - E.g. $x * 0 \to 0$, $x * 1 \to x$, ...
- Exploit equalities in conditions
  - `if (x==4) M[0]=x+1 else M[0]=x` $\to$ `if (x==4) M[0]=5 else M[0]=x`
  - Use

$$[\![\mathrm{Pos}(x == e)]\!]^\# = \begin{cases} D & \text{if } [\![x == e]\!]^\# D = 1 \\ \bot & \text{if } [\![x == e]\!]^\# D = 0 \\ D_1 & \text{otherwise} \end{cases}$$

    where $D_1 := D(x := D(x) \sqcap [\![e]\!]^\# D)$
  - Analogously for $\mathrm{Neg}(x \neq e)$

Table of Contents

Interval Analysis

- Constant propagation finds constants
- But sometimes, we can restrict the value of a variable to an interval, e.g., [0..42].

Example

```
int a[42];
for (i=0;i<42;++i) {
  if (0<=i && i<42)
    a[i] = i*2;
  else
    fail();
```

- Array access with bounds check

Example

```
int a[42];
for (i=0;i<42;++i) {
  if (0<=i && i<42)
    a[i] = i*2;
  else
    fail();
```

- Array access with bounds check
- From the for-loop, we know $i \in [0..41]$

Example

```
int a[42];
for (i=0;i<42;++i) {
  if (0<=i && i<42)
    a[i] = i*2;
  else
    fail();
```

- Array access with bounds check
- From the for-loop, we know $i \in [0..41]$
- Thus, bounds check not necessary

Intervals

Interval $\mathbb{I} := \{[l, u] \mid l \in \mathbb{Z}^{-\infty} \wedge u \in \mathbb{Z}^{+\infty} \wedge l \leq u\}$

Intervals

Interval $\mathbb{I} := \{[l, u] \mid l \in \mathbb{Z}^{-\infty} \wedge u \in \mathbb{Z}^{+\infty} \wedge l \leq u\}$

Ordering $\subseteq$, i.e. $[l_1, u_1] \sqsubseteq [l_2, u_2]$ iff $l_1 \geq l_2 \wedge u_1 \leq u_2$

- Smaller interval contained in larger one
- Hence:

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$
$$\top = [-\infty, +\infty]$$

Intervals

Interval $\mathbb{I} := \{[l, u] \mid l \in \mathbb{Z}^{-\infty} \wedge u \in \mathbb{Z}^{+\infty} \wedge l \leq u\}$

Ordering $\subseteq$, i.e. $[l_1, u_1] \sqsubseteq [l_2, u_2]$ iff $l_1 \geq l_2 \wedge u_1 \leq u_2$

- Smaller interval contained in larger one
- Hence:

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$
$$\top = [-\infty, +\infty]$$

Problems

Intervals

Interval $\mathbb{I} := \{[l, u] \mid l \in \mathbb{Z}^{-\infty} \wedge u \in \mathbb{Z}^{+\infty} \wedge l \leq u\}$

Ordering $\subseteq$, i.e. $[l_1, u_1] \sqsubseteq [l_2, u_2]$ iff $l_1 \geq l_2 \wedge u_1 \leq u_2$

- Smaller interval contained in larger one
- Hence:

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$
$$\top = [-\infty, +\infty]$$

Problems

- Not a complete lattice. (Will add $\perp$ - element later)

Intervals

Interval $\mathbb{I} := \{[l, u] \mid l \in \mathbb{Z}^{-\infty} \wedge u \in \mathbb{Z}^{+\infty} \wedge l \leq u\}$

Ordering $\subseteq$, i.e. $[l_1, u_1] \sqsubseteq [l_2, u_2]$ iff $l_1 \geq l_2 \wedge u_1 \leq u_2$

- Smaller interval contained in larger one
- Hence:

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$
$$\top = [-\infty, +\infty]$$

Problems

- Not a complete lattice. (Will add $\bot$ - element later)
- Infinite ascending chains: $[0, 0] \sqsubset [0, 1] \sqsubset [0, 2] \sqsubset \ldots$

Building the Domain

- Analogously to CP:
  - $\mathbb{D} := (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$
  - Intuition: Map variables to intervals their value must be contained in.
  - $\bot$ — unreachable

Building the Domain

- Analogously to CP:
  - $\mathbb{D} := (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$
  - Intuition: Map variables to intervals their value must be contained in.
  - $\bot$ — unreachable
- Description relation:

Building the Domain

- Analogously to CP:
  - $\mathbb{D} := (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$
  - Intuition: Map variables to intervals their value must be contained in.
  - $\bot$ — unreachable
- Description relation:
  - On values: $z \; \Delta \; [l, u]$ iff $l \leq z \leq u$

Building the Domain

- Analogously to CP:
  - $\mathbb{D} := (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$
  - Intuition: Map variables to intervals their value must be contained in.
  - $\bot$ — unreachable
- Description relation:
  - On values: $z \,\Delta\, [l, u]$ iff $l \leq z \leq u$
  - On register valuations: $\rho \,\Delta\, \rho^{\#}$ iff $\forall x. \rho(x) \,\Delta\, \rho^{\#}(x)$

Building the Domain

- Analogously to CP:
  - $\mathbb{D} := (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$
  - Intuition: Map variables to intervals their value must be contained in.
  - $\bot$ — unreachable
- Description relation:
  - On values: $z \; \Delta \; [l, u]$ iff $l \leq z \leq u$
  - On register valuations: $\rho \; \Delta \; \rho^\#$ iff $\forall x. \rho(x) \; \Delta \; \rho^\#(x)$
  - On configurations: $(\rho, \mu) \; \Delta \; I$ iff $\rho \; \Delta \; I$ and $I \neq \bot$

Building the Domain

- Analogously to CP:
  - $\mathbb{D} := (\text{Reg} \to \mathbb{I}) \cup \{\bot\}$
  - Intuition: Map variables to intervals their value must be contained in.
  - $\bot$ — unreachable
- Description relation:
  - On values: $z \;\Delta\; [l, u]$ iff $l \leq z \leq u$
  - On register valuations: $\rho \;\Delta\; \rho^{\#}$ iff $\forall x. \rho(x) \;\Delta\; \rho^{\#}(x)$
  - On configurations: $(\rho, \mu) \;\Delta\; I$ iff $\rho \;\Delta\; I$ and $I \neq \bot$
  - Obviously monotonic. (Larger interval admits more values)

Abstract operators

Constants $c^{\#} := [c, c]$

Abstract operators

Constants $c^{\#} := [c, c]$

Addition $[l_1, u_1] +^{\#} [l_2, u_2] := [l_1 + l_2, u_1 + u_2]$

- Where $-\infty + \_ := \_ + -\infty := -\infty$, $\infty + \_ := \_ + \infty := \infty$

Abstract operators

Constants $c^{\#} := [c, c]$

Addition $[l_1, u_1] +^{\#} [l_2, u_2] := [l_1 + l_2, u_1 + u_2]$

- Where $-\infty + \_ := \_ + -\infty := -\infty$, $\infty + \_ := \_ + \infty := \infty$

Negation $-^{\#}[l, u] := [-u, -l]$

Abstract operators

Constants $c^{\#} := [c, c]$

Addition $[l_1, u_1] +^{\#} [l_2, u_2] := [l_1 + l_2, u_1 + u_2]$

- Where $-\infty + \_ := \_ + -\infty := -\infty$, $\infty + \_ := \_ + \infty := \infty$

Negation $-^{\#}[l, u] := [-u, -l]$

Multiplication $[l_1, u_1] *^{\#} [l_2, u_2] :=$
$[\min\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}, \max\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}]$

Abstract operators

Constants $c^\# := [c, c]$

Addition $[l_1, u_1] +^\# [l_2, u_2] := [l_1 + l_2, u_1 + u_2]$

- Where $-\infty + \_ := \_ + -\infty := -\infty$, $\infty + \_ := \_ + \infty := \infty$

Negation $-^\#[l, u] := [-u, -l]$

Multiplication $[l_1, u_1] *^\# [l_2, u_2] :=$
$[\min\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}, \max\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}]$

Division $[l_1, u_1]/^\#[l_2, u_2] :=$
$[\min\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}, \max\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}]$

- If $0 \notin [l_2, u_2]$, otherwise $[l_1, u_1]/^\#[l_2, u_2] := \top$

Examples

- $5^{\#} =$

Examples

- $5^{\#} = [5, 5]$

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] =$

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] = [2, \infty]$

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] = [2, \infty]$
- $[-1, 3] *^{\#} [-5, -1] =$

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] = [2, \infty]$
- $[-1, 3] *^{\#} [-5, -1] = [-15, 5]$

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] = [2, \infty]$
- $[-1, 3] *^{\#} [-5, -1] = [-15, 5]$
- $-^{\#}[1, 5] =$

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] = [2, \infty]$
- $[-1, 3] *^{\#} [-5, -1] = [-15, 5]$
- $-^{\#}[1, 5] = [-5, -1]$

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] = [2, \infty]$
- $[-1, 3] *^{\#} [-5, -1] = [-15, 5]$
- $-^{\#}[1, 5] = [-5, -1]$
- $[3, 5]/^{\#}[2, 5] =$ (round towards zero)

Examples

- $5^{\#} = [5, 5]$
- $[3, \infty] +^{\#} [-1, 2] = [2, \infty]$
- $[-1, 3] *^{\#} [-5, -1] = [-15, 5]$
- $-^{\#}[1, 5] = [-5, -1]$
- $[3, 5]/^{\#}[2, 5] = [0, 2]$ (round towards zero)

Examples

- $5^{\#} = [5,5]$
- $[3,\infty] +^{\#} [-1,2] = [2,\infty]$
- $[-1,3] *^{\#} [-5,-1] = [-15,5]$
- $-^{\#}[1,5] = [-5,-1]$
- $[3,5]/^{\#}[2,5] = [0,2]$ (round towards zero)
- $[1,4]/^{\#}[-1,1] =$

Examples

- $5^\# = [5, 5]$
- $[3, \infty] +^\# [-1, 2] = [2, \infty]$
- $[-1, 3] *^\# [-5, -1] = [-15, 5]$
- $-^\#[1, 5] = [-5, -1]$
- $[3, 5]/^\#[2, 5] = [0, 2]$ (round towards zero)
- $[1, 4]/^\#[-1, 1] = \top$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } u_1 \leq l_2 \\ [0,0] & \text{if } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } u_1 \leq l_2 \\ [0,0] & \text{if } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Examples

- $[1, 2] ==^{\#} [4, 5] =$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } u_1 \leq l_2 \\ [0,0] & \text{if } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Examples

- $[1,2] ==^{\#} [4,5] = [0,0]$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } u_1 \leq l_2 \\ [0,0] & \text{if } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Examples

- $[1,2] ==^{\#} [4,5] = [0,0]$
- $[1,2] ==^{\#} [-1,1] =$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } u_1 \leq l_2 \\ [0,0] & \text{if } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Examples

- $[1,2] ==^{\#} [4,5] = [0,0]$
- $[1,2] ==^{\#} [-1,1] = [0,1]$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } u_1 \leq l_2 \\ [0,0] & \text{if } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Examples

- $[1,2] ==^{\#} [4,5] = [0,0]$
- $[1,2] ==^{\#} [-1,1] = [0,1]$
- $[1,2] \leq^{\#} [4,5] =$

Abstract operators

Equality

$$[l_1, u_1] ==^\# [l_2, u_2] := \begin{cases} [1, 1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0, 0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0, 1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^\# [l_2, u_2] := \begin{cases} [1, 1] & \text{if } u_1 \leq l_2 \\ [0, 0] & \text{if } l_1 > u_2 \\ [0, 1] & \text{otherwise} \end{cases}$$

Examples

- $[1, 2] ==^\# [4, 5] = [0, 0]$
- $[1, 2] ==^\# [-1, 1] = [0, 1]$
- $[1, 2] \leq^\# [4, 5] = [1, 1]$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1, 1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0, 0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0, 1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1, 1] & \text{if } u_1 \leq l_2 \\ [0, 0] & \text{if } l_1 > u_2 \\ [0, 1] & \text{otherwise} \end{cases}$$

Examples

- $[1, 2] ==^{\#} [4, 5] = [0, 0]$
- $[1, 2] ==^{\#} [-1, 1] = [0, 1]$
- $[1, 2] \leq^{\#} [4, 5] = [1, 1]$
- $[1, 2] \leq^{\#} [-1, 1] =$

Abstract operators

Equality

$$[l_1, u_1] ==^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \text{ or } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Less-or-equal

$$[l_1, u_1] \leq^{\#} [l_2, u_2] := \begin{cases} [1,1] & \text{if } u_1 \leq l_2 \\ [0,0] & \text{if } l_1 > u_2 \\ [0,1] & \text{otherwise} \end{cases}$$

Examples

- $[1,2] ==^{\#} [4,5] = [0,0]$
- $[1,2] ==^{\#} [-1,1] = [0,1]$
- $[1,2] \leq^{\#} [4,5] = [1,1]$
- $[1,2] \leq^{\#} [-1,1] = [0,1]$

Proof obligations

$c \mathrel{\Delta} c^{\#}$

$v_1 \mathrel{\Delta} d_1 \wedge v_2 \mathrel{\Delta} d_2 \implies v_1 \square v_2 \mathrel{\Delta} d_1 \square^{\#} d_2$

Analogously for unary, ternary, etc. operators

Proof obligations

$c \; \Delta \; c^{\#}$

$v_1 \; \Delta \; d_1 \wedge v_2 \; \Delta \; d_2 \implies v_1 \,\Box\, v_2 \; \Delta \; d_1 \,\Box^{\#}\, d_2$

Analogously for unary, ternary, etc. operators

Then, we get $\rho \; \Delta \; \rho^{\#} \implies [\![e]\!]\rho \; \Delta \; [\![e]\!]^{\#}\rho^{\#}$

- As for constant propagation

Effects of edges

For $\rho^\# \neq \bot$

$$
\begin{aligned}
[\![\cdot]\!]^\# \bot &= \bot \\
[\![\mathrm{Nop}]\!]^\# \rho^\# &= \rho^\# \\
[\![x = e]\!]^\# \rho^\# &= \rho^\#(x \mapsto [\![e]\!]^\# \rho^\#) \\
[\![x = M[e]]\!]^\# \rho^\# &= \rho^\#(x \mapsto \top) \\
[\![M[e_1] = e_2]\!]^\# \rho^\# &= \rho^\# \\
[\![\mathrm{Pos}(e)]\!]^\# \rho^\# &= \begin{cases} \bot & \text{if } [\![e]\!]^\# \rho^\# = [0,0] \\ \rho^\# & \text{otherwise} \end{cases} \\
[\![\mathrm{Neg}(e)]\!]^\# \rho^\# &= \begin{cases} \rho^\# & \text{if } [\![e]\!]^\# \rho^\# \sqsupseteq [0,0] \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Last lecture

- Constant propagation
  - Idea: Abstract description of values, lift to valuations, states
- Monotonic, but not distributive
  - MOP solution undecidable (Reduction to Hilbert's 10th problem)
- Interval analysis
  - Associate variables with intervals of possible values

Better exploitation of conditions

$$
[\![\mathrm{Pos}(e)]\!]^\# \rho^\# = \begin{cases}
\bot & \text{if } [\![e]\!]^\# \rho^\# = [0,0] \\
\rho^\# (x \mapsto \rho^\#(x) \sqcap [\![e_1]\!]^\# \rho^\#) & \text{if } e \ = \ x == e_1 \\
\rho^\# (x \mapsto \rho^\#(x) \sqcap [-\infty, u]) & \text{if } e \ = \ x \leq e_1 \text{ and } [\![e_1]\!]^\# \rho^\# = [\_, u] \\
\rho^\# (x \mapsto \rho^\#(x) \sqcap [l, \infty]) & \text{if } e \ = \ x \geq e_1 \text{ and } [\![e_1]\!]^\# \rho^\# = [l, \_] \\
\ldots & \\
\rho^\# & \text{otherwise}
\end{cases}
$$

$$
[\![\mathrm{Neg}(e)]\!]^\# \rho^\# = \begin{cases}
\bot & \text{if } [\![e]\!]^\# \rho^\# \not\sqsupseteq [0,0] \\
\rho^\# (x \mapsto \rho^\#(x) \sqcap [\![e_1]\!]^\# \rho^\#) & \text{if } e \ = \ x \neq e_1 \\
\rho^\# (x \mapsto \rho^\#(x) \sqcap [-\infty, u]) & \text{if } e \ = \ x > e_1 \text{ and } [\![e_1]\!]^\# \rho^\# = [\_, u] \\
\rho^\# (x \mapsto \rho^\#(x) \sqcap [l, \infty]) & \text{if } e \ = \ x < e_1 \text{ and } [\![e_1]\!]^\# \rho^\# = [l, \_] \\
\ldots & \\
\rho^\# & \text{otherwise}
\end{cases}
$$

- where $[l_1, u_1] \sqcap [l_2, u_2] = [\max(l_1, l_2), \min(u_1, u_2)]$
  - only exists if intervals overlap
  - this is guaranteed by conditions

Transformations

- Erase nodes u with $\mathrm{MOP}[u] = \bot$ (unreachable)

Transformations

- Erase nodes u with $\mathrm{MOP}[u] = \bot$ (unreachable)
- Replace subexpressions e with $[\![e]\!]^\# \rho^\# = [v, v]$ by v (constant propagation)

Transformations

- Erase nodes u with $\mathrm{MOP}[u] = \bot$ (unreachable)
- Replace subexpressions e with $[\![e]\!]^{\#}\rho^{\#} = [v, v]$ by v (constant propagation)
- Replace $\mathrm{Pos}(e)$ by Nop if $[0, 0] \not\sqsubseteq [\![e]\!]^{\#}\rho^{\#}$ (0 cannot occur)

Transformations

- Erase nodes u with $\mathrm{MOP}[u] = \bot$ (unreachable)
- Replace subexpressions e with $[\![e]\!]^{\#}\rho^{\#} = [v, v]$ by v (constant propagation)
- Replace $\mathrm{Pos}(e)$ by Nop if $[0, 0] \not\sqsubseteq [\![e]\!]^{\#}\rho^{\#}$ (0 cannot occur)
- Replace $\mathrm{Neg}(e)$ by Nop if $[\![e]\!]^{\#}\rho^{\#} = [0, 0]$ (Only 0 can occur)

Transformations

- Erase nodes u with $\mathrm{MOP}[u] = \bot$ (unreachable)
- Replace subexpressions e with $[\![e]\!]^{\#}\rho^{\#} = [v, v]$ by v (constant propagation)
- Replace $\mathrm{Pos}(e)$ by Nop if $[0, 0] \not\sqsubseteq [\![e]\!]^{\#}\rho^{\#}$ (0 cannot occur)
- Replace $\mathrm{Neg}(e)$ by Nop if $[\![e]\!]^{\#}\rho^{\#} = [0, 0]$ (Only 0 can occur)
- Yields function $\mathrm{tr}(k, \rho^{\#})$

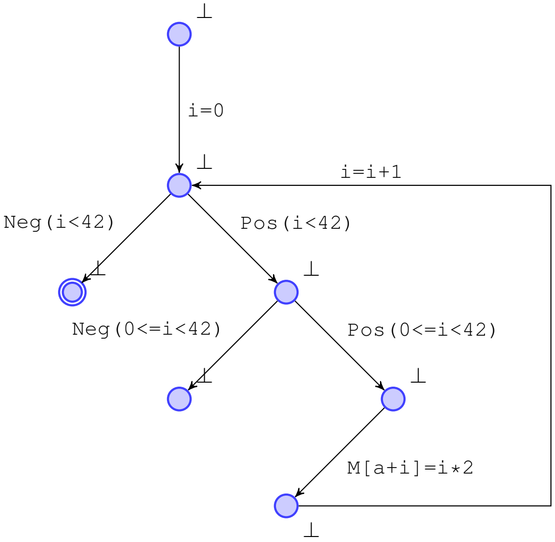
Transformations

- Erase nodes u with $\mathrm{MOP}[u] = \bot$ (unreachable)
- Replace subexpressions e with $[\![e]\!]^{\#}\rho^{\#} = [v, v]$ by v (constant propagation)
- Replace $\mathrm{Pos}(e)$ by Nop if $[0, 0] \not\sqsubseteq [\![e]\!]^{\#}\rho^{\#}$ (0 cannot occur)
- Replace $\mathrm{Neg}(e)$ by Nop if $[\![e]\!]^{\#}\rho^{\#} = [0, 0]$ (Only 0 can occur)
- Yields function $\mathrm{tr}(k, \rho^{\#})$
- Transformation: $(u, k, v) \mapsto (u, \mathrm{tr}(k, \mathrm{MFP}[u]), v)$

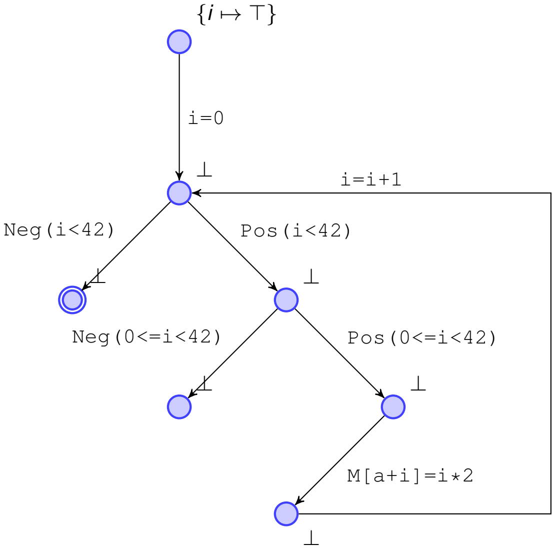
Transformations

- Erase nodes u with $\mathrm{MOP}[u] = \bot$ (unreachable)
- Replace subexpressions e with $[\![e]\!]^{\#}\rho^{\#} = [v, v]$ by v (constant propagation)
- Replace $\mathrm{Pos}(e)$ by Nop if $[0, 0] \not\sqsubseteq [\![e]\!]^{\#}\rho^{\#}$ (0 cannot occur)
- Replace $\mathrm{Neg}(e)$ by Nop if $[\![e]\!]^{\#}\rho^{\#} = [0, 0]$ (Only 0 can occur)
- Yields function $\mathrm{tr}(k, \rho^{\#})$
- Transformation: $(u, k, v) \mapsto (u, \mathrm{tr}(k, \mathrm{MFP}[u]), v)$
- Proof obligation:
  - $(\rho, \mu)\ \Delta\ \rho^{\#} \implies [\![k]\!](\rho, \mu) = [\![\mathrm{tr}(k, \rho^{\#})]\!](\rho, \mu)$

Example

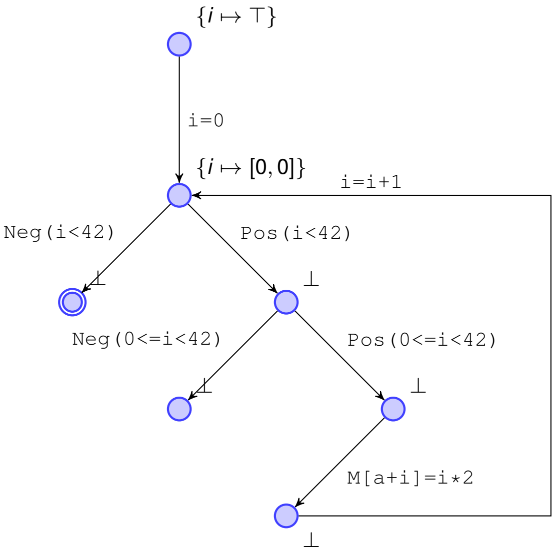


Example



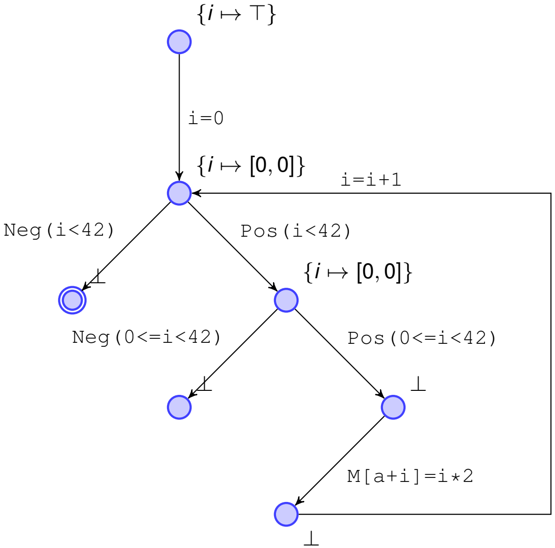
$\{i \mapsto \top\}$
i=0
$\perp$
i=i+1
Neg(i<42)
Pos(i<42)
$\perp$
$\perp$
Neg(0<=i<42)
Pos(0<=i<42)
$\perp$
$\perp$
M[a+i]=i*2
$\perp$

Example



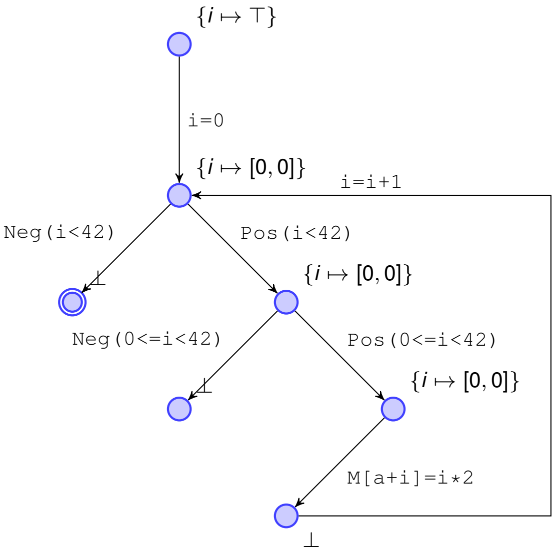
$\{i \mapsto \top\}$
i=0
$\{i \mapsto [0, 0]\}$
i=i+1
Neg(i<42)
Pos(i<42)
$\perp$
$\perp$
Neg(0<=i<42)
Pos(0<=i<42)
$\perp$
$\perp$
M[a+i]=i*2
$\perp$

Example



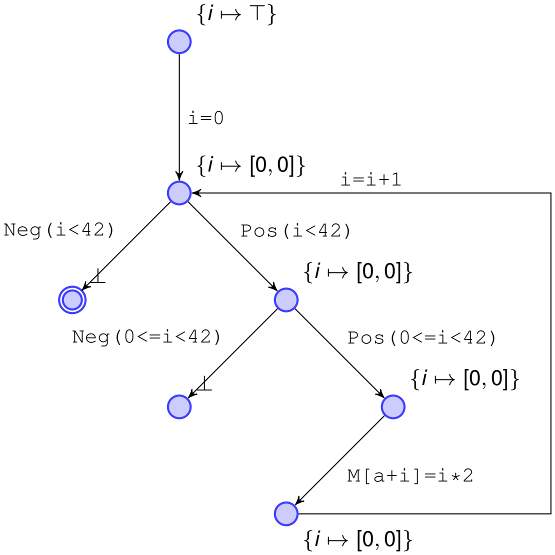
$\{i \mapsto \top\}$
i=0
$\{i \mapsto [0,0]\}$
i=i+1
Neg(i<42)
Pos(i<42)
$\perp$
$\{i \mapsto [0,0]\}$
Neg(0<=i<42)
Pos(0<=i<42)
$\perp$
$\perp$
M[a+i]=i*2
$\perp$

Example



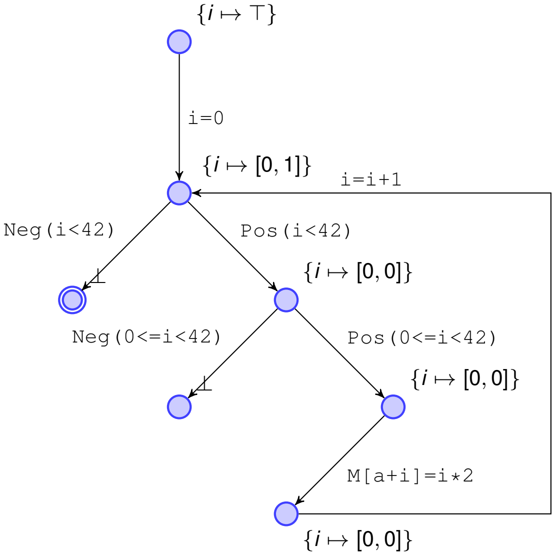
$\{i \mapsto \top\}$
i=0
$\{i \mapsto [0,0]\}$
i=i+1
Neg(i<42)
Pos(i<42)
$\bot$
$\{i \mapsto [0,0]\}$
Neg(0<=i<42)
Pos(0<=i<42)
$\bot$
$\{i \mapsto [0,0]\}$
M[a+i]=i*2
$\bot$

Example

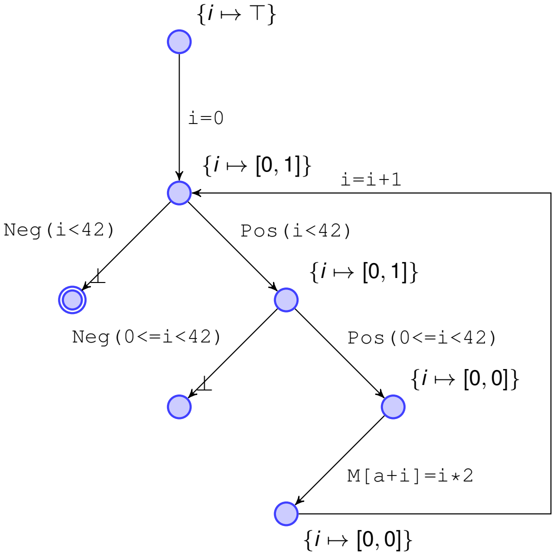


$\{i \mapsto \top\}$
i=0
$\{i \mapsto [0,0]\}$
i=i+1
Neg(i<42)
Pos(i<42)
$\perp$
$\{i \mapsto [0,0]\}$
Neg(0<=i<42)
Pos(0<=i<42)
$\perp$
$\{i \mapsto [0,0]\}$
M[a+i]=i*2
$\{i \mapsto [0,0]\}$

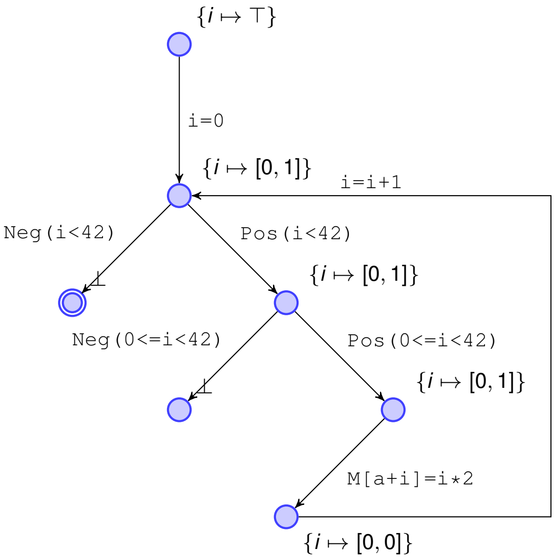
Example


$\{i \mapsto \top\}$
i=0
$\{i \mapsto [0,1]\}$
i=i+1
Neg(i<42)
Pos(i<42)
$\perp$
$\{i \mapsto [0,0]\}$
Neg(0<=i<42)
Pos(0<=i<42)
$\perp$
$\{i \mapsto [0,0]\}$
M[a+i]=i*2
$\{i \mapsto [0,0]\}$

Example

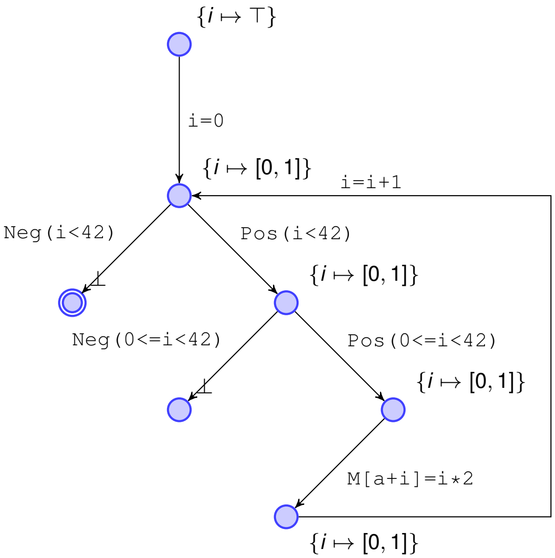


Example

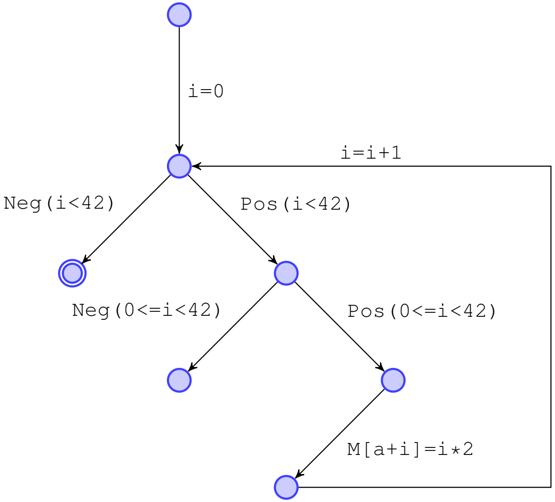


$\{i \mapsto \top\}$
i=0
$\{i \mapsto [0,1]\}$
i=i+1
Neg(i<42)
Pos(i<42)
$\bot$
$\{i \mapsto [0,1]\}$
Neg(0<=i<42)
Pos(0<=i<42)
$\bot$
$\{i \mapsto [0,1]\}$
M[a+i]=i*2
$\{i \mapsto [0,0]\}$

Example

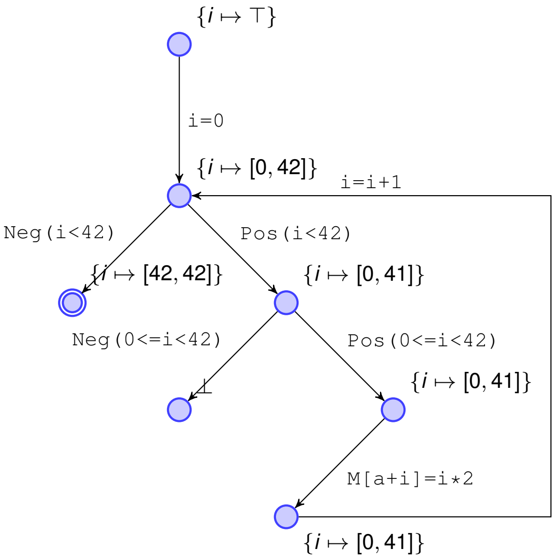


Example



About 40 iterations later ...

Example



About 40 iterations later ...

Problem

- Interval analysis takes many iterations
- May not terminate at all `for (i=0;x>0;x--) i=i+1`

Widening

- Idea: Accelerate the iteration

Widening

- Idea: Accelerate the iteration — at the price of imprecision

Widening

- Idea: Accelerate the iteration — at the price of imprecision
- Here: Disallow updates of interval bounds in $\mathbb{Z}$.
  - A maximal chain: $[3, 8] \sqsubseteq [-\infty, 8] \sqsubseteq [-\infty, \infty]$

Widening (Formally)

- Given: Constraint system (1) $x_i \sqsupseteq f_i(\vec{x})$
  - f_i not necessarily monotonic
- Regard the system (2) $x_i = x_i \sqcup f_i(\vec{x})$
- Obviously: $\vec{x}$ solution of (1) iff $\vec{x}$ solution of (2)
  - Note: $x \sqsubseteq y \iff x \sqcup y = y$
- (2) induces a function $G : \mathbb{D}^n \to \mathbb{D}^n$

$$G(\vec{x}) = \vec{x} \sqcup (f_1(\vec{x}), \ldots, f_n(\vec{x}))$$

- G is not necessarily monotonic, but increasing:

$$\forall \vec{x}.\ \vec{x} \sqsubseteq G(\vec{x})$$

Widening (Formally)

- G is increasing $\implies \bot \sqsubseteq G(\bot) \sqsubseteq G^2(\bot) \sqsubseteq \dots$
  - i.e., $\langle G^i(\bot)\rangle_{i\in\mathbb{N}}$ is ascending chain
- If it stabilizes, i.e., $\vec{x} = G^k(\bot) = G^{k+1}(\bot)$, then $\vec{x}$ is solution of (1)
- If $\mathbb{D}$ has infinite ascending chains, still no termination guaranteed
- Replace $\sqcup$ by *widening operator* $\sqcup\!\!\!\!\!\sqcup$
  - Get (3) $x_i = x_i \sqcup\!\!\!\!\!\sqcup f_i(\vec{x})$
- Widening: Any operation $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$
  1. with $x \sqcup y \sqsubseteq x \sqcup\!\!\!\!\!\sqcup y$
  2. and for every sequence $a_0, a_1, \dots$, the chain $b_0 = a_0$, $b_{i+1} = b_i \sqcup\!\!\!\!\!\sqcup a_{i+1}$ eventually stabilizes
- Using FP-iteration (naive, RR, worklist) on (3) will
  - compute a solution of (1)
  - terminate

To show

- Solutions of (3) are solutions of (1)
  - $x_i = x_i \sqcup f_i(\vec{x})$

To show

- Solutions of (3) are solutions of (1)
  - $x_i = x_i \,\underline{\sqcup}\, f_i(\vec{x}) \;\sqsupseteq x_i \sqcup f_i(\vec{x})$

To show

- Solutions of (3) are solutions of (1)
  - $x_i = x_i \sqcup f_i(\vec{x}) \sqsupseteq x_i \sqcup f_i(\vec{x}) \sqsupseteq f_i(\vec{x})$ $\square$

To show

- Solutions of (3) are solutions of (1)
  - $x_i = x_i \sqcup f_i(\vec{x}) \sqsupseteq x_i \sqcup f_i(\vec{x}) \sqsupseteq f_i(\vec{x})$ $\square$
- FP-iteration computes a solution of (3).
  - Valuation increases until it stabilizes (latest at $\vec{x} = (\top, \dots, \top)$)

To show

- Solutions of (3) are solutions of (1)
  - $x_i = x_i \sqcup\!\!\!\!\sqcup f_i(\vec{x}) \sqsupseteq x_i \sqcup f_i(\vec{x}) \sqsupseteq f_i(\vec{x})$ $\square$
- FP-iteration computes a solution of (3).
  - Valuation increases until it stabilizes (latest at $\vec{x} = (\top, \dots, \top)$)
- FP-iteration terminates
  - FP-iteration step: Replace (some) x_i by $x_i \sqcup\!\!\!\!\sqcup f_i(\vec{x})$
  - This only happens finitely many times (Widening operator, Criterion 2)

For interval analysis

- Widening defined as $[l_1, u_1] \sqcup\!\!\!\!\sqcup [l_2, u_2] := [l, u]$ with

$$l := \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases}$$

For interval analysis

- Widening defined as $[l_1, u_1] \sqcup [l_2, u_2] := [l, u]$ with

$$l := \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases}$$

- Lift to valuations: $(\rho_1^\# \sqcup \rho_2^\#)(x) := \rho_1^\#(x) \sqcup \rho_2^\#(x)$

For interval analysis

- Widening defined as $[l_1, u_1] \sqcup [l_2, u_2] := [l, u]$ with

$$l := \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases}$$

- Lift to valuations: $(\rho_1^\# \sqcup \rho_2^\#)(x) := \rho_1^\#(x) \sqcup \rho_2^\#(x)$
- and to $\mathbb{D} = (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$: $\bot \sqcup x = x \sqcup \bot = x$

For interval analysis

- Widening defined as $[l_1, u_1] \sqcup\!\!\!\sqcup [l_2, u_2] := [l, u]$ with

$$l := \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases}$$

- Lift to valuations: $(\rho_1^\# \sqcup\!\!\!\sqcup \rho_2^\#)(x) := \rho_1^\#(x) \sqcup\!\!\!\sqcup \rho_2^\#(x)$
- and to $\mathbb{D} = (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$: $\bot \sqcup\!\!\!\sqcup x = x \sqcup\!\!\!\sqcup \bot = x$
- $\sqcup\!\!\!\sqcup$ is widening operator
  1. $x \sqcup y \sqsubseteq x \sqcup\!\!\!\sqcup y$. Obvious
  2. Lower and upper bound updated at most once.

For interval analysis

- Widening defined as $[l_1, u_1] \sqcup\!\!\!\!\!\sqcup [l_2, u_2] := [l, u]$ with

$$l := \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases}$$

- Lift to valuations: $(\rho_1^\# \sqcup\!\!\!\!\!\sqcup \rho_2^\#)(x) := \rho_1^\#(x) \sqcup\!\!\!\!\!\sqcup \rho_2^\#(x)$
- and to $\mathbb{D} = (\mathrm{Reg} \to \mathbb{I}) \cup \{\bot\}$: $\bot \sqcup\!\!\!\!\!\sqcup x = x \sqcup\!\!\!\!\!\sqcup \bot = x$
- $\sqcup\!\!\!\!\!\sqcup$ is widening operator
  1. $x \sqcup y \sqsubseteq x \sqcup\!\!\!\!\!\sqcup y$. Obvious
  2. Lower and upper bound updated at most once.
- Note: $\sqcup\!\!\!\!\!\sqcup$ is not commutative.

Examples

- $[-2, 2] \sqcup [1, 2] =$

Examples

- $[-2, 2] \sqcup [1, 2] = [-2, 2]$

Examples

- $[-2,2] \sqcup [1,2] = [-2,2]$
- $[1,2] \sqcup [-2,2] =$

Examples

- $[-2,2] \sqcup [1,2] = [-2,2]$
- $[1,2] \sqcup [-2,2] = [-\infty,2]$

Examples

- $[-2, 2] \sqcup [1, 2] = [-2, 2]$
- $[1, 2] \sqcup [-2, 2] = [-\infty, 2]$
- $[1, 2] \sqcup [1, 3] =$

Examples

- $[-2, 2] \sqcup [1, 2] = [-2, 2]$
- $[1, 2] \sqcup [-2, 2] = [-\infty, 2]$
- $[1, 2] \sqcup [1, 3] = [1, +\infty]$

Examples

- $[-2,2] \sqcup [1,2] = [-2,2]$
- $[1,2] \sqcup [-2,2] = [-\infty,2]$
- $[1,2] \sqcup [1,3] = [1,+\infty]$
- Widening returns larger values more quickly

Widening (Intermediate Result)

- Define suitable widening

Widening (Intermediate Result)

- Define suitable widening
- Solve constraint system (3)

Widening (Intermediate Result)

- Define suitable widening
- Solve constraint system (3)
- Guaranteed to terminate and return over-approximation of MOP

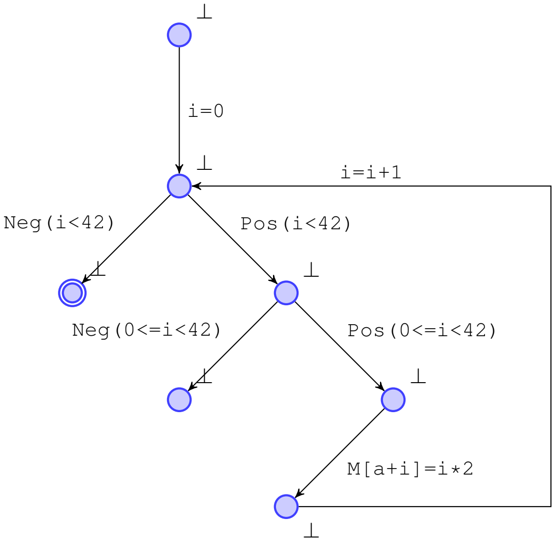
Widening (Intermediate Result)

- Define suitable widening
- Solve constraint system (3)
- Guaranteed to terminate and return over-approximation of MOP
- But: Construction of good widening is black magic

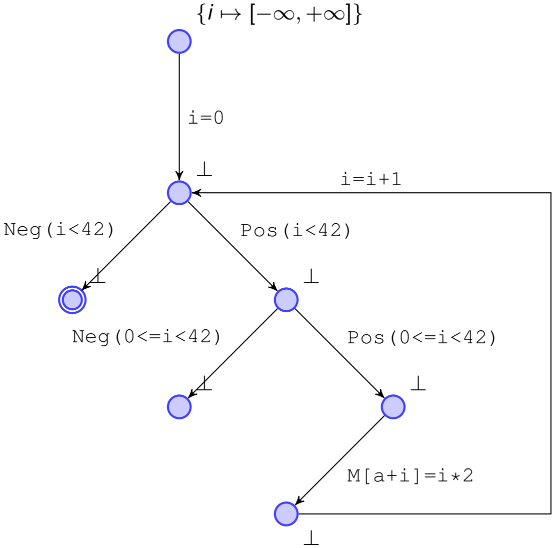
Widening (Intermediate Result)

- Define suitable widening
- Solve constraint system (3)
- Guaranteed to terminate and return over-approximation of MOP
- But: Construction of good widening is black magic
    - Even may choose $\sqcup$ dynamically during iteration, such that
        - Values do not get too complicated
        - Iteration is guaranteed to terminate

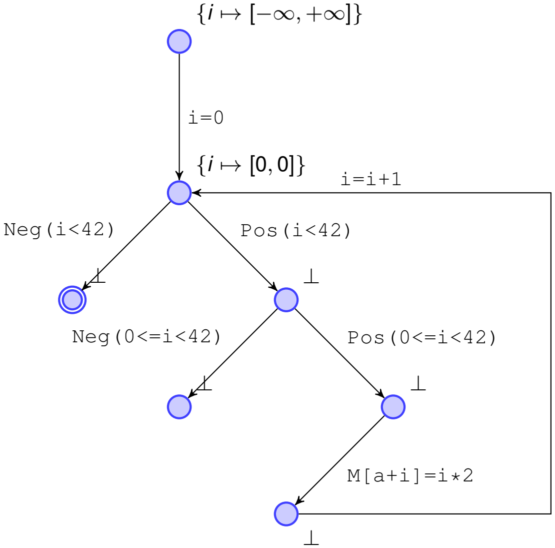
Example (Revisited)



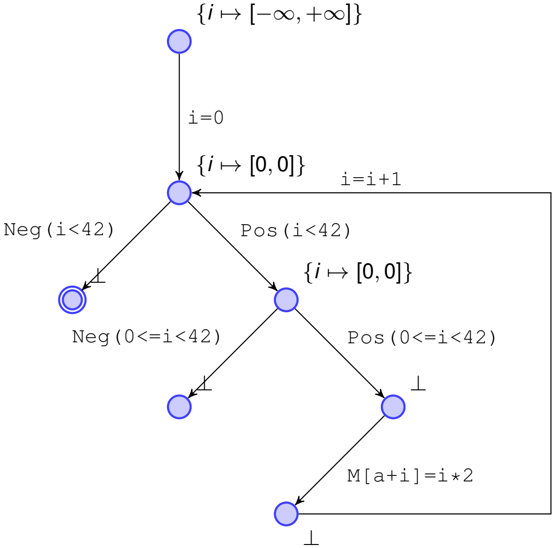
Example (Revisited)



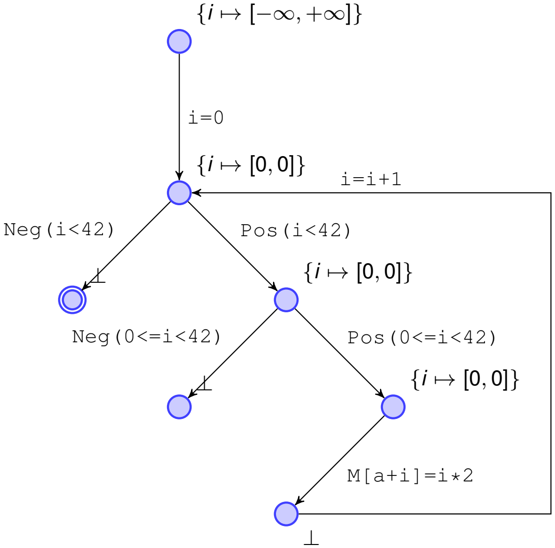
Example (Revisited)



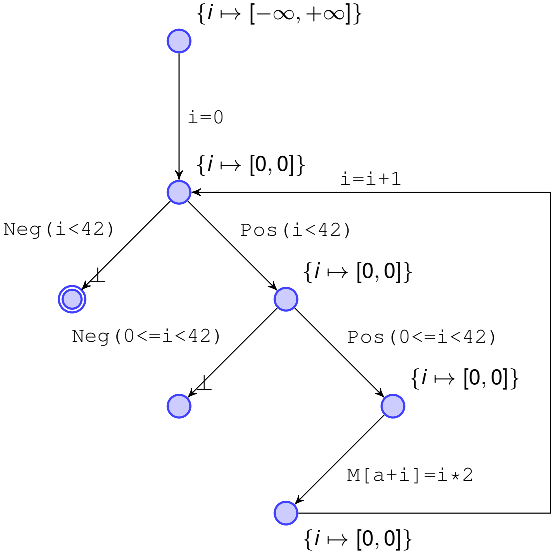
Example (Revisited)



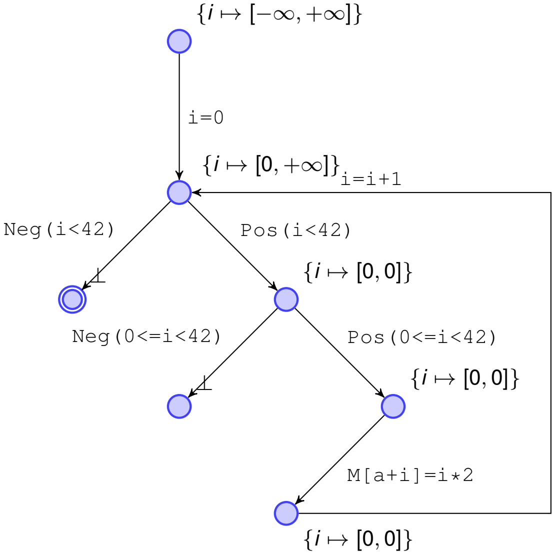
Example (Revisited)



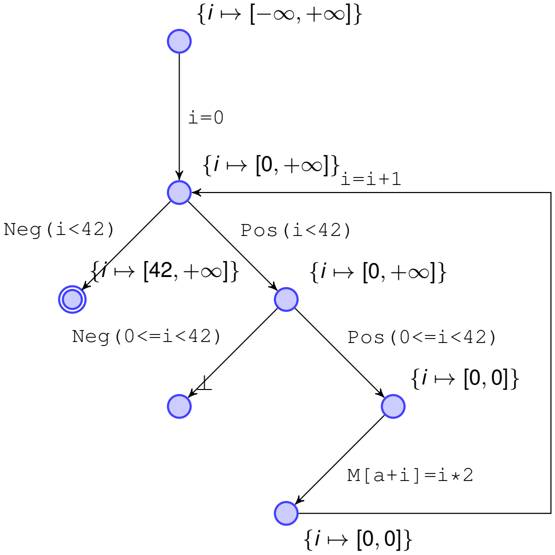
Example (Revisited)



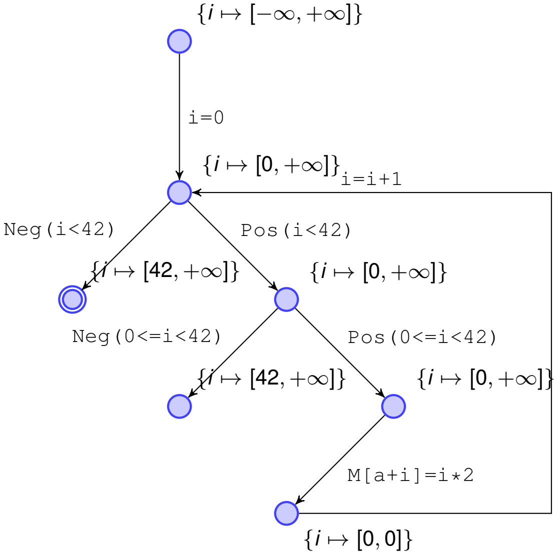
Example (Revisited)



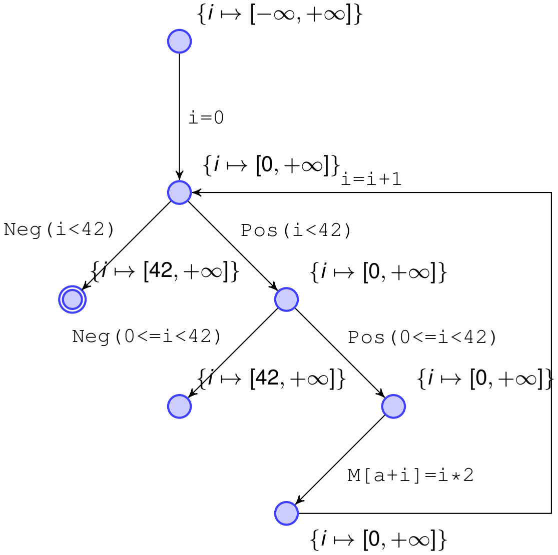
Example (Revisited)



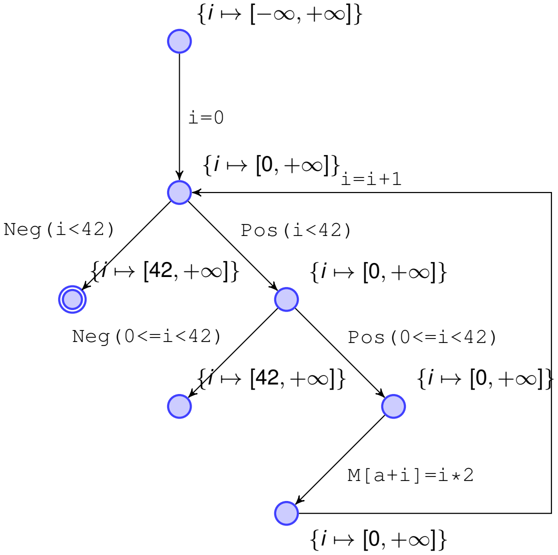
Example (Revisited)



Example (Revisited)



Example (Revisited)



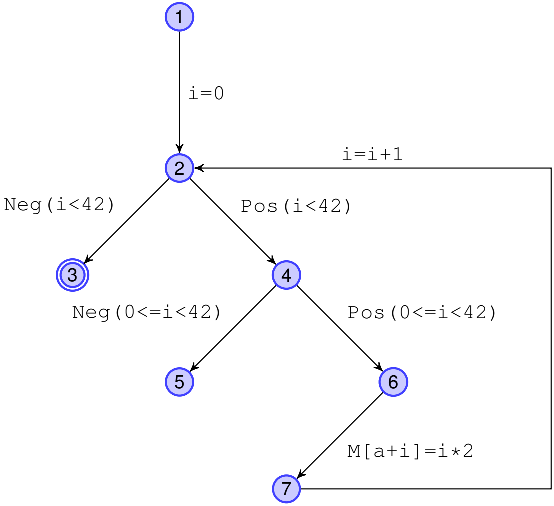
- Not exactly what we expected :(

Idea

- Only apply widening at loop separators
- A set $S \subseteq V$ is called loop separator, iff each cycle in the CFG contains a node from S.
- Intuition: Only loops can cause infinite chains of updates.
- Thus, FP-iteration still terminates

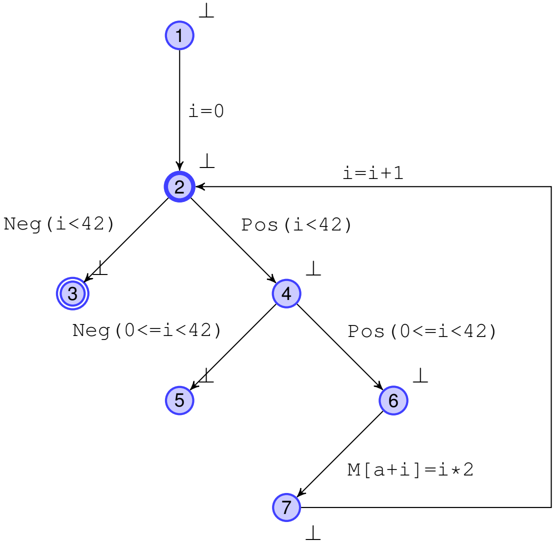
Problem

- How to find suitable loop separator

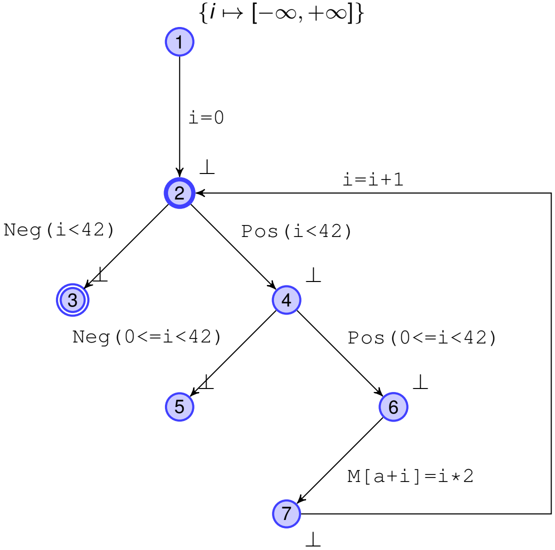


- We could take $S = \{2\}, S = \{4\}, \ldots$
- Results of FP-iteration are different!

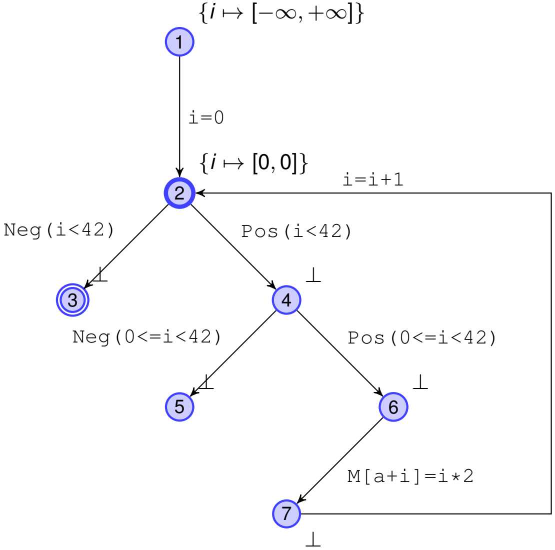
Loop Separator $S = \{2\}$



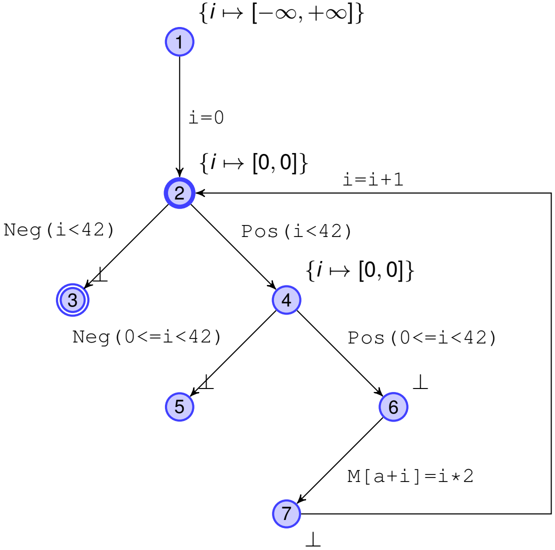
Loop Separator $S = \{2\}$



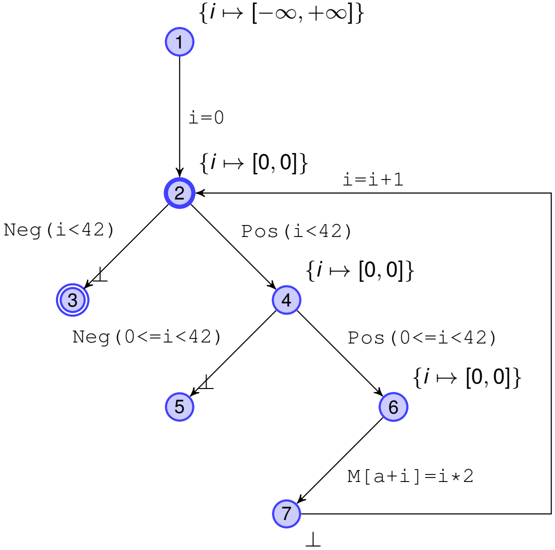
Loop Separator $S = \{2\}$



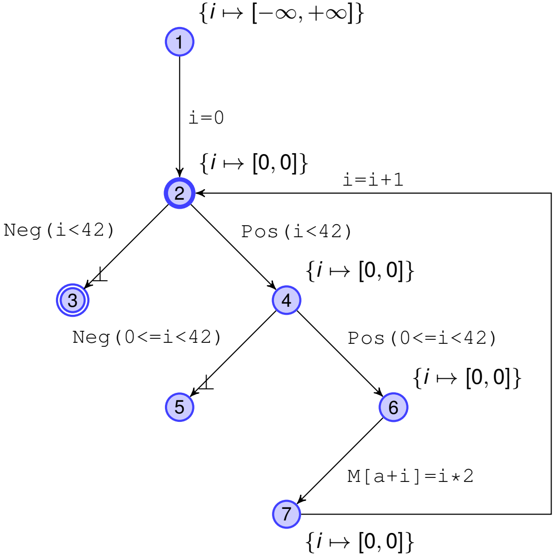
Loop Separator $S = \{2\}$



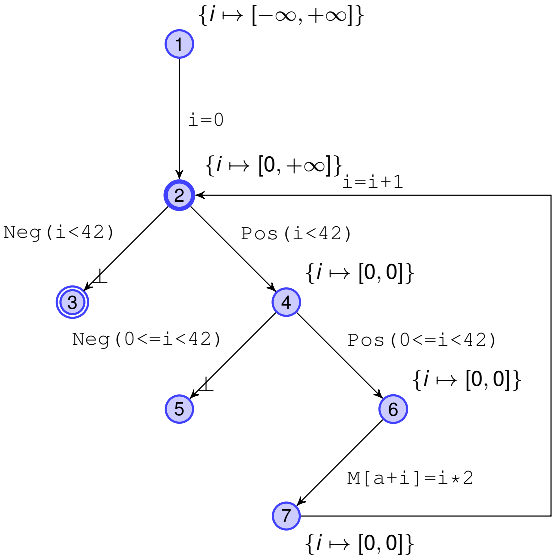
Loop Separator $S = \{2\}$



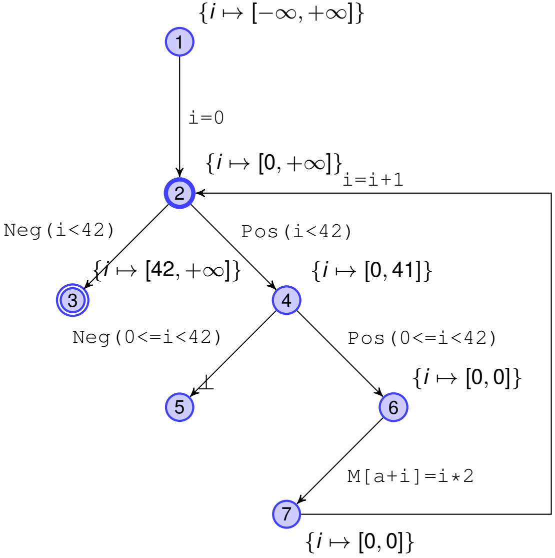
Loop Separator $S = \{2\}$



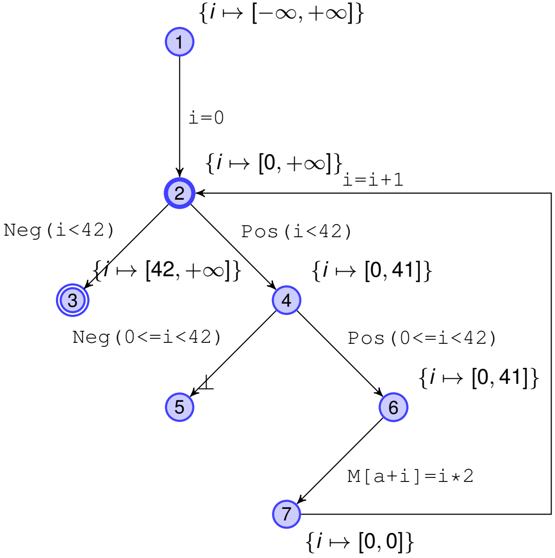
Loop Separator $S = \{2\}$



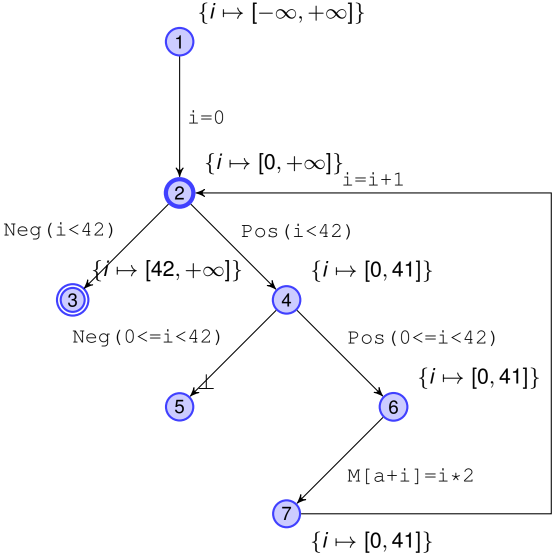
Loop Separator $S = \{2\}$



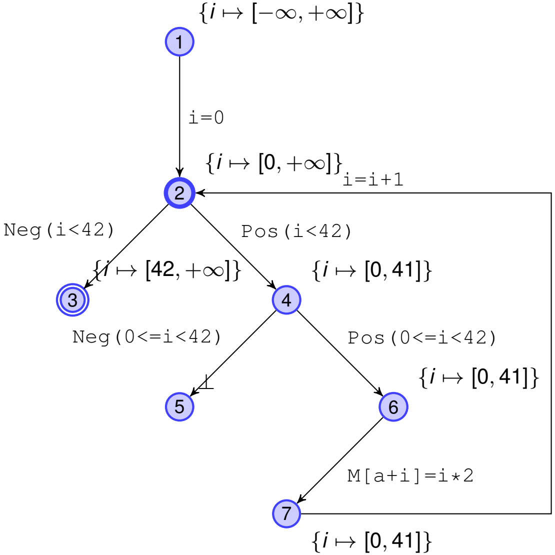
Loop Separator $S = \{2\}$



Loop Separator $S = \{2\}$

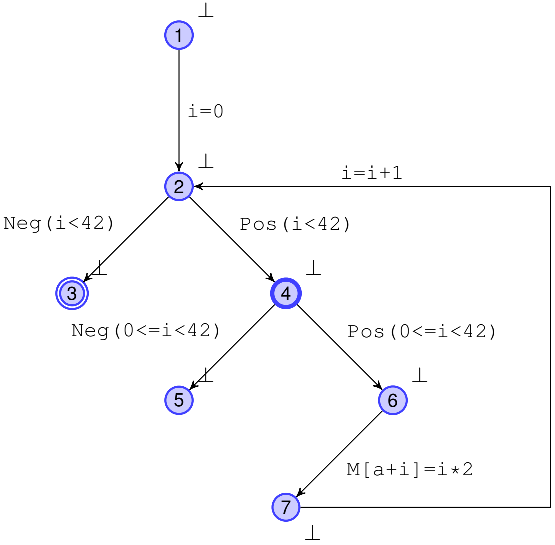


Loop Separator $S = \{2\}$

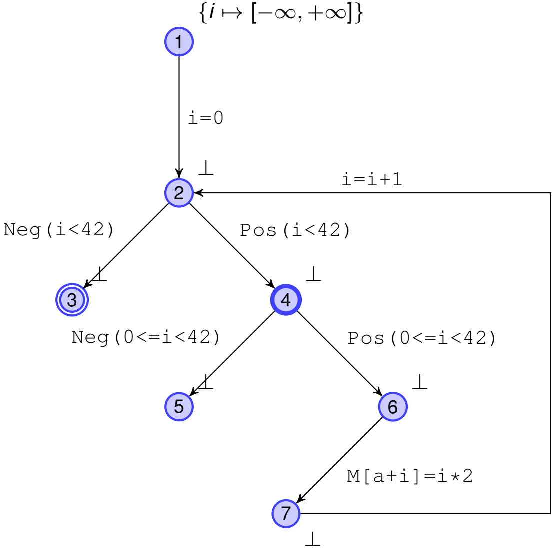


- Fixed point

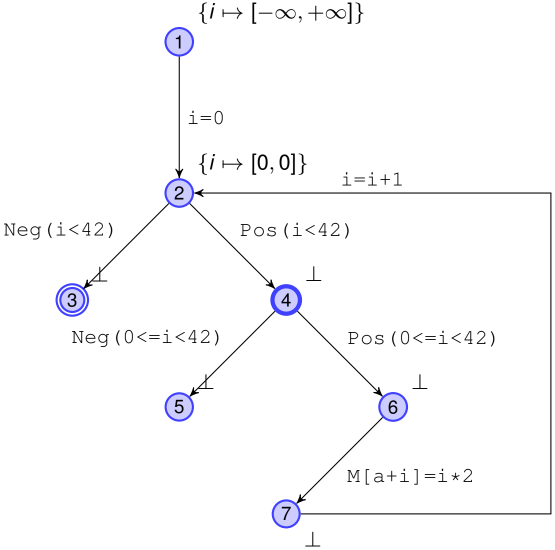
Loop Separator $S = \{4\}$



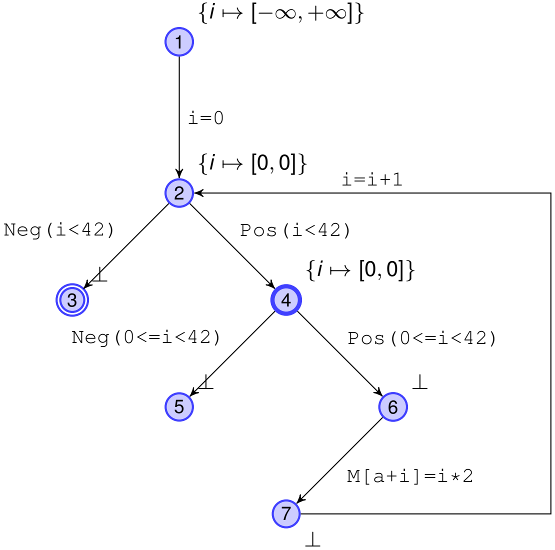
Loop Separator $S = \{4\}$



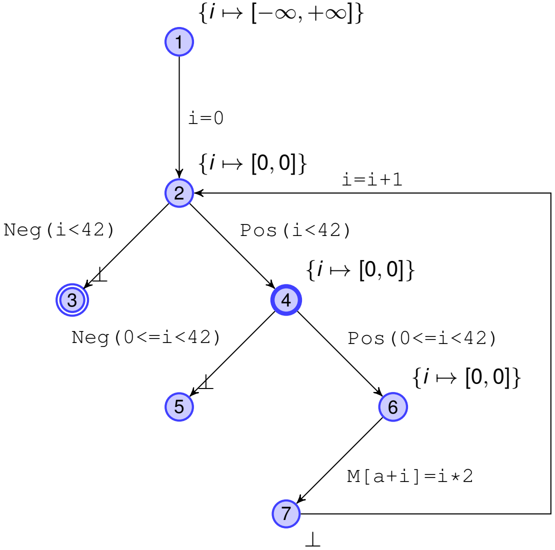
Loop Separator $S = \{4\}$



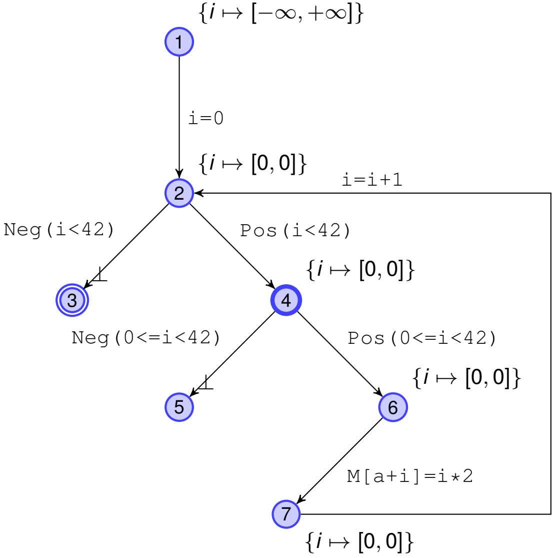
Loop Separator $S = \{4\}$



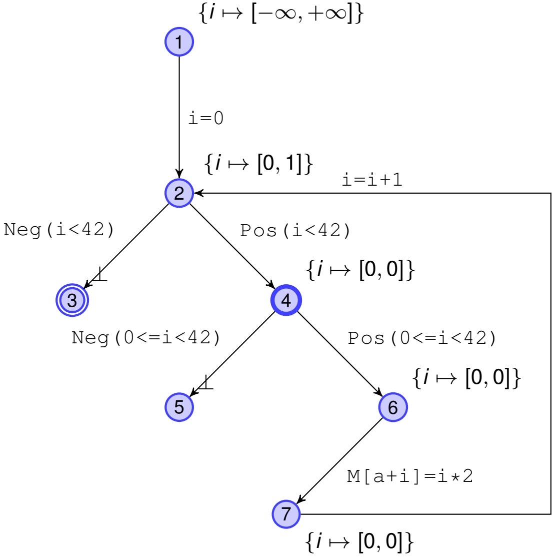
Loop Separator $S = \{4\}$



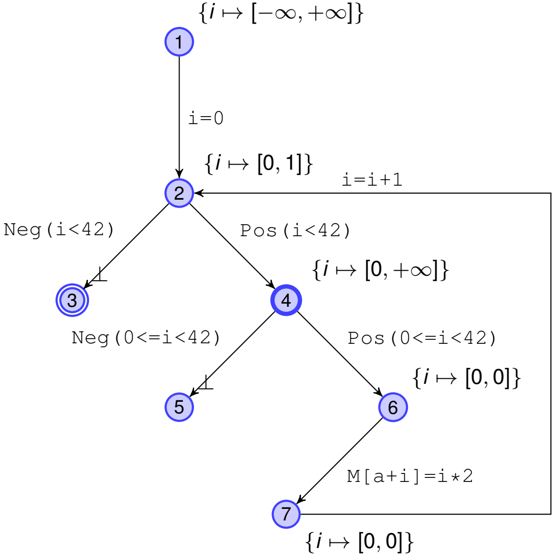
Loop Separator $S = \{4\}$



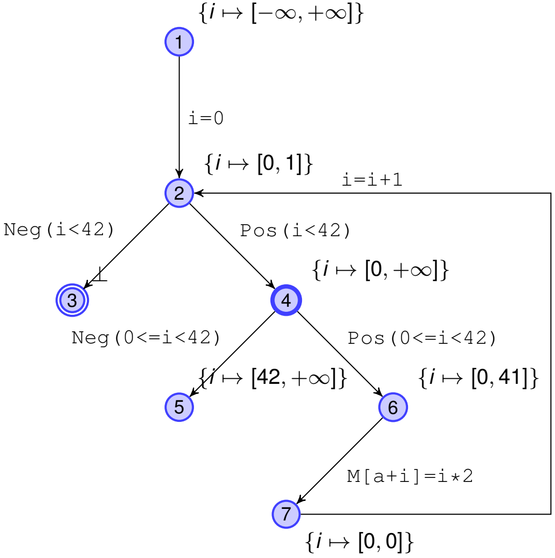
Loop Separator $S = \{4\}$



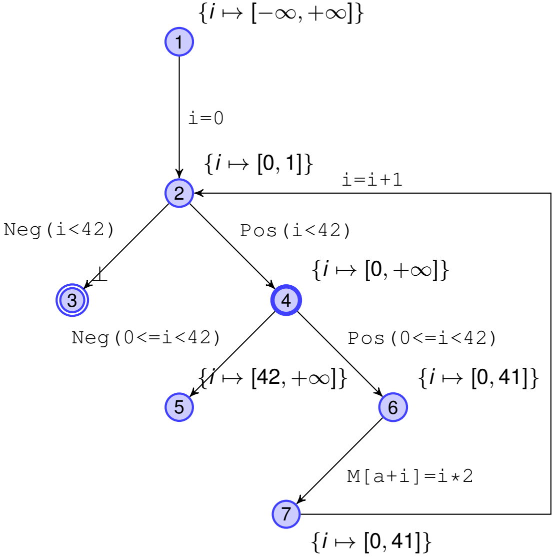
Loop Separator $S = \{4\}$



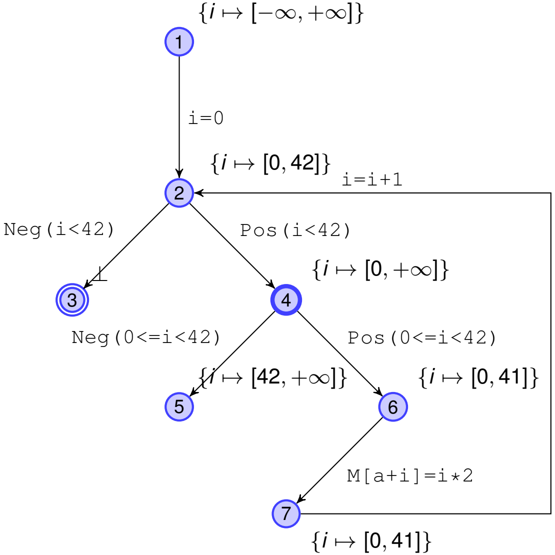
Loop Separator $S = \{4\}$



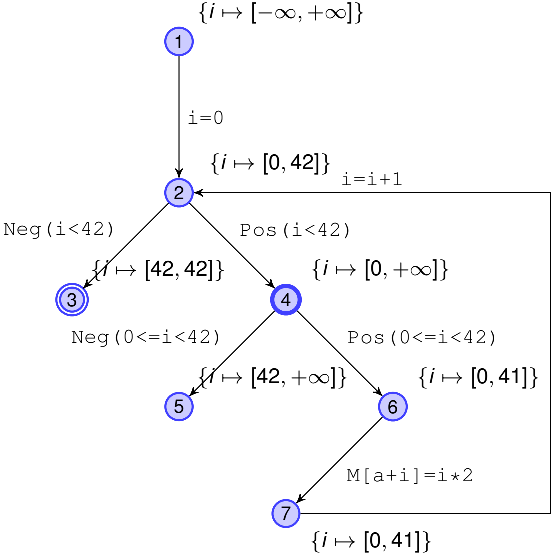
Loop Separator $S = \{4\}$



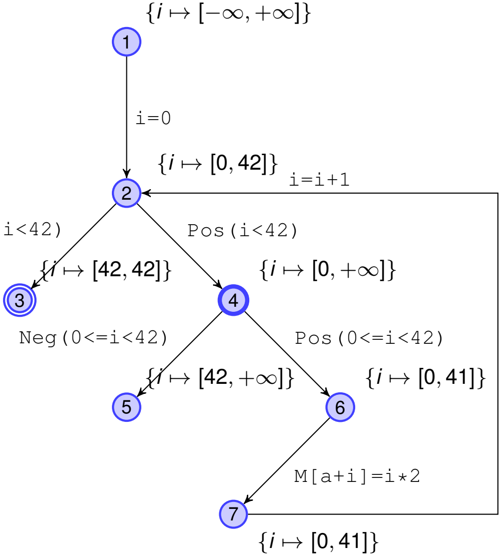
Loop Separator $S = \{4\}$



Loop Separator $S = \{4\}$



Loop Separator $S = \{4\}$



- Fixed point

Result

- Only $S = \{2\}$ identifies bounds check as superfluous

Result

- Only $S = \{2\}$ identifies bounds check as superfluous
- Only $S = \{4\}$ identifies $x = 42$ at end of program

Result

- Only $S = \{2\}$ identifies bounds check as superfluous
- Only $S = \{4\}$ identifies $x = 42$ at end of program
- We could combine the information

Result

- Only $S = \{2\}$ identifies bounds check as superfluous
- Only $S = \{4\}$ identifies $x = 42$ at end of program
- We could combine the information
  - But would be costly in general

Narrowing

- Let $\vec{x}$ be a solution of (1)

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \ldots$

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
    - By straightforward induction

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
    - By straightforward induction

$\Longrightarrow$ Every $F^k(\vec{x})$ is a solution of (1)!

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
    - By straightforward induction

$\Longrightarrow$ Every $F^k(\vec{x})$ is a solution of (1)!

- Narrowing iteration: Iterate until stabilization

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
    - By straightforward induction

$\Longrightarrow$ Every $F^k(\vec{x})$ is a solution of (1)!

- Narrowing iteration: Iterate until stabilization
  - Or some maximum number of iterations reached
    - Note: Need not stabilize within finite number of iterations

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
    - By straightforward induction

$\Longrightarrow$ Every $F^k(\vec{x})$ is a solution of (1)!

- Narrowing iteration: Iterate until stabilization
  - Or some maximum number of iterations reached
    - Note: Need not stabilize within finite number of iterations
- Solutions get smaller (more precise) with each iteration

Narrowing

- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
    - By straightforward induction

$\Longrightarrow$ Every $F^k(\vec{x})$ is a solution of (1)!

- Narrowing iteration: Iterate until stabilization
  - Or some maximum number of iterations reached
    - Note: Need not stabilize within finite number of iterations
- Solutions get smaller (more precise) with each iteration
- Round robin/Worklist iteration also works!

Narrowing

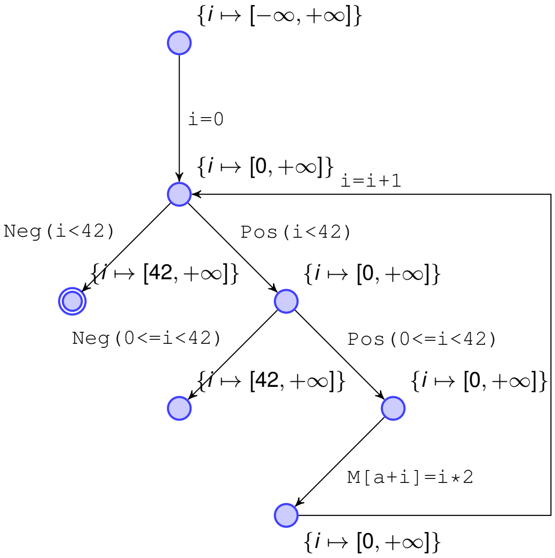
- Let $\vec{x}$ be a solution of (1)
- I.e., $x_i \sqsupseteq f_i(\vec{x})$
- Then, for monotonic f_i:
  - $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
    - By straightforward induction

$\Longrightarrow$ Every $F^k(\vec{x})$ is a solution of (1)!

- Narrowing iteration: Iterate until stabilization
  - Or some maximum number of iterations reached
    - Note: Need not stabilize within finite number of iterations
- Solutions get smaller (more precise) with each iteration
- Round robin/Worklist iteration also works!
  - Important to have only one constraint per x_i!

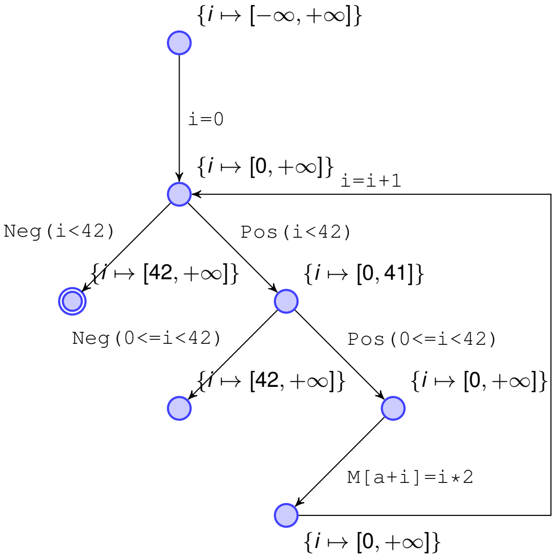
Example

- Start with over-approximation.



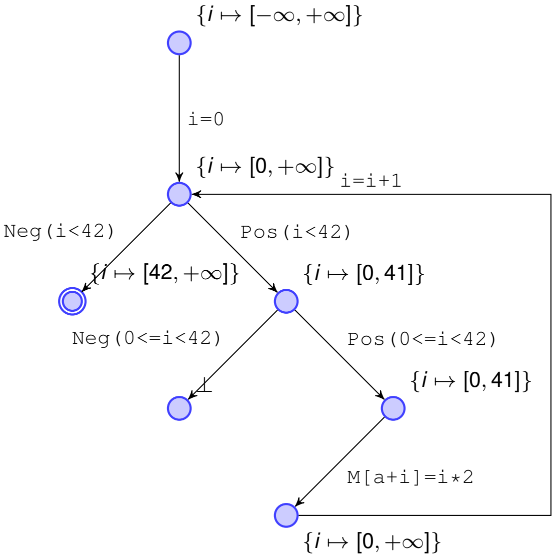
Example

- Start with over-approximation.



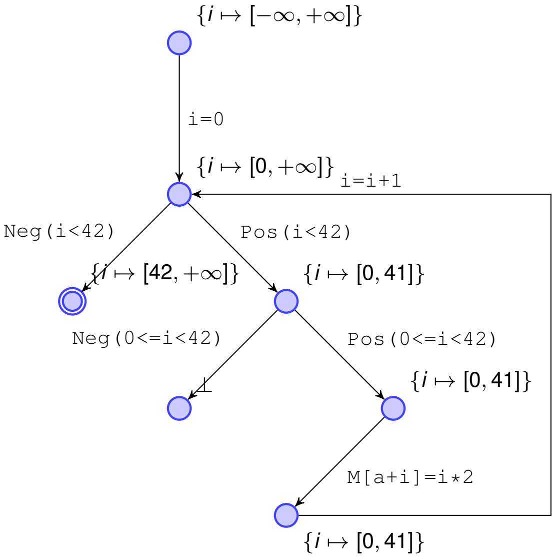
Example

- Start with over-approximation.



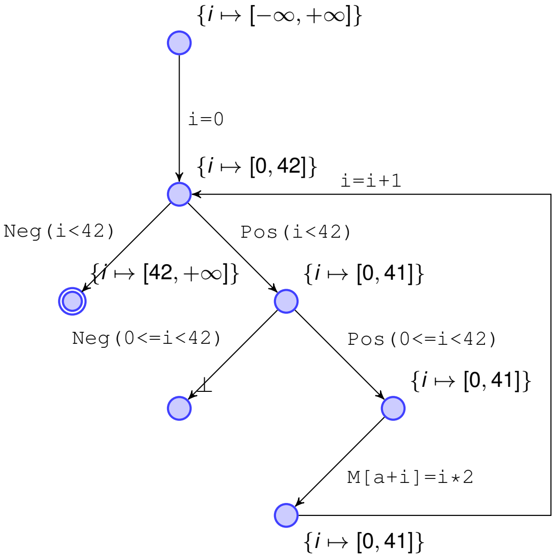
Example

- Start with over-approximation.



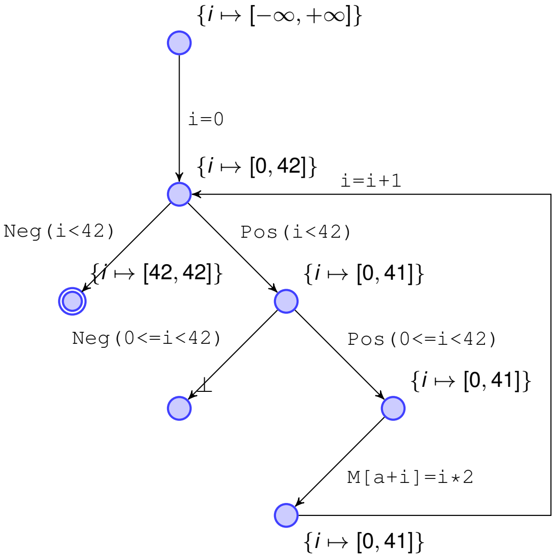
Example

- Start with over-approximation.



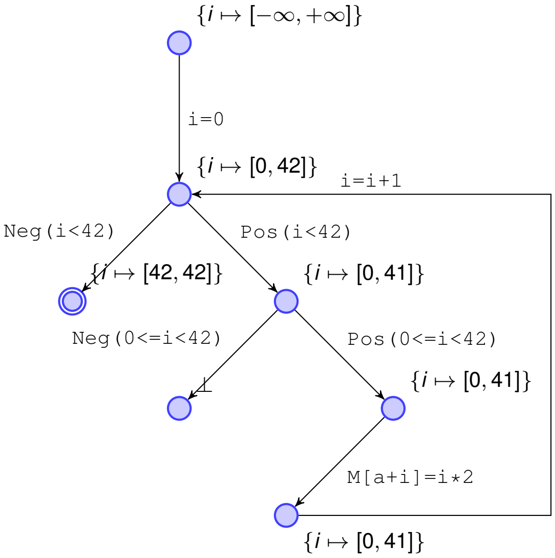
Example

- Start with over-approximation.



Example

- Start with over-approximation. Stabilized



Discussion

- Not necessary to find good loop separator

Discussion

- Not necessary to find good loop separator
- In our example, it even stabilizes

Discussion

- Not necessary to find good loop separator
- In our example, it even stabilizes
  - Otherwise: Limit number of iterations

Discussion

- Not necessary to find good loop separator
- In our example, it even stabilizes
  - Otherwise: Limit number of iterations
- Narrowing makes solution more precise in each step

Discussion

- Not necessary to find good loop separator
- In our example, it even stabilizes
  - Otherwise: Limit number of iterations
- Narrowing makes solution more precise in each step
- Question: Do we have to accept possible nontermination/large number of iterations?

Accelerated narrowing

- Let $\vec{x} \sqsupseteq F(\vec{x})$ be solution of (1)

Accelerated narrowing

- Let $\vec{x} \sqsupseteq F(\vec{x})$ be solution of (1)
- Consider function $H : \vec{x} \mapsto \vec{x} \sqcap F(\vec{x})$

Accelerated narrowing

- Let $\vec{x} \sqsupseteq F(\vec{x})$ be solution of (1)
- Consider function $H : \vec{x} \mapsto \vec{x} \sqcap F(\vec{x})$
- For monotonic F, we have $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \ldots$

Accelerated narrowing

- Let $\vec{x} \sqsupseteq F(\vec{x})$ be solution of (1)
- Consider function $H : \vec{x} \mapsto \vec{x} \sqcap F(\vec{x})$
- For monotonic F, we have $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \ldots$
  - and thus $H^k(\vec{x}) = F^k(\vec{x})$

Accelerated narrowing

- Let $\vec{x} \sqsupseteq F(\vec{x})$ be solution of (1)
- Consider function $H : \vec{x} \mapsto \vec{x} \sqcap F(\vec{x})$
- For monotonic F, we have $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \ldots$
  - and thus $H^k(\vec{x}) = F^k(\vec{x})$
- Now regard $I : (\vec{x}) \mapsto \vec{x} \sqcap\!\!\!\!\sqcap F(\vec{x})$, where

Accelerated narrowing

- Let $\vec{x} \sqsupseteq F(\vec{x})$ be solution of (1)
- Consider function $H : \vec{x} \mapsto \vec{x} \sqcap F(\vec{x})$
- For monotonic F, we have $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \dots$
  - and thus $H^k(\vec{x}) = F^k(\vec{x})$
- Now regard $I : (\vec{x}) \mapsto \vec{x} \bar{\sqcap} F(\vec{x})$, where
- $\bar{\sqcap}$: Narrowing operator, whith
  1. $x \sqcap y \sqsubseteq x \bar{\sqcap} y \sqsubseteq x$
  2. For every sequence $a_0, a_1, \dots$, the (down)chain $b_0 = a_0$, $b_{i+1} = b_i \bar{\sqcap} a_{i+1}$ eventually stabilizes

Accelerated narrowing

- Let $\vec{x} \sqsupseteq F(\vec{x})$ be solution of (1)
- Consider function $H : \vec{x} \mapsto \vec{x} \sqcap F(\vec{x})$
- For monotonic F, we have $\vec{x} \sqsupseteq F(\vec{x}) \sqsupseteq F^2(\vec{x}) \sqsupseteq \ldots$
  - and thus $H^k(\vec{x}) = F^k(\vec{x})$
- Now regard $I : (\vec{x}) \mapsto \vec{x} \bar{\sqcap} F(\vec{x})$, where
- $\bar{\sqcap}$: Narrowing operator, whith
  1. $x \sqcap y \sqsubseteq x \bar{\sqcap} y \sqsubseteq x$
  2. For every sequence $a_0, a_1, \ldots$, the (down)chain $b_0 = a_0$, $b_{i+1} = b_i \bar{\sqcap} a_{i+1}$ eventually stabilizes
- We have: $I^k(\vec{x}) \sqsupseteq H^k(\vec{x}) = F^k(\vec{x}) \sqsupseteq F^{k+1}(\vec{x})$.
  - I.e., $I^k(\vec{x})$ greater (valid approx.) than a solution.

For interval analysis

- Preserve (finite) interval bounds: $[l_1, u_1] \sqcap [l_2, u_2] := [l, u]$, where

$$l := \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}$$

For interval analysis

- Preserve (finite) interval bounds: $[l_1, u_1] \sqcap [l_2, u_2] := [l, u]$, where

$$l := \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}$$

- Check:

For interval analysis

- Preserve (finite) interval bounds: $[l_1, u_1] \mathbin{\overline{\sqcap}} [l_2, u_2] := [l, u]$, where

$$l := \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}$$

- Check:
  - $[l_1, u_1] \sqcap [l_2, u_2] \sqsubseteq [l_1, u_1] \mathbin{\overline{\sqcap}} [l_2, u_2] \sqsubseteq [l_1, u_1]$

For interval analysis

- Preserve (finite) interval bounds: $[l_1, u_1] \bar{\sqcap} [l_2, u_2] := [l, u]$, where

$$l := \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}$$

- Check:
  - $[l_1, u_1] \sqcap [l_2, u_2] \sqsubseteq [l_1, u_1] \bar{\sqcap} [l_2, u_2] \sqsubseteq [l_1, u_1]$
  - Stabilizes after at most two narrowing steps

For interval analysis

- Preserve (finite) interval bounds: $[l_1, u_1] \sqcap\!\!\!\!\!\!\sqcap [l_2, u_2] := [l, u]$, where

$$l := \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}$$

- Check:
  - $[l_1, u_1] \sqcap [l_2, u_2] \sqsubseteq [l_1, u_1] \sqcap\!\!\!\!\!\!\sqcap [l_2, u_2] \sqsubseteq [l_1, u_1]$
  - Stabilizes after at most two narrowing steps
- $\sqcap\!\!\!\!\!\!\sqcap$ is not commutative

For interval analysis

- Preserve (finite) interval bounds: $[l_1, u_1] \bar{\sqcap} [l_2, u_2] := [l, u]$, where

$$l := \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases}$$

$$u := \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}$$

- Check:
  - $[l_1, u_1] \sqcap [l_2, u_2] \sqsubseteq [l_1, u_1] \bar{\sqcap} [l_2, u_2] \sqsubseteq [l_1, u_1]$
  - Stabilizes after at most two narrowing steps
- $\bar{\sqcap}$ is not commutative
- For our example: Same result as non-accelerated narrowing!

Discussion

- Narrowing only works for monotonic functions
  - Widening worked for all functions

Discussion

- Narrowing only works for monotonic functions
    - Widening worked for all functions
- Accelerated narrowing can be iterated until stabilization

Discussion

- Narrowing only works for monotonic functions
  - Widening worked for all functions
- Accelerated narrowing can be iterated until stabilization
- However: Design of good widening/narrowing remains black magic

Last Lecture

- Interval analysis (ctd)
  - Abstract values: Intervals $[l, u]$ with $l \leq u$, $l \in \mathbb{Z}_{-\infty}$, $u \in \mathbb{Z}^{+\infty}$
  - Abstract operators: Interval arithmetic

Last Lecture

- Interval analysis (ctd)
  - Abstract values: Intervals $[l, u]$ with $l \leq u$, $l \in \mathbb{Z}_{-\infty}$, $u \in \mathbb{Z}^{+\infty}$
  - Abstract operators: Interval arithmetic
- Main problem: Infinite ascending chains
  - Analysis not guaranteed to terminate

Last Lecture

- Interval analysis (ctd)
  - Abstract values: Intervals $[l, u]$ with $l \leq u$, $l \in \mathbb{Z}_{-\infty}$, $u \in \mathbb{Z}^{+\infty}$
  - Abstract operators: Interval arithmetic
- Main problem: Infinite ascending chains
  - Analysis not guaranteed to terminate
- Widening: Accelerate convergence by over-approximating join
  - Here: Update interval bounds to $-\infty$/$+\infty$

Last Lecture

- Interval analysis (ctd)
  - Abstract values: Intervals $[l, u]$ with $l \leq u$, $l \in \mathbb{Z}_{-\infty}$, $u \in \mathbb{Z}^{+\infty}$
  - Abstract operators: Interval arithmetic
- Main problem: Infinite ascending chains
  - Analysis not guaranteed to terminate
- Widening: Accelerate convergence by over-approximating join
  - Here: Update interval bounds to $-\infty$/$+\infty$
- Problem: makes analysis imprecise
  - Idea 1: Widening only at loop separators
  - Idea 2: Narrowing
    - FP-Iteration on solution preserves solution
    - But may make it smaller
  - Accelerated narrowing:
  - Use narrowing operator for update, that lies "in between" $\sqcap$ and original value
    - ... and converges within finite time
    - Here: Keep finite interval bounds

Recipe: Abstract Interpretation (I)

- Define abstract value domain $\mathbb{A}$, with partial order $\sqsubseteq$
  - $\sqcup$ must be totally defined ($\sqcap$ need not always exists)
- Define description relation between values: $\Delta \subseteq \mathbb{Z} \times \mathbb{A}$
  - Show: Monotonicity: $\forall a_1 \sqsubseteq a_2, v.\ v\ \Delta\ a_1 \implies v\ \Delta\ a_2$
  - Standard: Lift to valuations $(\mathrm{Reg} \to \mathbb{A})$, domain $(\mathbb{D} := (\mathrm{Reg} \to \mathbb{A}) \cup \{\bot\})$
  - Define abstract operators $v^\# : \mathbb{A}, \square^\# : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$, etc.
    - Show soundness wrt. concrete ones:

$$\forall c \in \mathbb{Z}.\ v\ \Delta\ v^\#$$

$$\forall v_1, v_2 \in \mathbb{Z},\ d_1, d_2 \in \mathbb{A}.\ v_1\ \Delta\ d_1 \wedge v_2\ \Delta\ d_2 \implies v_1 \square v_2\ \Delta\ d_1 \square^\# d_2$$

    - For free: $\rho\ \Delta\ \rho^\# \implies [\![e]\!]\rho\ \Delta\ [\![e]\!]^\# \rho^\#$
- Define transformation tr $:: \mathrm{Act} \times \mathbb{D} \to \mathrm{Act}$
  - Show correctness: $(\rho, \mu)\ \Delta\ d \implies [\![a]\!](\rho, \mu) = [\![\mathrm{tr}(a, d)]\!](\rho, \mu)$
- Define abstract effects $[\![\cdot]\!]^\# : \mathrm{Act} \to \mathbb{D} \to \mathbb{D}$, initial value $d_0 \in \mathbb{D}$
  - Usually: Creativity only required on Pos,Neg
  - Show: Monotonicity: $\forall d_1 \sqsubseteq d_2, a.\ [\![a]\!]^\# d_1 \sqsubseteq [\![a]\!]^\# d_2$ and simulation:

$$\forall \rho, \mu.\ (\rho, \mu)\ \Delta\ d_0$$

$$\forall (\rho, \mu) \in \mathrm{dom}([\![a]\!]), d.\ (\rho, \mu)\ \Delta\ d \implies [\![a]\!](\rho, \mu)\ \Delta\ [\![a]\!]^\# d$$

Recipe: Abstract Interpretation (II)

- Check finite chain height of domain
  - Finite: Done
  - Infinite (or too high)
    - Define widening, narrowing operator

Short recapture of methods so far

- Operational semantics on flowgraphs

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation
  - $\mathrm{MOP}[u]$ — Abstract effects reachable at u

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation
  - $\mathrm{MOP}[u]$ — Abstract effects reachable at u
  - Special case: abstract interpretation — domain describes abstract values

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation
  - $\mathrm{MOP}[u]$ — Abstract effects reachable at u
  - Special case: abstract interpretation — domain describes abstract values
- Transformation: Must be compatible with states/leaving paths described by abstract effects

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation
  - $\mathrm{MOP}[u]$ — Abstract effects reachable at u
  - Special case: abstract interpretation — domain describes abstract values
- Transformation: Must be compatible with states/leaving paths described by abstract effects
- Computing analysis result

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation
  - $\mathrm{MOP}[u]$ — Abstract effects reachable at u
  - Special case: abstract interpretation — domain describes abstract values
- Transformation: Must be compatible with states/leaving paths described by abstract effects
- Computing analysis result
  - Constraint system. For monotonic abstract effects. Precise if distributive.

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation
  - $\mathrm{MOP}[u]$ — Abstract effects reachable at u
  - Special case: abstract interpretation — domain describes abstract values
- Transformation: Must be compatible with states/leaving paths described by abstract effects
- Computing analysis result
  - Constraint system. For monotonic abstract effects. Precise if distributive.
  - Solving algorithms: Naive iteration, RR-iteration, worklist algorithm

Short recapture of methods so far

- Operational semantics on flowgraphs
  - Edges have effect on states. Extend to paths.
  - Collecting semantics: $[\![u]\!]$ — States reachable at u.
- Program analysis
  - Abstract description of
    - Forward: States reachable at u
    - Backward: Executions leaving u
  - Abstract effects of edges:
    - Must be compatible with concrete effects
    - Forward: Simulation; Backward: Also (kind of) simulation
  - $\mathrm{MOP}[u]$ — Abstract effects reachable at u
  - Special case: abstract interpretation — domain describes abstract values
- Transformation: Must be compatible with states/leaving paths described by abstract effects
- Computing analysis result
  - Constraint system. For monotonic abstract effects. Precise if distributive.
  - Solving algorithms: Naive iteration, RR-iteration, worklist algorithm
- Forcing convergence: Widening, Narrowing

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \,\Delta\, D$ iff $\forall x \in D, v.\ [\![\pi]\!](\rho(x := v), \mu) = [\![\pi]\!](\rho, \mu)$

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \;\Delta\; D$ iff $\forall x \in D, v.\ [\![\pi]\!](\rho(x := v), \mu) = [\![\pi]\!](\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \;\Delta\; D_0$
  2. $[\![a]\!](\rho, \mu), \pi \;\Delta\; D \implies (\rho, \mu), a\pi \;\Delta\; [\![a]\!]^{\#} D$

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \,\Delta\, D$ iff $\forall x \in D, v.\ [\![\pi]\!](\rho(x := v), \mu) = [\![\pi]\!](\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \,\Delta\, D_0$
  2. $[\![a]\!](\rho, \mu), \pi \,\Delta\, D \implies (\rho, \mu), a\pi \,\Delta\, [\![a]\!]^{\#} D$
- Yields: $\forall \rho, \mu.\ (\rho, \mu), \pi \,\Delta\, [\![\pi]\!]^{\#} D_0$
  - Note: Could even restrict to reachable states ρ, μ.

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \,\Delta\, D$ iff $\forall x \in D, v.\ [\![\pi]\!](\rho(x := v), \mu) = [\![\pi]\!](\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \,\Delta\, D_0$
  2. $[\![a]\!](\rho, \mu), \pi \,\Delta\, D \implies (\rho, \mu), a\pi \,\Delta\, [\![a]\!]^{\#} D$
- Yields: $\forall \rho, \mu.\ (\rho, \mu), \pi \,\Delta\, [\![\pi]\!]^{\#} D_0$
  - Note: Could even restrict to reachable states ρ, μ.

$$\begin{tikzcd}
(\rho_n, \mu_n), \varepsilon \arrow[d, dash, "\Delta"] \\
D_0
\end{tikzcd}$$

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \;\Delta\; D$ iff $\forall x \in D, v.\ [\![\pi]\!](\rho(x := v), \mu) = [\![\pi]\!](\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \;\Delta\; D_0$
  2. $[\![a]\!](\rho, \mu), \pi \;\Delta\; D \implies (\rho, \mu), a\pi \;\Delta\; [\![a]\!]^{\#} D$
- Yields: $\forall \rho, \mu.\ (\rho, \mu), \pi \;\Delta\; [\![\pi]\!]^{\#} D_0$
  - Note: Could even restrict to reachable states ρ, μ.

$$
\begin{tikzcd}
(\rho_{n-1}, \mu_{n-1}), a \arrow[r, "{[\![a]\!]}"] & (\rho_n, \mu_n), \varepsilon \arrow[d, dash, "\Delta"] \\
D_1 & D_0 \arrow[l, "{[\![a]\!]^{\#}}"']
\end{tikzcd}
$$

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \ \Delta \ D$ iff $\forall x \in D, v.\ [\![\pi]\!](\rho(x := v), \mu) = [\![\pi]\!](\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \ \Delta \ D_0$
  2. $[\![a]\!](\rho, \mu), \pi \ \Delta \ D \implies (\rho, \mu), a\pi \ \Delta \ [\![a]\!]^{\#} D$
- Yields: $\forall \rho, \mu.\ (\rho, \mu), \pi \ \Delta \ [\![\pi]\!]^{\#} D_0$
  - Note: Could even restrict to reachable states ρ, μ.

$$
\begin{tikzcd}
(\rho_{n-1}, \mu_{n-1}), a \arrow[r, "{[\![a]\!]}"] \arrow[d, dash, "\Delta"] & (\rho_n, \mu_n), \varepsilon \arrow[d, dash, "\Delta"] \\
D_1 & D_0 \arrow[l, "{[\![a]\!]^{\#}}"]
\end{tikzcd}
$$

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \;\Delta\; D$ iff $\forall x \in D, v.\; \llbracket \pi \rrbracket(\rho(x := v), \mu) = \llbracket \pi \rrbracket(\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \;\Delta\; D_0$
  2. $\llbracket a \rrbracket(\rho, \mu), \pi \;\Delta\; D \implies (\rho, \mu), a\pi \;\Delta\; \llbracket a \rrbracket^{\#} D$
- Yields: $\forall \rho, \mu.\; (\rho, \mu), \pi \;\Delta\; \llbracket \pi \rrbracket^{\#} D_0$
  - Note: Could even restrict to reachable states ρ, μ.

$$
\begin{tikzcd}
\cdots \arrow[r] & (\rho_{n-1}, \mu_{n-1}), a \arrow[r, "\llbracket a \rrbracket"] \arrow[d, dash, "\Delta"] & (\rho_n, \mu_n), \varepsilon \arrow[d, dash, "\Delta"] \\
\cdots & D_1 \arrow[l] & D_0 \arrow[l, "\llbracket a \rrbracket^{\#}"']
\end{tikzcd}
$$

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \; \Delta \; D$ iff $\forall x \in D, v.\ \llbracket \pi \rrbracket (\rho(x := v), \mu) = \llbracket \pi \rrbracket (\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \; \Delta \; D_0$
  2. $\llbracket a \rrbracket (\rho, \mu), \pi \; \Delta \; D \implies (\rho, \mu), a\pi \; \Delta \; \llbracket a \rrbracket^{\#} D$
- Yields: $\forall \rho, \mu.\ (\rho, \mu), \pi \; \Delta \; \llbracket \pi \rrbracket^{\#} D_0$
  - Note: Could even restrict to reachable states ρ, μ.

$$
\begin{tikzcd}
(\rho_0, \mu_0), \pi \arrow[r] & \cdots \arrow[r] & (\rho_{n-1}, \mu_{n-1}), a \arrow[r, "\llbracket a \rrbracket"] \arrow[d, dash, "\Delta"] & (\rho_n, \mu_n), \varepsilon \arrow[d, dash, "\Delta"] \\
D_n & \cdots \arrow[l] & D_1 \arrow[l] & D_0 \arrow[l, "\llbracket a \rrbracket^{\#}"']
\end{tikzcd}
$$

Remark: Simulation (Backwards)

- Describe execution to end node (state, path)
- Dead variables: Execution does not depend on dead variables
  - $(\rho, \mu), \pi \;\Delta\; D$ iff $\forall x \in D, v.\ [\![\pi]\!](\rho(x := v), \mu) = [\![\pi]\!](\rho, \mu)$
- Proof obligations
  1. $(\rho, \mu), \varepsilon \;\Delta\; D_0$
  2. $[\![a]\!](\rho, \mu), \pi \;\Delta\; D \implies (\rho, \mu), a\pi \;\Delta\; [\![a]\!]^{\#} D$
- Yields: $\forall \rho, \mu.\ (\rho, \mu), \pi \;\Delta\; [\![\pi]\!]^{\#} D_0$
  - Note: Could even restrict to reachable states ρ, μ.

$$\begin{tikzcd}
(\rho_0, \mu_0), \pi \arrow[r] \arrow[d, dash, "\Delta"] & \cdots \arrow[r] & (\rho_{n-1}, \mu_{n-1}), a \arrow[r, "{[\![a]\!]}"] \arrow[d, dash, "\Delta"] & (\rho_n, \mu_n), \varepsilon \arrow[d, dash, "\Delta"] \\
D_n & \cdots \arrow[l] & D_1 \arrow[l] & D_0 \arrow[l, "{[\![a]\!]^{\#}}"]
\end{tikzcd}$$

Table of Contents

Motivation

- Want to consider memory

Motivation

- Want to consider memory
- E.g. `M[y] = 5; x = M[y] + 1` $\mapsto$ `M[y] = 5; x=6`

Motivation

- Want to consider memory
- E.g. `M[y] = 5; x = M[y] + 1` $\mapsto$ `M[y] = 5; x=6`
- Here: Assume analyzed program is the only one who accesses memory

Motivation

- Want to consider memory
- E.g. `M[y] = 5; x = M[y] + 1` $\mapsto$ `M[y] = 5; x=6`
- Here: Assume analyzed program is the only one who accesses memory
  - In reality: Shared variables (interrupts, threads), DMA, memory-mapped hardware, ...

Motivation

- Want to consider memory
- E.g. `M[y] = 5; x = M[y] + 1` $\mapsto$ `M[y] = 5; x=6`
- Here: Assume analyzed program is the only one who accesses memory
  - In reality: Shared variables (interrupts, threads), DMA, memory-mapped hardware, ...
  - Compilers provide, e.g., volatile annotation

First Attempt

- Available expressions:
  - Memorize loads: Load: `x = M[e]` $\mapsto$ { $T_{M[e]}$ `= M[e]; x=`$T_{M[e]}$ }
  - Effects

$$[\![T_e = e]\!]^\# A = [\![A]\!]^\# \cup \{e\} \qquad [\![T_{M[e]} = M[e]]\!]^\# A = [\![A]\!]^\# \cup \{M[e]\}$$

$$[\![x = e]\!]^\# A = [\![A]\!]^\# \setminus \mathrm{Expr}_x \qquad [\![M[e_1] = e_2]\!]^\# A = [\![A]\!]^\# \setminus \mathrm{loads}$$

$$\dots$$

First Attempt

- Available expressions:
  - Memorize loads: Load: `x = M[e]` $\mapsto$ `{` $T_{M[e]}$ `= M[e]; x=`$T_{M[e]}$ `}`
  - Effects

$$
\begin{aligned}
[\![T_e = e]\!]^\# A &= [\![A]\!]^\# \cup \{e\} & \qquad [\![T_{M[e]} = M[e]]\!]^\# A &= [\![A]\!]^\# \cup \{M[e]\} \\
[\![x = e]\!]^\# A &= [\![A]\!]^\# \setminus \mathrm{Expr}_x & \qquad [\![M[e_1] = e_2]\!]^\# A &= [\![A]\!]^\# \setminus \mathrm{loads} \\
&\ldots
\end{aligned}
$$

- Problem: Need to be conservative on store
  - Store destroys all information about memory

Constant propagation

- Apply constant propagation to addresses?

Constant propagation

- Apply constant propagation to addresses?
- Exact addresses not known at compile time

Constant propagation

- Apply constant propagation to addresses?
- Exact addresses not known at compile time
- Usually, different addresses accessed at same program point

Constant propagation

- Apply constant propagation to addresses?
- Exact addresses not known at compile time
- Usually, different addresses accessed at same program point
  - E.g., iterate over array

Constant propagation

- Apply constant propagation to addresses?
- Exact addresses not known at compile time
- Usually, different addresses accessed at same program point
    - E.g., iterate over array
- Storing at unknown address destroys all information

Last Lecture

- Motivation to consider memory
  - Alias analysis required!
- Changing the semantics of memory
  - Pointers to start of blocks, indexing within blocks
  - No pointer arithmetic
  - Some assumptions about program correctness: Semantics undefined if
    - Program accesses address that has not been allocated
    - Indexes block out of bounds
    - Computes with addresses

Extending semantics by blocked memory

- Organize memory into blocks
  - `p = new(e)` allocates new block of size *e*
  - `x = p[e]` loads cell *e* from block *p*
  - $\texttt{p[e}_1\texttt{] = e}_2$ writes cell e_1 from block *p*

Extending semantics by blocked memory

- Organize memory into blocks
  - `p = new(e)` allocates new block of size *e*
  - `x = p[e]` loads cell *e* from block *p*
  - $\texttt{p[e}_1\texttt{] = e}_2$ writes cell e_1 from block *p*
- Semantics

Extending semantics by blocked memory

- Organize memory into blocks
  - `p = new(e)` allocates new block of size *e*
  - `x = p[e]` loads cell *e* from block *p*
  - $\texttt{p[e}_1\texttt{] = e}_2$ writes cell e_1 from block *p*
- Semantics
  - Value: $\mathrm{Val} = \mathbb{Z} \,\dot{\cup}\, \mathrm{Addr}$
    - Integer values and block addresses

Extending semantics by blocked memory

- Organize memory into blocks
  - `p = new(e)` allocates new block of size *e*
  - `x = p[e]` loads cell *e* from block *p*
  - $\texttt{p[e}_1\texttt{] = e}_2$ writes cell e_1 from block *p*
- Semantics
  - Value: $\mathrm{Val} = \mathbb{Z} \mathbin{\dot\cup} \mathrm{Addr}$
    - Integer values and block addresses
  - Memory described by $\mu : \mathrm{Addr} \rightharpoonup \mathbb{Z} \rightharpoonup \mathrm{Val}$
    - Maps addresses of blocks to arrays of values
    - $\rightharpoonup$ - partial function (Not all addresses/indexes are valid)

Extending semantics by blocked memory

- Organize memory into blocks
  - `p = new(e)` allocates new block of size *e*
  - `x = p[e]` loads cell *e* from block *p*
  - $\texttt{p[e}_1\texttt{] = e}_2$ writes cell e_1 from block *p*
- Semantics
  - Value: $\mathrm{Val} = \mathbb{Z} \dot{\cup} \mathrm{Addr}$
    - Integer values and block addresses
  - Memory described by $\mu : \mathrm{Addr} \rightharpoonup \mathbb{Z} \rightharpoonup \mathrm{Val}$
    - Maps addresses of blocks to arrays of values
    - $\rightharpoonup$ - partial function (Not all addresses/indexes are valid)
  - Assumption: Type correct
    - In reality: Type system

Extending semantics by blocked memory

- Organize memory into blocks
  - `p = new(e)` allocates new block of size *e*
  - `x = p[e]` loads cell *e* from block *p*
  - $\texttt{p[e}_1\texttt{] = e}_2$ writes cell e_1 from block *p*
- Semantics
  - Value: $\mathrm{Val} = \mathbb{Z} \,\dot\cup\, \mathrm{Addr}$
    - Integer values and block addresses
  - Memory described by $\mu : \mathrm{Addr} \rightharpoonup \mathbb{Z} \rightharpoonup \mathrm{Val}$
    - Maps addresses of blocks to arrays of values
    - $\rightharpoonup$ - partial function (Not all addresses/indexes are valid)
  - Assumption: Type correct
    - In reality: Type system
  - We write null and 0 synonymously

Semantics

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho,\mu) &= (\rho,\mu)\\
[\![x = e]\!](\rho,\mu) &= (\rho(x \mapsto [\![e]\!]\rho),\mu)\\
[\![\mathrm{Pos}(e)]\!](\rho,\mu) &= [\![e]\!]\rho \neq 0 ? (\rho,\mu) : \text{undefined}\\
[\![\mathrm{Neg}(e)]\!](\rho,\mu) &= [\![e]\!]\rho = 0 ? (\rho,\mu) : \text{undefined}\\
[\![x = p[e]]\!](\rho,\mu) &= (\rho(x \mapsto \mu([\![p]\!]\rho,[\![e]\!]\rho)),\mu)\\
[\![p[e_1] = e_2]\!](\rho,\mu) &= (\rho,\mu([\![p]\!]\rho,[\![e_1]\!]\rho) \mapsto [\![e_2]\!]\rho)\\
[\![x = \mathrm{new}(e)]\!](\rho,\mu) &= (\rho(x \mapsto a),\mu(a \mapsto (i \mapsto 0 \mid 0 \leq i < [\![e]\!]\rho))) \quad a \notin \mathrm{dom}(\mu)
\end{aligned}
$$

- New initializes the block

Semantics

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho,\mu) &= (\rho,\mu)\\
[\![x = e]\!](\rho,\mu) &= (\rho(x \mapsto [\![e]\!]\rho),\mu)\\
[\![\mathrm{Pos}(e)]\!](\rho,\mu) &= [\![e]\!]\rho \neq 0 ? (\rho,\mu) : \text{undefined}\\
[\![\mathrm{Neg}(e)]\!](\rho,\mu) &= [\![e]\!]\rho = 0 ? (\rho,\mu) : \text{undefined}\\
[\![x = p[e]]\!](\rho,\mu) &= (\rho(x \mapsto \mu([\![p]\!]\rho, [\![e]\!]\rho)),\mu)\\
[\![p[e_1] = e_2]\!](\rho,\mu) &= (\rho, \mu([\![p]\!]\rho, [\![e_1]\!]\rho) \mapsto [\![e_2]\!]\rho)\\
[\![x = \mathrm{new}(e)]\!](\rho,\mu) &= (\rho(x \mapsto a), \mu(a \mapsto (i \mapsto 0 \mid 0 \leq i < [\![e]\!]\rho))) \quad a \notin \mathrm{dom}(\mu
\end{aligned}
$$

- New initializes the block
  - Java: OK, C/C++: ???

Semantics

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho,\mu) &= (\rho,\mu)\\
[\![x = e]\!](\rho,\mu) &= (\rho(x \mapsto [\![e]\!]\rho),\mu)\\
[\![\mathrm{Pos}(e)]\!](\rho,\mu) &= [\![e]\!]\rho \neq 0 ? (\rho,\mu) : \text{undefined}\\
[\![\mathrm{Neg}(e)]\!](\rho,\mu) &= [\![e]\!]\rho = 0 ? (\rho,\mu) : \text{undefined}\\
[\![x = p[e]]\!](\rho,\mu) &= (\rho(x \mapsto \mu([\![p]\!]\rho, [\![e]\!]\rho)),\mu)\\
[\![p[e_1] = e_2]\!](\rho,\mu) &= (\rho, \mu([\![p]\!]\rho, [\![e_1]\!]\rho) \mapsto [\![e_2]\!]\rho)\\
[\![x = \mathrm{new}(e)]\!](\rho,\mu) &= (\rho(x \mapsto a), \mu(a \mapsto (i \mapsto 0 \mid 0 \leq i < [\![e]\!]\rho))) \quad a \notin \mathrm{dom}(\mu)
\end{aligned}
$$

- New initializes the block
  - Java: OK, C/C++: ???
- Assume that only valid addresses are used
  - Otherwise, we formally get undefined

Semantics

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho, \mu) &= (\rho, \mu) \\
[\![x = e]\!](\rho, \mu) &= (\rho(x \mapsto [\![e]\!]\rho), \mu) \\
[\![\mathrm{Pos}(e)]\!](\rho, \mu) &= [\![e]\!]\rho \neq 0 ? (\rho, \mu) : \mathrm{undefined} \\
[\![\mathrm{Neg}(e)]\!](\rho, \mu) &= [\![e]\!]\rho = 0 ? (\rho, \mu) : \mathrm{undefined} \\
[\![x = p[e]]\!](\rho, \mu) &= (\rho(x \mapsto \mu([\![p]\!]\rho, [\![e]\!]\rho)), \mu) \\
[\![p[e_1] = e_2]\!](\rho, \mu) &= (\rho, \mu([\![p]\!]\rho, [\![e_1]\!]\rho) \mapsto [\![e_2]\!]\rho) \\
[\![x = \mathrm{new}(e)]\!](\rho, \mu) &= (\rho(x \mapsto a), \mu(a \mapsto (i \mapsto 0 \mid 0 \leq i < [\![e]\!]\rho))) \quad a \notin \mathrm{dom}(\mu
\end{aligned}
$$

- New initializes the block
  - Java: OK, C/C++: ???
- Assume that only valid addresses are used
  - Otherwise, we formally get undefined
- Assume that no arithmetic on addresses is done

Semantics

$$
\begin{aligned}
[\![\mathrm{Nop}]\!](\rho,\mu) &= (\rho,\mu)\\
[\![x = e]\!](\rho,\mu) &= (\rho(x \mapsto [\![e]\!]\rho),\mu)\\
[\![\mathrm{Pos}(e)]\!](\rho,\mu) &= [\![e]\!]\rho \neq 0 ? (\rho,\mu) : \mathrm{undefined}\\
[\![\mathrm{Neg}(e)]\!](\rho,\mu) &= [\![e]\!]\rho = 0 ? (\rho,\mu) : \mathrm{undefined}\\
[\![x = p[e]]\!](\rho,\mu) &= (\rho(x \mapsto \mu([\![p]\!]\rho,[\![e]\!]\rho)),\mu)\\
[\![p[e_1] = e_2]\!](\rho,\mu) &= (\rho,\mu([\![p]\!]\rho,[\![e_1]\!]\rho) \mapsto [\![e_2]\!]\rho)\\
[\![x = \mathrm{new}(e)]\!](\rho,\mu) &= (\rho(x \mapsto a),\mu(a \mapsto (i \mapsto 0 \mid 0 \leq i < [\![e]\!]\rho))) \quad a \notin \mathrm{dom}(\mu)
\end{aligned}
$$

- New initializes the block
  - Java: OK, C/C++: ???
- Assume that only valid addresses are used
  - Otherwise, we formally get undefined
- Assume that no arithmetic on addresses is done
- Assume infinite supply of addresses

Equivalence

- Note: Semantics does not clearly specify how addresses are allocated

Equivalence

- Note: Semantics does not clearly specify how addresses are allocated
- This is irrelevant, consider e.g.
  `x=new(4); y=new(4)` and `y=new(4); x=new(4)`

Equivalence

- Note: Semantics does not clearly specify how addresses are allocated
- This is irrelevant, consider e.g.
  `x=new(4); y=new(4)` and `y=new(4); x=new(4)`
  - Programs should be equivalent

Equivalence

- Note: Semantics does not clearly specify how addresses are allocated
- This is irrelevant, consider e.g.
  `x=new(4); y=new(4)` and `y=new(4); x=new(4)`
  - Programs should be equivalent
  - Although memory manager would probably assign different physical addresses

Equivalence

- Note: Semantics does not clearly specify how addresses are allocated
- This is irrelevant, consider e.g.
  `x=new(4); y=new(4)` and `y=new(4); x=new(4)`
  - Programs should be equivalent
  - Although memory manager would probably assign different physical addresses
- Two states (ρ, μ) and (ρ', μ') are considered equivalent, iff they are equivalent up to permutation of addresses
  - We write $(\rho, \mu) \equiv (\rho', \mu')$

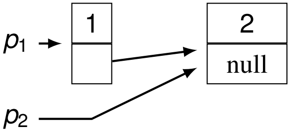
Equivalence

- Note: Semantics does not clearly specify how addresses are allocated
- This is irrelevant, consider e.g.
  `x=new(4); y=new(4)` and `y=new(4); x=new(4)`
  - Programs should be equivalent
  - Although memory manager would probably assign different physical addresses
- Two states (ρ, μ) and (ρ', μ') are considered equivalent, iff they are equivalent up to permutation of addresses
  - We write $(\rho, \mu) \equiv (\rho', \mu')$
- Note: To avoid this nondeterminism in semantics:
  - Choose Addr to be totally ordered
  - Always take the smallest free address

Examples

- Building the linked list $[1, 2]$

```
p₁ = new (2)
p₂ = new (2)
p₁[0] = 1
p₁[1] = p₂
p₂[0] = 2
p₂[1] = null
```



Examples

- List reversal

```
R = null
while (T != null) {
  H = T
  T = T[0]
  H[0] = R
  R = H
}
```

Examples

- List reversal

```
R = null
while (T != null) {
  H = T
  T = T[0]
  H[0] = R
  R = H
}
```

- Sketch algorithm on whiteboard

Alias analysis

- May alias: May two pointers point to the same address
  - On store: Only destroy information for addresses that may alias with stored address

Alias analysis

- May alias: May two pointers point to the same address
  - On store: Only destroy information for addresses that may alias with stored address
- Must alias: Must two pointers point to the same address
  - If so, store to one can update information for the other

Alias analysis

- May alias: May two pointers point to the same address
  - On store: Only destroy information for addresses that may alias with stored address
- Must alias: Must two pointers point to the same address
  - If so, store to one can update information for the other
- Here: Focus on may-alias
  - Important to limit the destructive effect of memory updates
  - Must alias: Usually only done in local scope, by, e.g., copy propagation

First Idea

- Summarize (arbitrarily many) blocks of memory by (fixed number of) allocation sites
  - Use start node of edge in CFG to identify allocation site
  - Abstract values $\mathrm{Addr}^\# = V$, $\mathrm{Val}^\# = 2^{\mathrm{Addr}^\#}$ (Possible targets for pointer)
  - Domain: $(\mathrm{Reg} \to \mathrm{Val}^\#) \times (\mathrm{Addr}^\# \to \mathrm{Val}^\#)$
- Effects

$$
\begin{aligned}
&\dots \\
[\![\_, x = y, \_]\!]^\#(R, M) &= (R(x \mapsto R(y)), M) && \text{for } y \in \mathrm{Reg} \\
[\![\_, x = e, \_]\!]^\#(R, M) &= (R(x \mapsto \emptyset), M) && \text{for } e \notin \mathrm{Reg} \\
[\![u, x = \mathrm{new}(e), v]\!]^\#(R, M) &= (R(x \mapsto \{u\}), M) \\
[\![\_, x = p[e], \_]\!]^\#(R, M) &= (R(x \mapsto \bigcup\{M[a] \mid a \in R[p]\}), M) \\
[\![\_, p[e_1] = y, \_]\!]^\#(R, M) &= (R, M(a \mapsto M(a) \cup R(y) \mid a \in R(p))) && \text{for } y \in \mathrm{Reg} \\
[\![\_, p[e_1] = e, \_]\!]^\#(R, M) &= (R, M) && \text{for } e \notin \mathrm{Reg}
\end{aligned}
$$

First Idea

- Summarize (arbitrarily many) blocks of memory by (fixed number of) allocation sites
  - Use start node of edge in CFG to identify allocation site
  - Abstract values $\mathrm{Addr}^\# = V$, $\mathrm{Val}^\# = 2^{\mathrm{Addr}^\#}$ (Possible targets for pointer)
  - Domain: $(\mathrm{Reg} \to \mathrm{Val}^\#) \times (\mathrm{Addr}^\# \to \mathrm{Val}^\#)$
- Effects

$$
\begin{aligned}
&\ldots \\
[\![\_, \mathbf{x} = \mathbf{y}, \_]\!]^\#(\mathbf{R}, \mathbf{M}) &= (R(x \mapsto R(y)), M) && \text{for } y \in \mathrm{Reg} \\
[\![\_, x = e, \_]\!]^\#(R, M) &= (R(x \mapsto \emptyset), M) && \text{for } e \notin \mathrm{Reg} \\
[\![u, x = \mathrm{new}(e), v]\!]^\#(R, M) &= (R(x \mapsto \{u\}), M) \\
[\![\_, x = p[e], \_]\!]^\#(R, M) &= (R(x \mapsto \bigcup\{M[a] \mid a \in R[p]\}), M) \\
[\![\_, p[e_1] = y, \_]\!]^\#(R, M) &= (R, M(a \mapsto M(a) \cup R(y) \mid a \in R(p))) && \text{for } y \in \mathrm{Reg} \\
[\![\_, p[e_1] = e, \_]\!]^\#(R, M) &= (R, M) && \text{for } e \notin \mathrm{Reg}
\end{aligned}
$$

x may point to addresses where y may point to.

First Idea

- Summarize (arbitrarily many) blocks of memory by (fixed number of) allocation sites
  - Use start node of edge in CFG to identify allocation site
  - Abstract values $\mathrm{Addr}^\# = V$, $\mathrm{Val}^\# = 2^{\mathrm{Addr}^\#}$ (Possible targets for pointer)
  - Domain: $(\mathrm{Reg} \to \mathrm{Val}^\#) \times (\mathrm{Addr}^\# \to \mathrm{Val}^\#)$
- Effects

$$\begin{aligned}
&\dots \\
[\![_, x = y, _]\!]^\#(R, M) &= (R(x \mapsto R(y)), M) && \text{for } y \in \mathrm{Reg} \\
\color{red}{[\![_, \mathbf{x} = \mathbf{e}, _]\!]^\#(\mathbf{R}, \mathbf{M})} &= (R(x \mapsto \emptyset), M) && \text{for } e \notin \mathrm{Reg} \\
[\![u, x = \mathrm{new}(e), v]\!]^\#(R, M) &= (R(x \mapsto \{u\}), M) \\
[\![_, x = p[e], _]\!]^\#(R, M) &= (R(x \mapsto \bigcup\{M[a] \mid a \in R[p]\}), M) \\
[\![_, p[e_1] = y, _]\!]^\#(R, M) &= (R, M(a \mapsto M(a) \cup R(y) \mid a \in R(p))) && \text{for } y \in \mathrm{Reg} \\
[\![_, p[e_1] = e, _]\!]^\#(R, M) &= (R, M) && \text{for } e \notin \mathrm{Reg}
\end{aligned}$$

Expressions are never pointers.

First Idea

- Summarize (arbitrarily many) blocks of memory by (fixed number of) allocation sites
  - Use start node of edge in CFG to identify allocation site
  - Abstract values $\mathrm{Addr}^\# = V$, $\mathrm{Val}^\# = 2^{\mathrm{Addr}^\#}$ (Possible targets for pointer)
  - Domain: $(\mathrm{Reg} \to \mathrm{Val}^\#) \times (\mathrm{Addr}^\# \to \mathrm{Val}^\#)$
- Effects

$$\begin{array}{rcll}
 & \dots & & \\
[\![_, x = y, _]\!]^\#(R, M) & = & (R(x \mapsto R(y)), M) & \text{for } y \in \mathrm{Reg} \\
[\![_, x = e, _]\!]^\#(R, M) & = & (R(x \mapsto \emptyset), M) & \text{for } e \notin \mathrm{Reg} \\
[\![\mathbf{u}, \mathbf{x} = \mathrm{new}(\mathbf{e}), \mathbf{v}]\!]^\#(\mathbf{R}, \mathbf{M}) & = & (R(x \mapsto \{u\}), M) & \\
[\![_, x = p[e], _]\!]^\#(R, M) & = & (R(x \mapsto \bigcup\{M[a] \mid a \in R[p]\}), M) & \\
[\![_, p[e_1] = y, _]\!]^\#(R, M) & = & (R, M(a \mapsto M(a) \cup R(y) \mid a \in R(p))) & \text{for } y \in \mathrm{Reg} \\
[\![_, p[e_1] = e, _]\!]^\#(R, M) & = & (R, M) & \text{for } e \notin \mathrm{Reg}
\end{array}$$

x points to this allocation site.

First Idea

- Summarize (arbitrarily many) blocks of memory by (fixed number of) allocation sites
  - Use start node of edge in CFG to identify allocation site
  - Abstract values $\mathrm{Addr}^\# = V$, $\mathrm{Val}^\# = 2^{\mathrm{Addr}^\#}$ (Possible targets for pointer)
  - Domain: $(\mathrm{Reg} \to \mathrm{Val}^\#) \times (\mathrm{Addr}^\# \to \mathrm{Val}^\#)$
- Effects

$$
\begin{aligned}
&\ldots \\
[\![\_, x = y, \_]\!]^\#(R, M) &= (R(x \mapsto R(y)), M) && \text{for } y \in \mathrm{Reg} \\
[\![\_, x = e, \_]\!]^\#(R, M) &= (R(x \mapsto \emptyset), M) && \text{for } e \notin \mathrm{Reg} \\
[\![u, x = \mathrm{new}(e), v]\!]^\#(R, M) &= (R(x \mapsto \{u\}), M) \\
\boldsymbol{[\![\_, x = p[e], \_]\!]^\#(R, M)} &= (R(x \mapsto \bigcup\{M[a] \mid a \in R[p]\}), M) \\
[\![\_, p[e_1] = y, \_]\!]^\#(R, M) &= (R, M(a \mapsto M(a) \cup R(y) \mid a \in R(p))) && \text{for } y \in \mathrm{Reg} \\
[\![\_, p[e_1] = e, \_]\!]^\#(R, M) &= (R, M) && \text{for } e \notin \mathrm{Reg}
\end{aligned}
$$

x may point to everything that *a* may point to, for *p* pointing to *a*

First Idea

- Summarize (arbitrarily many) blocks of memory by (fixed number of) allocation sites
  - Use start node of edge in CFG to identify allocation site
  - Abstract values $\mathrm{Addr}^\# = V$, $\mathrm{Val}^\# = 2^{\mathrm{Addr}^\#}$ (Possible targets for pointer)
  - Domain: $(\mathrm{Reg} \to \mathrm{Val}^\#) \times (\mathrm{Addr}^\# \to \mathrm{Val}^\#)$
- Effects

$$
\begin{array}{rcll}
 & \dots & & \\
[\![\_, x = y, \_]\!]^\#(R, M) & = & (R(x \mapsto R(y)), M) & \text{for } y \in \mathrm{Reg} \\
[\![\_, x = e, \_]\!]^\#(R, M) & = & (R(x \mapsto \emptyset), M) & \text{for } e \notin \mathrm{Reg} \\
[\![u, x = \mathrm{new}(e), v]\!]^\#(R, M) & = & (R(x \mapsto \{u\}), M) & \\
[\![\_, x = p[e], \_]\!]^\#(R, M) & = & (R(x \mapsto \bigcup\{M[a] \mid a \in R[p]\}), M) & \\
\boldsymbol{[\![\_, p[e_1] = y, \_]\!]^\#(R, M)} & = & (R, M(a \mapsto M(a) \cup R(y) \mid a \in R(p))) & \text{for } y \in \mathrm{Reg} \\
[\![\_, p[e_1] = e, \_]\!]^\#(R, M) & = & (R, M) & \text{for } e \notin \mathrm{Reg}
\end{array}
$$

Add addresses from y to each possible address of p

First Idea

- Summarize (arbitrarily many) blocks of memory by (fixed number of) allocation sites
  - Use start node of edge in CFG to identify allocation site
  - Abstract values $\mathrm{Addr}^\# = V$, $\mathrm{Val}^\# = 2^{\mathrm{Addr}^\#}$ (Possible targets for pointer)
  - Domain: $(\mathrm{Reg} \to \mathrm{Val}^\#) \times (\mathrm{Addr}^\# \to \mathrm{Val}^\#)$
- Effects

$$
\begin{array}{rcll}
 & \dots & & \\
[\![\_, x = y, \_]\!]^\#(R, M) &=& (R(x \mapsto R(y)), M) & \text{for } y \in \mathrm{Reg} \\
[\![\_, x = e, \_]\!]^\#(R, M) &=& (R(x \mapsto \emptyset), M) & \text{for } e \notin \mathrm{Reg} \\
[\![u, x = \mathrm{new}(e), v]\!]^\#(R, M) &=& (R(x \mapsto \{u\}), M) & \\
[\![\_, x = p[e], \_]\!]^\#(R, M) &=& (R(x \mapsto \bigcup\{M[a] \mid a \in R[p]\}), M) & \\
[\![\_, p[e_1] = y, \_]\!]^\#(R, M) &=& (R, M(a \mapsto M(a) \cup R(y) \mid a \in R(p))) & \text{for } y \in \mathrm{Reg} \\
\color{red}{[\![\_, \mathbf{p}[\mathbf{e_1}] = \mathbf{e}, \_]\!]^\#(\mathbf{R}, \mathbf{M})} &=& (R, M) & \text{for } e \notin \mathrm{Reg}
\end{array}
$$

Expressions are never pointers.

Example

```
u: p₁ = new (2)
v: p₂ = new (2)
p₁[0] = 1
p₁[1] = p₂
p₂[0] = 2
p₂[1] = null
```

- At end of program, we have

$$R = p_1 \mapsto \{u\}, p_2 \mapsto \{v\}$$
$$M = u \mapsto \{v\}, v \mapsto \{\}$$

Description Relation

$$(\rho, \mu) \; \Delta \; (R, M) \text{ iff } \exists s : \mathrm{Addr} \to V.\, \forall a, a' \in \mathrm{Addr}.\, \forall x, i.$$

$$\rho(x) = a \implies s(a) \in R(x) \quad (1)$$

$$\wedge\, \mu(a, i) = a' \implies s(a') \in M(s(a)) \quad (2)$$

Intuitively: There is a mapping s from addresses to allocation sites, with:

(1) If a register contains an address, its abstract value contains the corresponding allocation site

(2) If a memory block contains an address (at any index), its abstract value contains the corresponding allocation site

Description Relation

$$(\rho, \mu) \, \Delta \, (R, M) \text{ iff } \exists s : \mathrm{Addr} \to V. \, \forall a, a' \in \mathrm{Addr}. \, \forall x, i.$$

$$\rho(x) = a \implies s(a) \in R(x) \tag{1}$$

$$\wedge \, \mu(a, i) = a' \implies s(a') \in M(s(a)) \tag{2}$$

Intuitively: There is a mapping s from addresses to allocation sites, with:

(1) If a register contains an address, its abstract value contains the corresponding allocation site

(2) If a memory block contains an address (at any index), its abstract value contains the corresponding allocation site

From this, we can extract may-alias information: Pointers p_1, p_2 may only alias (i.e., $\rho(p_1) = \rho(p_2) \in \mathrm{Addr}$), if $R(p_1) \cap R(p_2) \neq \emptyset$.

- B/c if $\rho(p_1) = \rho(p_2) = a \in \mathrm{Addr}$, we have $s(a) \in R(p_1) \cap R(p_2)$

Description Relation

$$(\rho, \mu) \; \Delta \; (R, M) \text{ iff } \exists s : \mathrm{Addr} \to V. \; \forall a, a' \in \mathrm{Addr}. \; \forall x, i.$$

$$\rho(x) = a \implies s(a) \in R(x) \tag{1}$$

$$\wedge \; \mu(a, i) = a' \implies s(a') \in M(s(a)) \tag{2}$$

Intuitively: There is a mapping s from addresses to allocation sites, with:

(1) If a register contains an address, its abstract value contains the corresponding allocation site

(2) If a memory block contains an address (at any index), its abstract value contains the corresponding allocation site

From this, we can extract may-alias information: Pointers p_1, p_2 may only alias (i.e., $\rho(p_1) = \rho(p_2) \in \mathrm{Addr}$), if $R(p_1) \cap R(p_2) \neq \emptyset$.

- B/c if $\rho(p_1) = \rho(p_2) = a \in \mathrm{Addr}$, we have $s(a) \in R(p_1) \cap R(p_2)$

Correctness of abstract effects (sketch)

- On whiteboard

Discussion

- May-point-to information accumulates for store.
    - If store is not initialized, we find out nothing

Discussion

- May-point-to information accumulates for store.
  - If store is not initialized, we find out nothing
- Analysis can be quite expensive
  - Abstract representation of memory at each program point
  - Does not scale to large programs

Flow insensitive analysis

- Idea: Do not consider ordering of statements

Flow insensitive analysis

- Idea: Do not consider ordering of statements
- Compute information that holds for any program point

Flow insensitive analysis

- Idea: Do not consider ordering of statements
- Compute information that holds for any program point
  - Only one instance of abstract registers/memory needed

Flow insensitive analysis

- Idea: Do not consider ordering of statements
- Compute information that holds for any program point
  - Only one instance of abstract registers/memory needed
- For our simple example: No loss in precision

First attempt

- Each edge (u, a, v) gives rise to constraints

a	constraints
$x = y$	$R(x) \supseteq R(y)$
$x = \mathrm{new}(e)$	$R(x) \supseteq \{u\}$
$x = p[e]$	$R(x) \supseteq \bigcup\{M(a) \mid a \in R(p)\}$
$p[e_1] = x$	$M(a) \supseteq (a \in R(p) ? R(x) : \emptyset)$ for all $a \in V$

First attempt

- Each edge (u, a, v) gives rise to constraints

a	constraints
$x = y$	$R(x) \supseteq R(y)$
$x = \mathrm{new}(e)$	$R(x) \supseteq \{u\}$
$x = p[e]$	$R(x) \supseteq \bigcup\{M(a) \mid a \in R(p)\}$
$p[e_1] = x$	$M(a) \supseteq (a \in R(p) ? R(x) : \emptyset)$ for all $a \in V$

- Other edges have no effect

First attempt

- Each edge (u, a, v) gives rise to constraints

a	constraints
$x = y$	$R(x) \supseteq R(y)$
$x = \mathrm{new}(e)$	$R(x) \supseteq \{u\}$
$x = p[e]$	$R(x) \supseteq \bigcup\{M(a) \mid a \in R(p)\}$
$p[e_1] = x$	$M(a) \supseteq (a \in R(p) ? R(x) : \emptyset)$ for all $a \in V$

- Other edges have no effect
- Problem: Too many constraints
  - $O(kn)$ for k allocation sites and n edges.

First attempt

- Each edge (u, a, v) gives rise to constraints

a	constraints
$x = y$	$R(x) \supseteq R(y)$
$x = \mathrm{new}(e)$	$R(x) \supseteq \{u\}$
$x = p[e]$	$R(x) \supseteq \bigcup\{M(a) \mid a \in R(p)\}$
$p[e_1] = x$	$M(a) \supseteq (a \in R(p) ? R(x) : \emptyset)$ for all $a \in V$

- Other edges have no effect
- Problem: Too many constraints
  - $O(kn)$ for k allocation sites and n edges.
- Does not scale to big programs

Last Lecture

- Flow sensitive points-to analysis
  - Identify blocks in memory with allocation sites
  - Does not scale. One abstract memory per program point.
- Flow-insensitive points-to analysis
  - Compute one abstract memory that approximates all program points.
  - Does not scale. Too many constraints
- Flow-insensitive alias analysis
  - Compute equivalence classes of p and $p[]$

Alias analysis

- Idea: Maintain equivalence relation between variables p and memory accesses $p[]$
  - $x \sim y$ whenever x and y may contain the same address (at any two program points)

```
u: p1 = new (2)
v: p2 = new (2)
p1[0] = 1
p1[1] = p2
p2[0] = 2
p2[1] = null
```

- $\sim = \{\{p_1[], p_2\}, \{p_1\}, \{p_2[]\}\}$

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)
  - The set of all equivalence relations on R with $\subseteq$ forms a complete lattice

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)
  - The set of all equivalence relations on R with $\subseteq$ forms a complete lattice
  - $\sim_\perp :=$

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)
  - The set of all equivalence relations on R with $\subseteq$ forms a complete lattice
  - $\sim_\bot := (=)$

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)
  - The set of all equivalence relations on R with $\subseteq$ forms a complete lattice
  - $\sim_\perp := (=)$
  - $\sim_\top :=$

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)
  - The set of all equivalence relations on R with $\subseteq$ forms a complete lattice
  - $\sim_\bot := (=)$
  - $\sim_\top := R \times R$

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)
  - The set of all equivalence relations on R with $\subseteq$ forms a complete lattice
  - $\sim_\bot := (=)$
  - $\sim_\top := R \times R$
  - $\bigsqcup S :=$

Equivalence relations

- Relation $\sim \subseteq R \times R$ that is reflexive, transitive, symmetric
- Equivalence class $[p] := \{p' \in R \mid p \sim p'\}$
- The equivalence classes partition R. Conversely, any partition of R defines an equivalence relation.
- $\sim \subseteq \sim'$ ($\sim$ finer than $\sim'$)
  - The set of all equivalence relations on R with $\subseteq$ forms a complete lattice
  - $\sim_\perp := (=)$
  - $\sim_\top := R \times R$
  - $\bigsqcup S := (\bigcup S)^*$

Operations on ERs

- `find(~,p)`: Return equivalence class of *p*

Operations on ERs

- `find(~,p)`: Return equivalence class of p
- **`union`**`(~,p,p')`: Return finest ER $\sim'$ with $p \sim' p'$ and $\sim \subseteq \sim'$

Operations on ERs

- `find(~,p)`: Return equivalence class of p
- **`union`**`(~,p,p')`: Return finest ER $\sim'$ with $p \sim' p'$ and $\sim \subseteq \sim'$
- On partitions of finite sets: Let $R = [p_1]_\sim \dot\cup \ldots \dot\cup [p_n]_\sim$
  - **`union`**`(~,p,p')`: Let $p \in [p_i]_\sim, p' \in [p_j]_\sim$
    Result: $\{[p_i]_\sim \cup [p_j]_\sim\} \dot\cup \{[p_k] \mid 1 \leq k \leq n \wedge k \notin \{i,j\}\}$

Recursive Union

- If $x \sim y$, then also $x[] \sim y[]$ (rec)

Recursive Union

- If $x \sim y$, then also $x[] \sim y[]$ (rec)
- After union, we have to add those equivalences!

Recursive Union

- If $x \sim y$, then also $x[] \sim y[]$ (rec)
- After union, we have to add those equivalences!
- `union*(~,p,p')`:
  - The finest ER that is coarser than `union(~,p,p')` and satisfies (rec)

Alias analysis

```
π = { {x}, {x[]} | x ∈ Vars } // Finest ER

for (_,a,_) in E do {
  case a of
    x=y: π = union*(π,x,y)
  | x=y[e]: π = union*(π,x,y[]) // y variable
  | y[e]=x: π = union*(π,x,y[]) // y variable
}
```

- Start with finest ER (=)

Alias analysis

```
π = { {x}, {x[]} | x ∈ Vars } // Finest ER

for (_,a,_) in E do {
  case a of
    x=y: π = union*(π,x,y)
  | x=y[e]: π = union*(π,x,y[]) // y variable
  | y[e]=x: π = union*(π,x,y[]) // y variable
}
```

- Start with finest ER (=)
- Iterate over edges, and union equivalence classes

Example

```
1: p₁ = new (2)
2: p₂ = new (2)
3: p₁[0] = 1
4: p₁[1] = p₂
5: p₂[0] = 2
6: p₂[1] = null
```

init	$\{\{p_1\}, \{p_2\}, \{p_1[]\}, \{p_2[]\}\}$
$1 \to 2$	$\{\{p_1\}, \{p_2\}, \{p_1[]\}, \{p_2[]\}\}$
$2 \to 3$	$\{\{p_1\}, \{p_2\}, \{p_1[]\}, \{p_2[]\}\}$
$3 \to 4$	$\{\{p_1\}, \{p_2\}, \{p_1[]\}, \{p_2[]\}\}$
$4 \to 5$	$\{\{p_1\}, \{p_2, p_1[]\}, \{p_2[]\}\}$
$5 \to 6$	$\{\{p_1\}, \{p_2, p_1[]\}, \{p_2[]\}\}$

Example

```
1: R = null
2: if Neg (T != null) goto 8
3:   H = T
4:   T = T[0]
5:   H[0] = R
6:   R = H
7: goto 2
8:
```

init	$\{\{H\},\{R\},\{T\},\{H[]\},\{T[]\}\}$
$3 \to 4$	$\{\{H,T\},\{R\},\{H[],T[]\}\}$
$4 \to 5$	$\{\{H,T,H[],T[]\},\{R\}\}$
$5 \to 6$	$\{\{H,T,H[],T[],R\}\}$
$6 \to 7$	$\{\{H,T,H[],T[],R\}\}$

Discussion

- All memory content must have been constructed by analyzed program

Discussion

- All memory content must have been constructed by analyzed program
  - `p=p[]; p=p[]; q=q[]`

Discussion

- All memory content must have been constructed by analyzed program
  - `p=p[]; p=p[]; q=q[]`
  - What if *q* points to third element of linked list at *p*.

Discussion

- All memory content must have been constructed by analyzed program
  - `p=p[]; p=p[]; q=q[]`
  - What if *q* points to third element of linked list at *p*.

$\Longrightarrow$ Only works for whole programs, no input via memory

Correctness

- Intuition: Each address ever created represented by register

Correctness

- Intuition: Each address ever created represented by register
- Invariant:

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
    1. If register holds address, it is in the same class as address' representative

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced
- Formally: For all reachable states (ρ, μ), there exists a map $m : \mathrm{Addr} \to \mathrm{Reg}$, such that

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced
- Formally: For all reachable states (ρ, μ), there exists a map $m : \mathrm{Addr} \to \mathrm{Reg}$, such that
  1. $\rho(x) \in \mathrm{Addr} \implies x \sim m(\rho(x))$

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced
- Formally: For all reachable states (ρ, μ), there exists a map $m : \mathrm{Addr} \to \mathrm{Reg}$, such that
  1. $\rho(x) \in \mathrm{Addr} \implies x \sim m(\rho(x))$
  2. $\mu(a, i) \in \mathrm{Addr} \implies m(a)[] \sim m(\mu(a, i))$

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced
- Formally: For all reachable states (ρ, μ), there exists a map $m : \mathrm{Addr} \to \mathrm{Reg}$, such that
  1. $\rho(x) \in \mathrm{Addr} \implies x \sim m(\rho(x))$
  2. $\mu(a, i) \in \mathrm{Addr} \implies m(a)[] \sim m(\mu(a, i))$
- Extracting alias information: x, y may alias, if $x \sim y$.
  - $\rho(x) = \rho(y) = a \in \mathrm{Addr} \implies x \sim m(a) \sim y$

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced
- Formally: For all reachable states (ρ, μ), there exists a map $m : \mathrm{Addr} \to \mathrm{Reg}$, such that
  1. $\rho(x) \in \mathrm{Addr} \implies x \sim m(\rho(x))$
  2. $\mu(a, i) \in \mathrm{Addr} \implies m(a)[] \sim m(\mu(a, i))$
- Extracting alias information: x, y may alias, if $x \sim y$.
  - $\rho(x) = \rho(y) = a \in \mathrm{Addr} \implies x \sim m(a) \sim y$
- To show: Invariant holds initially, and preserved by steps

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced
- Formally: For all reachable states (ρ, μ), there exists a map $m : \mathrm{Addr} \to \mathrm{Reg}$, such that
  1. $\rho(x) \in \mathrm{Addr} \implies x \sim m(\rho(x))$
  2. $\mu(a, i) \in \mathrm{Addr} \implies m(a)[] \sim m(\mu(a, i))$
- Extracting alias information: x, y may alias, if $x \sim y$.
  - $\rho(x) = \rho(y) = a \in \mathrm{Addr} \implies x \sim m(a) \sim y$
- To show: Invariant holds initially, and preserved by steps
  - Initially: By assumption, neither registers nor memory hold addresses!

Correctness

- Intuition: Each address ever created represented by register
- Invariant:
  1. If register holds address, it is in the same class as address' representative
  2. If memory holds address, it is in the same class as address of address dereferenced
- Formally: For all reachable states (ρ, μ), there exists a map $m : \mathrm{Addr} \to \mathrm{Reg}$, such that
  1. $\rho(x) \in \mathrm{Addr} \implies x \sim m(\rho(x))$
  2. $\mu(a, i) \in \mathrm{Addr} \implies m(a)[] \sim m(\mu(a, i))$
- Extracting alias information: x, y may alias, if $x \sim y$.
  - $\rho(x) = \rho(y) = a \in \mathrm{Addr} \implies x \sim m(a) \sim y$
- To show: Invariant holds initially, and preserved by steps
  - Initially: By assumption, neither registers nor memory hold addresses!
  - Preservation: On whiteboard

Implementation

- Need to implement `union`$*$ operation efficiently

Implementation

- Need to implement **`union`**$*$ operation efficiently
- Use Union-Find data structure

Implementation

- Need to implement **`union`**$\ast$ operation efficiently
- Use Union-Find data structure
- Equivalence classes identified by unique representative

Implementation

- Need to implement **`union*`** operation efficiently
- Use Union-Find data structure
- Equivalence classes identified by unique representative
- Operations:

Implementation

- Need to implement **`union*`** operation efficiently
- Use Union-Find data structure
- Equivalence classes identified by unique representative
- Operations:
  - `find(x)`: Return representative of $[x]$

Implementation

- Need to implement **`union*`** operation efficiently
- Use Union-Find data structure
- Equivalence classes identified by unique representative
- Operations:
  - `find(x)`: Return representative of $[x]$
  - **`union`**`(x,y)`: Join equivalence classes represented by x and y
    - Destructive update!

Union-Find: Idea

- ER represented as forest.

Union-Find: Idea

- ER represented as forest.
- Each node contains element and parent pointer.

Union-Find: Idea

- ER represented as forest.
- Each node contains element and parent pointer.
- Elements of trees are equivalence classes

Union-Find: Idea

- ER represented as forest.
- Each node contains element and parent pointer.
- Elements of trees are equivalence classes
- Representatives are roots of trees

Union-Find: Idea

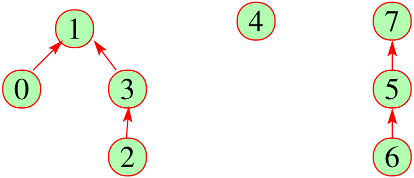
- ER represented as forest.
- Each node contains element and parent pointer.
- Elements of trees are equivalence classes
- Representatives are roots of trees
- Find: Follow tree upwards

Union-Find: Idea

- ER represented as forest.
- Each node contains element and parent pointer.
- Elements of trees are equivalence classes
- Representatives are roots of trees
- Find: Follow tree upwards
- Union: Link root node of one tree to other tree

Union-Find: Idea

- ER represented as forest.
- Each node contains element and parent pointer.
- Elements of trees are equivalence classes
- Representatives are roots of trees
- Find: Follow tree upwards
- Union: Link root node of one tree to other tree



0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

Union-Find: Optimizations

- Complexity: Union: $O(1)$, find: $O(n)$:(

Union-Find: Optimizations

- Complexity: Union: $O(1)$, find: $O(n)$:(
- Union by size: Connect root of smaller tree to root of bigger one

Union-Find: Optimizations

- Complexity: Union: $O(1)$, find: $O(n)$:(
- Union by size: Connect root of smaller tree to root of bigger one
  - Store size of tree in root node

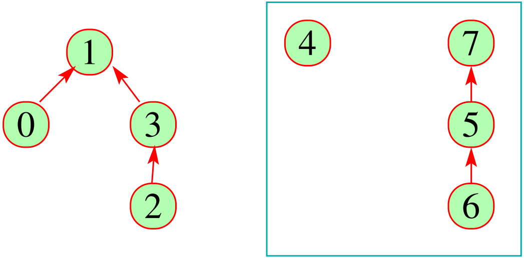
Union-Find: Optimizations

- Complexity: Union: $O(1)$, find: $O(n)$:(
- Union by size: Connect root of smaller tree to root of bigger one
  - Store size of tree in root node
  - C - implementation hack: Re/ab-use parent-pointer field for that

Union-Find: Optimizations

- Complexity: Union: $O(1)$, find: $O(n)$:(
- Union by size: Connect root of smaller tree to root of bigger one
  - Store size of tree in root node
  - C - implementation hack: Re/ab-use parent-pointer field for that
  - Complexity: Union: $O(1)$, find: $O(\log n)$:|

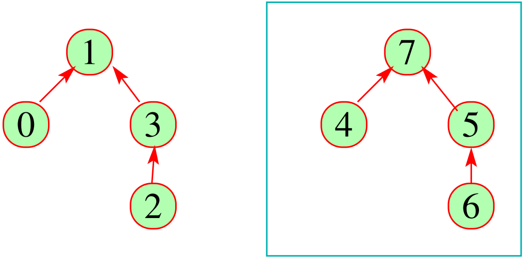
Union by size: Example



0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7

Union by size: Example



0	1	2	3	4	5	6	7

1	1	3	1	7	7	5	7

Path compression

- After find, redirect pointers on path to root node

Path compression

- After find, redirect pointers on path to root node
- Requires second pass for find

Path compression

- After find, redirect pointers on path to root node
- Requires second pass for find
  - Alternative: Connect each node on find-path to its grandfather

Path compression

- After find, redirect pointers on path to root node
- Requires second pass for find
  - Alternative: Connect each node on find-path to its grandfather
- Complexity, amortized for m find and $n-1$ union operations
  - $O(n + m\alpha(n))$
  - Where α is the inverse Ackerman-function

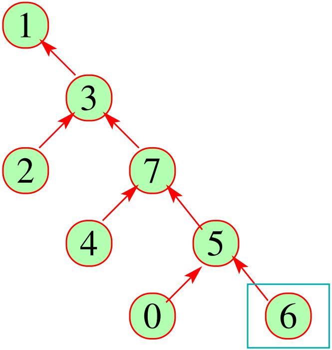
Path compression

- After find, redirect pointers on path to root node
- Requires second pass for find
  - Alternative: Connect each node on find-path to its grandfather
- Complexity, amortized for m find and $n-1$ union operations
  - $O(n + m\alpha(n))$
  - Where α is the inverse Ackerman-function
  - Note $n < 10^{80} \implies \alpha(n) < 5$

Path compression

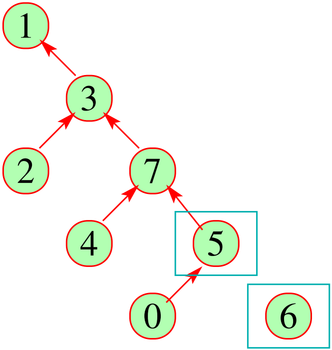
- After find, redirect pointers on path to root node
- Requires second pass for find
  - Alternative: Connect each node on find-path to its grandfather
- Complexity, amortized for m find and $n-1$ union operations
  - $O(n + m\alpha(n))$
  - Where α is the inverse Ackerman-function
  - Note $n < 10^{80} \implies \alpha(n) < 5$
  - Note: This complexity is optimal :)

Path compression: Example



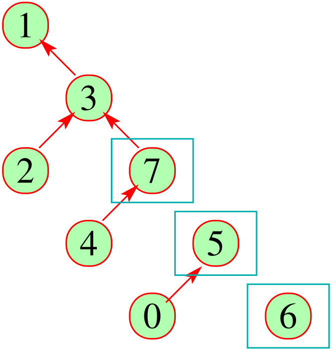
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3

Path compression: Example



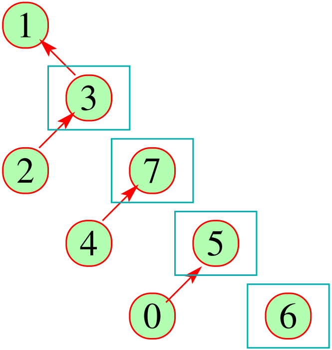
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3

Path compression: Example



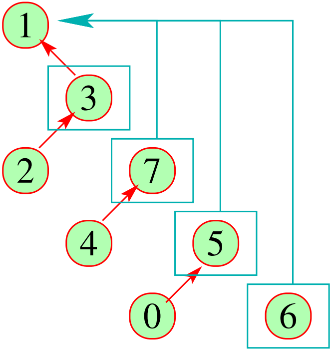
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3

Path compression: Example



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3

Path compression: Example



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1

Placing registers on top

- Try to preserve invariant:

Placing registers on top

- Try to preserve invariant:
  - If equivalence class contains register, its representative (root node) is register

Placing registers on top

- Try to preserve invariant:
  - If equivalence class contains register, its representative (root node) is register
  - On union, if linking register class to non-register class:

Placing registers on top

- Try to preserve invariant:
  - If equivalence class contains register, its representative (root node) is register
  - On union, if linking register class to non-register class:
    - Swap stored values in roots

Placing registers on top

- Try to preserve invariant:
  - If equivalence class contains register, its representative (root node) is register
  - On union, if linking register class to non-register class:
    - Swap stored values in roots
- Then, register equivalence class can be identified by its representative

Implementing union*

```
union⋆(x,y):
  x = find(x); y=find(y)
  if x != y then
    union(x,y)
    if x ∈ Regs & y ∈ Regs then
      union⋆(x[],y[])
```

Summary

- Complexity:
  - $O(|E| + |\mathrm{Reg}|)$ calls to union*, find. $O(|\mathrm{Reg}|)$ calls to union.

Summary

- Complexity:
  - $O(|E| + |\mathrm{Reg}|)$ calls to union*, find. $O(|\mathrm{Reg}|)$ calls to union.
- Analysis is fast. But may be imprecise.

Summary

- Complexity:
  - $O(|E| + |\mathrm{Reg}|)$ calls to union*, find. $O(|\mathrm{Reg}|)$ calls to union.
- Analysis is fast. But may be imprecise.
- More precise analysis too expensive for compilers.

Last Lecture

- Alias analysis by merging equivalence classes
- Implementation by union-find structure
  - Optimizations: Union-by-size, path-compression
  - Implementing union*

Evaluation

Please fill out evaluation forms online.

Table of Contents

Table of Contents

Idea

```
if * {
  x = M[5]
} else {
  y₁ = x + 1
}
y₂ = x + 1
M[1]=y₁ + y₂
```

Idea

```
if * {
  x = M[5]
} else {
  y₁ = x + 1
}
y₂ = x + 1
M[1]=y₁ + y₂
```

- `x+1` is evaluated on every path

Idea

```
if * {
  x = M[5]
} else {
  y₁ = x + 1
}
y₂ = x + 1
M[1]=y₁ + y₂
```

- `x+1` is evaluated on every path
- On else-path even two times

Goal

```
if * {
  x = M[5]
} else {
  y₁ = x + 1
}
y₂ = x + 1
M[1]=y₁ + y₂
```

Goal

```
if * {
  x = M[5]
} else {
  y₁ = x + 1
}
y₂ = x + 1
M[1]=y₁ + y₂
```

$\longmapsto$

```
if * {
  x = M[5]
  T=x + 1
} else {
  T = x + 1
  y₁ = T
}
y₂=T
M[1]=y₁ + T
```

Idea

- Insert assignments $T_e = e$, such that e is available at all program points where it is required.

Idea

- Insert assignments $T_e = e$, such that e is available at all program points where it is required.
- Insert assignments as early as possible.

Idea

- Insert assignments $T_e = e$, such that e is available at all program points where it is required.
- Insert assignments as early as possible.
- Do not add evaluations of e that would not have been executed at all.
  - `if x!=0 then y=6 div x` $\not\mapsto$ `T=6 div x; if x!=0 then y=T`

Very busy expressions

- An expression *e* is busy on path π, if it is evaluated on π before a variable of *e* is changed.

Very busy expressions

- An expression e is busy on path π, if it is evaluated on π before a variable of e is changed.
- e is very busy at u, if it is busy for all path from u to an end node.

Very busy expressions

- An expression *e* is busy on path π, if it is evaluated on π before a variable of *e* is changed.
- *e* is very busy at *u*, if it is busy for all path from *u* to an end node.
- Backwards must analysis, i.e., $\sqsubseteq = \supseteq$, $\sqcup = \cap$

Very busy expressions

- An expression e is busy on path π, if it is evaluated on π before a variable of e is changed.
- e is very busy at u, if it is busy for all path from u to an end node.
- Backwards must analysis, i.e., $\sqsubseteq = \supseteq$, $\sqcup = \cap$
- Semantic intuition:
  - e busy on π — evaluation of e can be placed at start of path
  - e very busy at u — evaluation can be placed at u
    - Without inserting unwanted additional evaluations

Abstract effects

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#} B &= B \\
[\![\mathrm{Pos}(e)]\!]^{\#} B &= B \cup \{e\} \\
[\![\mathrm{Neg}(e)]\!]^{\#} B &= B \cup \{e\} \\
[\![x := e]\!]^{\#} B &= (B \setminus \mathrm{Expr}_x) \cup \{e\} \\
[\![x := M[e]]\!]^{\#} B &= (B \setminus \mathrm{Expr}_x) \cup \{e\} \\
[\![M[e_1] = e_2]\!]^{\#} B &= B \cup \{e_1, e_2\}
\end{aligned}
$$

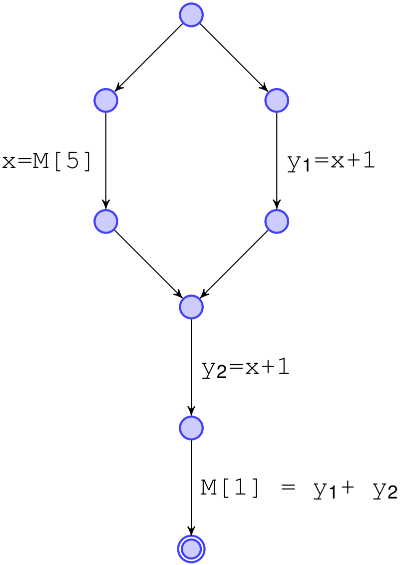
- Initial value: $\emptyset$
  - No very busy expressions at end nodes

Abstract effects

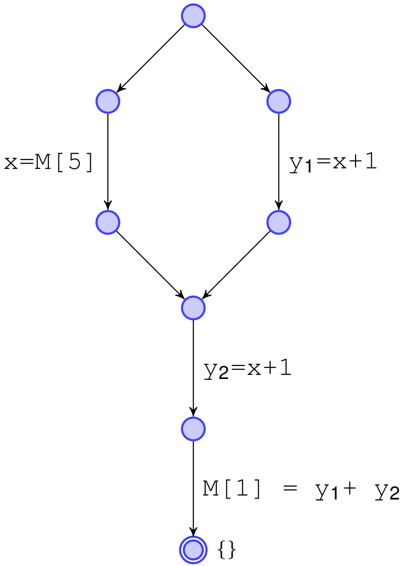
$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^\# B &= B \\
[\![\mathrm{Pos}(e)]\!]^\# B &= B \cup \{e\} \\
[\![\mathrm{Neg}(e)]\!]^\# B &= B \cup \{e\} \\
[\![x := e]\!]^\# B &= (B \setminus \mathrm{Expr}_x) \cup \{e\} \\
[\![x := M[e]]\!]^\# B &= (B \setminus \mathrm{Expr}_x) \cup \{e\} \\
[\![M[e_1] = e_2]\!]^\# B &= B \cup \{e_1, e_2\}
\end{aligned}
$$

- Initial value: $\emptyset$
  - No very busy expressions at end nodes
- Kill/Gen analysis, i.e., distributive
  - $\mathrm{MOP} = \mathrm{MFP}$, if end node reachable from every node

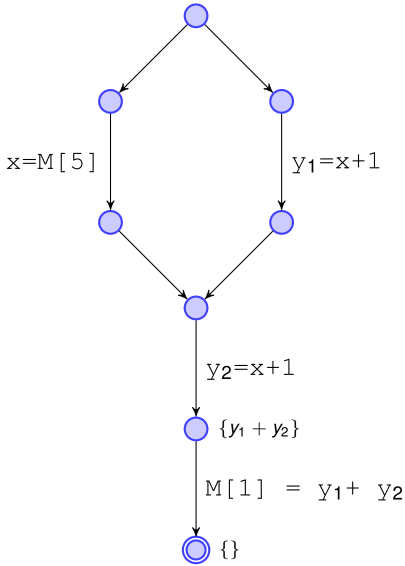
Example (Very Busy Expressions)



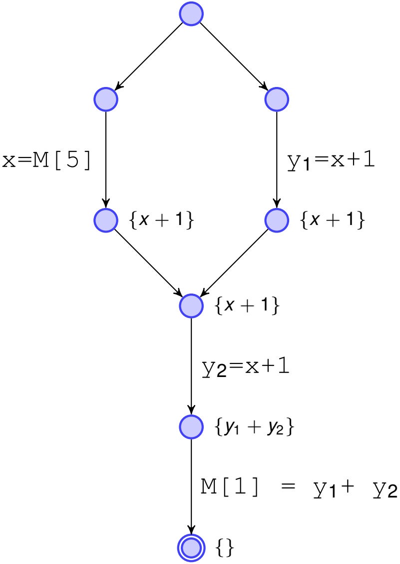
Example (Very Busy Expressions)



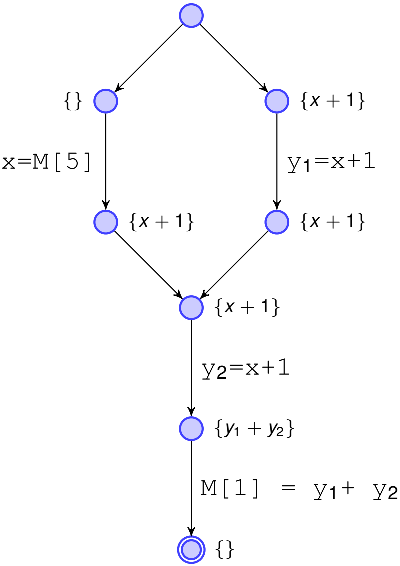
Example (Very Busy Expressions)



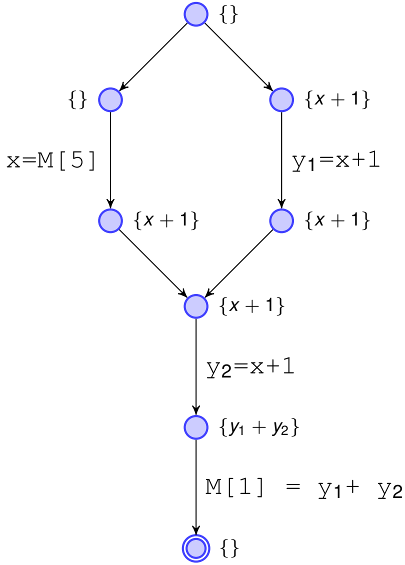
Example (Very Busy Expressions)



Example (Very Busy Expressions)



Example (Very Busy Expressions)



Available expressions

- Recall: Available expressions before memo-transformation

$$\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#}_{\mathcal{A}} A &:= A \\
[\![\mathrm{Pos}(e)]\!]^{\#}_{\mathcal{A}} A &:= A \cup \{e\} \\
[\![\mathrm{Neg}(e)]\!]^{\#}_{\mathcal{A}} A &:= A \cup \{e\} \\
[\![R = e]\!]^{\#}_{\mathcal{A}} A &:= (A \cup \{e\}) \setminus \mathrm{Expr}_R \\
[\![R = M[e]]\!]^{\#}_{\mathcal{A}} A &:= (A \cup \{e\}) \setminus \mathrm{Expr}_R \\
[\![M[e_1] = e_2]\!]^{\#}_{\mathcal{A}} A &:= A \cup \{e_1, e_2\}
\end{aligned}$$

Transformation

- Insert $T_e = e$ after edge (u, a, v), if

Transformation

- Insert $T_e = e$ after edge (u, a, v), if
  - e is very busy at v

Transformation

- Insert $T_e = e$ after edge (u, a, v), if
  - e is very busy at v
  - Evaluation could not have been inserted before, b/c
    - e destroyed by a, or
    - e neither available, nor very busy at u

Transformation

- Insert $T_e = e$ after edge (u, a, v), if
  - e is very busy at v
  - Evaluation could not have been inserted before, b/c
    - e destroyed by a, or
    - e neither available, nor very busy at u
  - Formally: $e \in B[v] \setminus [\![a]\!]^{\#}_{\mathcal{A}}(A[u] \cup B[u])$

Transformation

- Insert $T_e = e$ after edge (u, a, v), if
  - e is very busy at v
  - Evaluation could not have been inserted before, b/c
    - e destroyed by a, or
    - e neither available, nor very busy at u
  - Formally: $e \in B[v] \setminus [\![a]\!]^{\#}_{\mathcal{A}}(A[u] \cup B[u])$
- At program start, insert evaluations of $B[v_0]$

Transformation

- Insert $T_e = e$ after edge (u, a, v), if
  - e is very busy at v
  - Evaluation could not have been inserted before, b/c
    - e destroyed by a, or
    - e neither available, nor very busy at u
  - Formally: $e \in B[v] \setminus [\![a]\!]^{\#}_{\mathcal{A}}(A[u] \cup B[u])$
- At program start, insert evaluations of $B[v_0]$
  - Note: Order does not matter

Transformation

- Place evaluations of expressions
  - $(u, a, v) \mapsto \{(u, a, w), (w, T_e = e, v)\}$ for $e \in B[v] \setminus [\![a]\!]^{\#}_{\mathcal{A}}(A[u] \cup B[u])$
  - For fresh node w

Transformation

- Place evaluations of expressions
  - $(u, a, v) \mapsto \{(u, a, w), (w, T_e = e, v)\}$ for $e \in B[v] \setminus [\![a]\!]^\#_\mathcal{A}(A[u] \cup B[u])$
  - For fresh node w
- $v_0 \mapsto {v_0}'$ with $({v_0}', T_e = e, v_0)$ for $e \in B[v_0]$

Transformation

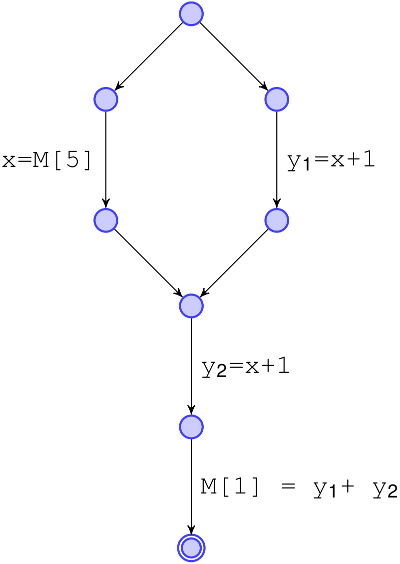
- Place evaluations of expressions
  - $(u, a, v) \mapsto \{(u, a, w), (w, T_e = e, v)\}$ for $e \in B[v] \setminus [\![a]\!]^{\#}_{\mathcal{A}}(A[u] \cup B[u])$
  - For fresh node w
- $v_0 \mapsto v_0{}'$ with $(v_0{}', T_e = e, v_0)$ for $e \in B[v_0]$
- Note: Multiple memo-assignments on one edge
  - Can just be expanded in any order

Transformation

- Place evaluations of expressions
  - $(u, a, v) \mapsto \{(u, a, w), (w, T_e = e, v)\}$ for $e \in B[v] \setminus [\![a]\!]^\#_{\mathcal{A}}(A[u] \cup B[u])$
  - For fresh node w
- $v_0 \mapsto {v_0}'$ with $({v_0}', T_e = e, v_0)$ for $e \in B[v_0]$
- Note: Multiple memo-assignments on one edge
  - Can just be expanded in any order
- Replace usages of expressions
  - $(u, x = e, v) \mapsto (u, x = T_e, v)$
  - analogously for other uses of e

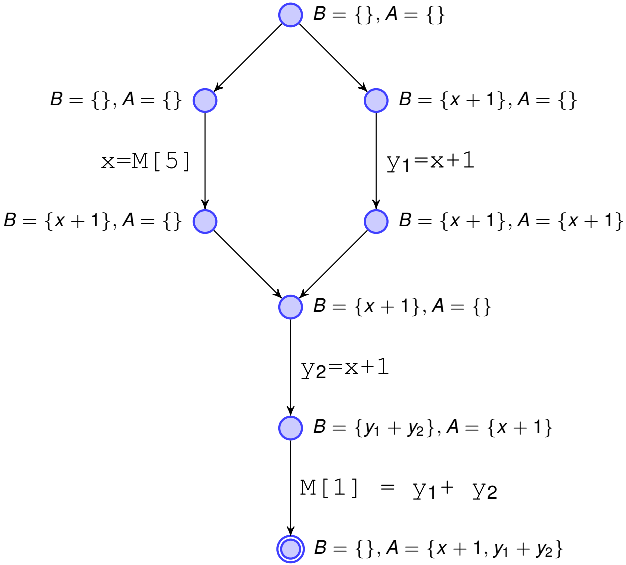
Example

- For expression $x + 1$ only



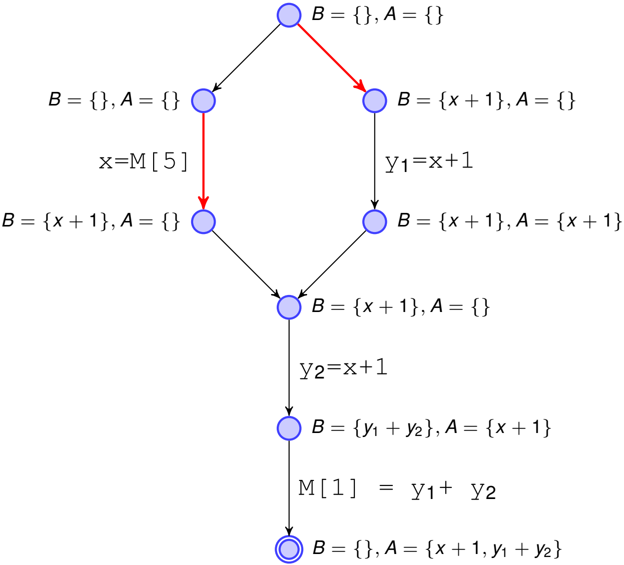
Example

- For expression $x+1$ only



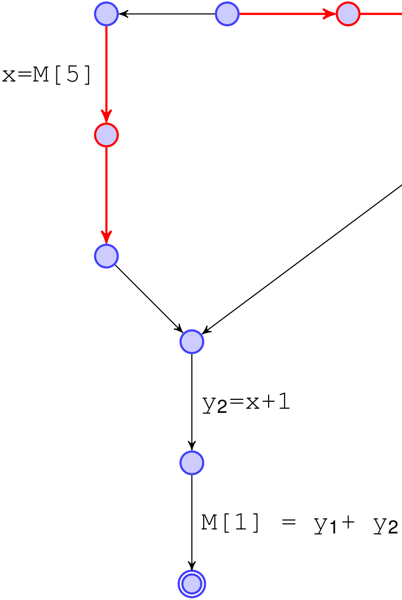
Example

- For expression $x + 1$ only



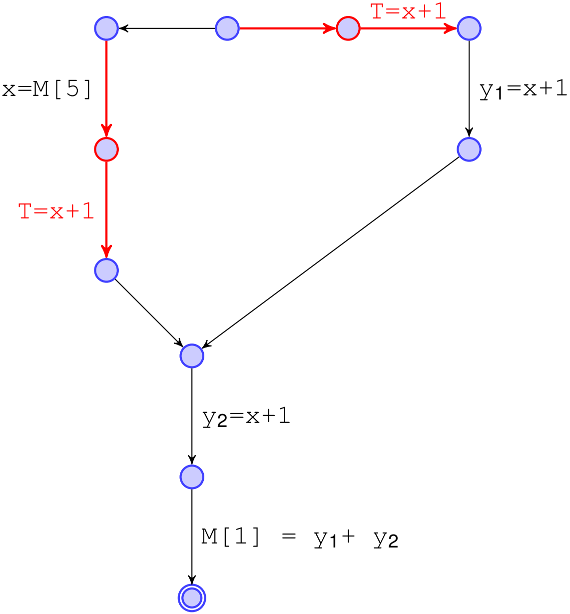
Example

- For expression $x + 1$ only



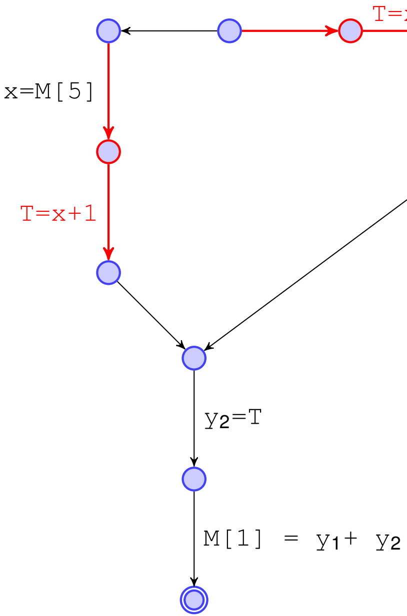
Example

- For expression $x + 1$ only



Example

- For expression $x + 1$ only



Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.
  - Empty path: Evaluation placed before start node

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.
  - Empty path: Evaluation placed before start node
  - $\pi = \pi'(u, a, v)$:

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.
  - Empty path: Evaluation placed before start node
  - $\pi = \pi'(u, a, v)$:
    - Case a modifies $e \implies e \notin [\![a]\!]^{\#}_{\mathcal{A}}(\ldots) \implies$ Evaluation placed here.

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.
  - Empty path: Evaluation placed before start node
  - $\pi = \pi'(u, a, v)$:
    - Case a modifies $e \implies e \notin [\![a]\!]^{\#}_{\mathcal{A}}(\ldots) \implies$ Evaluation placed here.
    - Case $e \notin A[u] \cup B[u] \implies$ Evaluation placed here.

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.
  - Empty path: Evaluation placed before start node
  - $\pi = \pi'(u, a, v)$:
    - Case a modifies $e \implies e \notin [\![a]\!]^{\#}_{\mathcal{A}}(\dots) \implies$ Evaluation placed here.
    - Case $e \notin A[u] \cup B[u] \implies$ Evaluation placed here.
    - Assume: a does not modify e

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.
  - Empty path: Evaluation placed before start node
  - $\pi = \pi'(u, a, v)$:
    - Case a modifies $e \implies e \notin [\![a]\!]^\#_{\mathcal{A}}(\ldots) \implies$ Evaluation placed here.
    - Case $e \notin A[u] \cup B[u] \implies$ Evaluation placed here.
    - Assume: a does not modify e
    - Case $e \in B[u]$. Induction hypothesis.

Correctness (Sketch)

- Assumption: Same set of expressions occur at all outgoing edges of a node
  - True for our translation scheme
  - Be careful in general!

$\Longrightarrow$ Required expressions are very busy at start node of edge

- Regard path from start node over edge to end node: $\pi_1(u, a, v)\pi_2$
  - Assume expression e required by a
  - $e \in B[u]$
- Show: On any path π from v_0 to v with $e \in B[v]$, evaluation of e is placed such that it is available at v
- Induction on π.
  - Empty path: Evaluation placed before start node
  - $\pi = \pi'(u, a, v)$:
    - Case a modifies $e \implies e \notin [\![a]\!]^{\#}_{\mathcal{A}}(\dots) \implies$ Evaluation placed here.
    - Case $e \notin A[u] \cup B[u] \implies$ Evaluation placed here.
    - Assume: a does not modify e
    - Case $e \in B[u]$. Induction hypothesis.
    - Case $e \in A[u] \implies \pi' = \pi_1'(u', a', v')\pi_2'$, such that π_2' does not modify e, and e required by $a' \implies e \in B[u']$. Induction hypothesis.

Non-degradation of performance

- On any path: Placement of $T_e = e$ corresponds to replacing an e by T_e
    - e not evaluated more often than in original program

Non-degradation of performance

- On any path: Placement of $T_e = e$ corresponds to replacing an e by T_e
  - e not evaluated more often than in original program
- Proof sketch: Placement only done where e is very busy

Non-degradation of performance

- On any path: Placement of $T_e = e$ corresponds to replacing an e by T_e
  - e not evaluated more often than in original program
- Proof sketch: Placement only done where e is very busy
  - I.e., every path from placement contains evaluation of e, which will be replaced

Non-degradation of performance

- On any path: Placement of $T_e = e$ corresponds to replacing an e by T_e
  - e not evaluated more often than in original program
- Proof sketch: Placement only done where e is very busy
  - I.e., every path from placement contains evaluation of e, which will be replaced
  - Moreover, no path contains two evaluations of e, without usage of e in between

Non-degradation of performance

- On any path: Placement of $T_e = e$ corresponds to replacing an e by T_e
  - e not evaluated more often than in original program
- Proof sketch: Placement only done where e is very busy
  - I.e., every path from placement contains evaluation of e, which will be replaced
  - Moreover, no path contains two evaluations of e, without usage of e in between
    - By contradiction. Sketch on board.

Last Lecture

- Partial Redundancy Elimination
  - Place evaluations such that
    - They are evaluated as early as possible, such that:
    - Expressions are only evaluated if also evaluated in original program
- Analysis: Very Busy Expressions
- Transformation: Placement on edges
  - where expression stops to be very busy
  - or is destroyed (and very busy at target)
- Placement only if expression is not avalable

Application: Moving loop-invariant code

```
for (i=0;i<N; ++i)
  a[i] = b + 3
```

- `b+3` evaluated in every iteration.

Application: Moving loop-invariant code

```
for (i=0;i<N; ++i)
  a[i] = b + 3
```

- `b+3` evaluated in every iteration.
- To the same value

Application: Moving loop-invariant code

```
for (i=0;i<N; ++i)
  a[i] = b + 3
```

- `b+3` evaluated in every iteration.
- To the same value
- Should be avoided!

Example (CFG)

CFG of previous example

```
1: i=0;
2: if (i<N) {
3:    a[i] = b + 3
4:    i=i+1
5:    goto 2
6: }
```

Example (CFG)

Analysis results for expression $b + 3$

```
1: i=0;
2: if (i<N) {
3:   a[i] = b + 3 // B
4:   i=i+1        // A
5:   goto 2       // A
6: }
```

Example (CFG)

Placement happens inside loop, on edge $(2, \mathrm{Pos}(i < N), 3)$:(

```
1: i=0;
2: if (i<N) {
x:   T=b+3
3:   a[i] = T
4:   i=i+1
5:   goto 2
6: }
```

Example (CFG)

There is no node outside loop for placing *e*!

```
1: i=0;
2: if (i<N) {
x:   T=b+3
3:   a[i] = T
4:   i=i+1
5:   goto 2
6: }
```

Solution: Loop inversion

- Idea: Convert while-loop to do-while loop

`while (b) do c` $\longmapsto$ `if (b) {do c while (b)}`

Solution: Loop inversion

- Idea: Convert while-loop to do-while loop

`while (b) do c` $\longmapsto$ `if (b) {do c while (b)}`

- Does not change semantics

Solution: Loop inversion

- Idea: Convert while-loop to do-while loop

`while (b) do c` $\longmapsto$ `if (b) {do c while (b)}`

- Does not change semantics
- But creates node for placing loop invariant code

Example

CFG after loop inversion

```
1: i=0;
2: if (i<N) {
3:   a[i] = b + 3
4:   i=i+1
5:   if (i<N) goto 3
6: }
```

Example

Analysis results for expression $b + 3$

```
1: i=0;
2: if (i<N) {
3:    a[i] = b + 3      // B
4:    i=i+1             // A
5:    if (i<N) goto 3   // A
6: }
```

Example

Placement happens outside loop, on edge $(2, \mathrm{Pos}(i < N), 3)$:)

```
1: i=0;
2: if (i<N) {
x:   T=b+3;
3:   a[i] = T
4:   i=i+1
5:   if (i<N) goto 3
6: }
```

Conclusion

- PRE may move loop-invariant code out of the loop

Conclusion

- PRE may move loop-invariant code out of the loop
- Only for do-while loops

Conclusion

- PRE may move loop-invariant code out of the loop
- Only for do-while loops
- To also cover while-loops: Apply loop-inversion first

Conclusion

- PRE may move loop-invariant code out of the loop
- Only for do-while loops
- To also cover while-loops: Apply loop-inversion first
- Loop inversion: No additional statements executed.
  - But slight increase in code size.
  - Side note: Better pipelining behavior (Less jumps executed)

Detecting loops in CFG

- Loop inversion can be done in AST

Detecting loops in CFG

- Loop inversion can be done in AST
  - But only if AST is available

Detecting loops in CFG

- Loop inversion can be done in AST
  - But only if AST is available
  - What if some other CFG-based transformations have already been run?

Detecting loops in CFG

- Loop inversion can be done in AST
  - But only if AST is available
  - What if some other CFG-based transformations have already been run?
- Need CFG-based detection of loop headers

Detecting loops in CFG

- Loop inversion can be done in AST
  - But only if AST is available
  - What if some other CFG-based transformations have already been run?
- Need CFG-based detection of loop headers
- Idea: Predominators

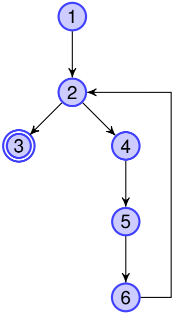
Predominators

- A node u pre-dominates v ($u \Rightarrow v$), iff every path $v_0 \to^* v$ contains u.

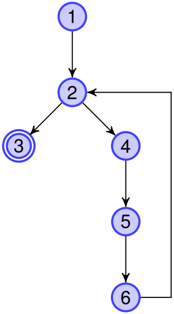
Predominators

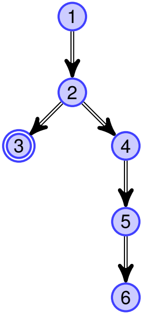
- A node u pre-dominates v ($u \Rightarrow v$), iff every path $v_0 \rightarrow^* v$ contains u.
- $\Rightarrow$ is a partial order.
  - reflexive, transitive, anti-symmetric

Predominator example



Predominator example





Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree

Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree
  - Clearly, v_0 dominates every node (root of tree)

Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree
  - Clearly, v_0 dominates every node (root of tree)
  - Every node has at most one immediate predecessor:

Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree
  - Clearly, v_0 dominates every node (root of tree)
  - Every node has at most one immediate predecessor:
    - Assume $u_1 \Rightarrow v$, $u_2 \Rightarrow v$, and neither $u_1 \Rightarrow u_2$ nor $u_2 \Rightarrow u_1$

Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree
  - Clearly, v_0 dominates every node (root of tree)
  - Every node has at most one immediate predecessor:
    - Assume $u_1 \Rightarrow v$, $u_2 \Rightarrow v$, and neither $u_1 \Rightarrow u_2$ nor $u_2 \Rightarrow u_1$
    - Regard path π to v. Assume, wlog, $\pi = \pi_1 u_1 \pi_2 v$, such that $u_1, u_2 \notin \pi_2$

Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree
  - Clearly, v_0 dominates every node (root of tree)
  - Every node has at most one immediate predecessor:
    - Assume $u_1 \Rightarrow v$, $u_2 \Rightarrow v$, and neither $u_1 \Rightarrow u_2$ nor $u_2 \Rightarrow u_1$
    - Regard path π to v. Assume, wlog, $\pi = \pi_1 u_1 \pi_2 v$, such that $u_1, u_2 \notin \pi_2$
    - Then, every path π' to u_1 gives rise to path $\pi' \pi_2$ to v.

Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree
    - Clearly, v_0 dominates every node (root of tree)
    - Every node has at most one immediate predecessor:
        - Assume $u_1 \Rightarrow v$, $u_2 \Rightarrow v$, and neither $u_1 \Rightarrow u_2$ nor $u_2 \Rightarrow u_1$
        - Regard path π to v. Assume, wlog, $\pi = \pi_1 u_1 \pi_2 v$, such that $u_1, u_2 \notin \pi_2$
        - Then, every path π' to u_1 gives rise to path $\pi' \pi_2$ to v.
        - Thus, $u_2 \in \pi' \pi_2$. By asm, not in π_2. I.e. $u_2 \in \pi'$.

Remark: Immediate Predominator

- The $\Rightarrow$-relation, with reflexivity and transitivity removed, is a tree
  - Clearly, v_0 dominates every node (root of tree)
  - Every node has at most one immediate predecessor:
    - Assume $u_1 \Rightarrow v$, $u_2 \Rightarrow v$, and neither $u_1 \Rightarrow u_2$ nor $u_2 \Rightarrow u_1$
    - Regard path π to v. Assume, wlog, $\pi = \pi_1 u_1 \pi_2 v$, such that $u_1, u_2 \notin \pi_2$
    - Then, every path π' to u_1 gives rise to path $\pi' \pi_2$ to v.
    - Thus, $u_2 \in \pi' \pi_2$. By asm, not in π_2. I.e. $u_2 \in \pi'$.
    - Thus, $u_2 \Rightarrow u_1$, contradiction.

Computing predominators

- Use a (degenerate) dataflow analysis. Forward, Must. Domain 2^V.

Computing predominators

- Use a (degenerate) dataflow analysis. Forward, Must. Domain 2^V.
- $[\![(\_,\_,v)]\!]^\# P = P \cup \{v\}$, $d_0 = \{v_0\}$

Computing predominators

- Use a (degenerate) dataflow analysis. Forward, Must. Domain 2^V.
- $[\![(\_,\_,v)]\!]^\# P = P \cup \{v\}$, $d_0 = \{v_0\}$
    - Collects nodes on paths

Computing predominators

- Use a (degenerate) dataflow analysis. Forward, Must. Domain 2^V.
- $[\![(\_, \_, v)]\!]^\# P = P \cup \{v\}$, $d_0 = \{v_0\}$
  - Collects nodes on paths
  - Distributive, i.e. MOP can be precisely computed

Computing predominators

- Use a (degenerate) dataflow analysis. Forward, Must. Domain 2^V.
- $[\![(\_,\_,v)]\!]^{\#} P = P \cup \{v\}$, $d_0 = \{v_0\}$
    - Collects nodes on paths
    - Distributive, i.e. MOP can be precisely computed
- $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^{\#}\{v_0\} \mid v_0 \rightarrow^* u\}$

Computing predominators

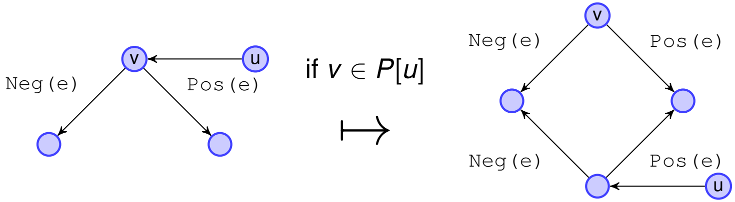
- Use a (degenerate) dataflow analysis. Forward, Must. Domain 2^V.
- $[\![(\_,\_,v)]\!]^\# P = P \cup \{v\}$, $d_0 = \{v_0\}$
    - Collects nodes on paths
    - Distributive, i.e. MOP can be precisely computed
- $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^\#\{v_0\} \mid v_0 \to^* u\}$
    - Which is precisely the set of nodes occurring on all paths to u

Computing predominators

- Use a (degenerate) dataflow analysis. Forward, Must. Domain 2^V.
- $[\![(\_,\_,v)]\!]^\# P = P \cup \{v\}$, $d_0 = \{v_0\}$
  - Collects nodes on paths
  - Distributive, i.e. MOP can be precisely computed
- $\mathrm{MOP}[u] = \bigcap\{[\![\pi]\!]^\#\{v_0\} \mid v_0 \rightarrow^* u\}$
  - Which is precisely the set of nodes occurring on all paths to u
  - I.e. the predominators of u

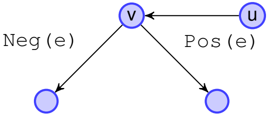
Detecting loops using predominators

- Observation: Entry node of loop predominates all nodes in loop body.



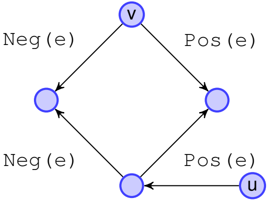
Detecting loops using predominators

- Observation: Entry node of loop predominates all nodes in loop body.
  - In particular the start node of the back edge



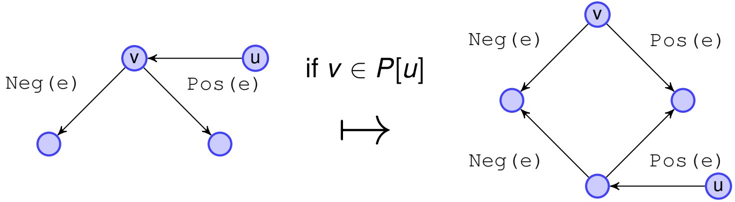
if $v \in P[u]$

$\mapsto$



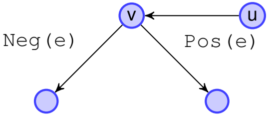
Detecting loops using predominators

- Observation: Entry node of loop predominates all nodes in loop body.
  - In particular the start node of the back edge
- Loop inversion transformation



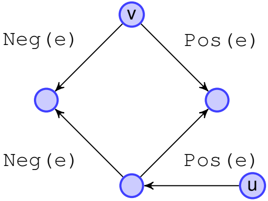
Detecting loops using predominators

- Observation: Entry node of loop predominates all nodes in loop body.
  - In particular the start node of the back edge
- Loop inversion transformation



if $v \in P[u]$

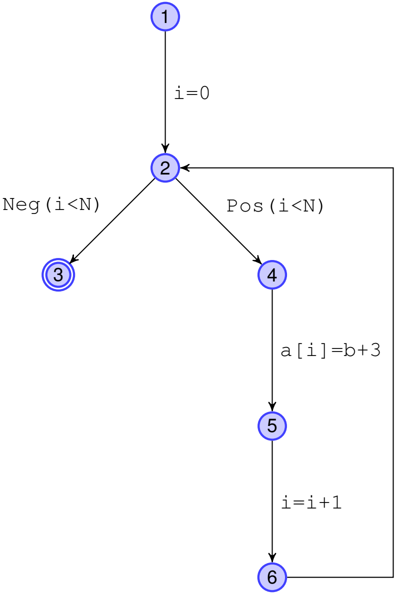
$\mapsto$



- Obviously correct

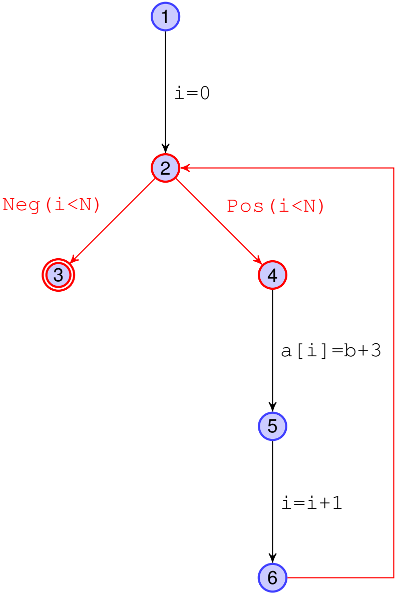
Example

CFG of running example



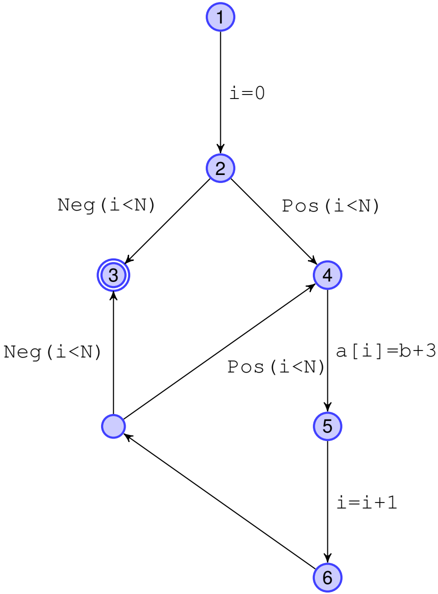
Example

$2 \in P[6]$, identified pattern for transformation



Example

Inverted loop



Warning

- Transformation fails to invert all loops

Warning

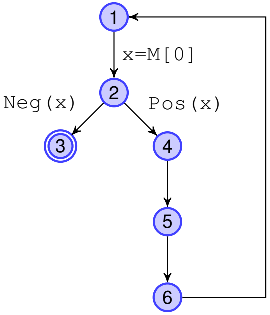
- Transformation fails to invert all loops
- E.g., if evaluation of condition is more complex

Warning

- Transformation fails to invert all loops
- E.g., if evaluation of condition is more complex
    - E.g., condition contains loads
    - `while (M[0]) ...`

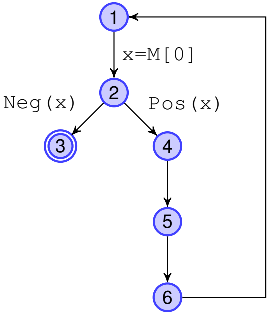
Warning

- Transformation fails to invert all loops
- E.g., if evaluation of condition is more complex
    - E.g., condition contains loads
    - `while (M[0]) ...`



Warning

- Transformation fails to invert all loops
- E.g., if evaluation of condition is more complex
  - E.g., condition contains loads
  - `while (M[0]) ...`



- We would have to duplicate the load-edge, too

Last Lecture

- Partial redundancy elimination
  - Very busy expressions
  - Place evaluations as early as possible
- Loop inversion
  - **`while`** $\rightarrow$ **`do-while`**
  - Enables moving loop-invariant code out of loops
  - Computation on CFG: Use pre-dominators

Table of Contents

Motivation

- Consider program

```
T = x+1
if (*) then M[0]=T
```

  - Assume (*) does not use T, and T dead at end

Motivation

- Consider program

```
T = x+1
if (*) then M[0]=T
```

  - Assume (*) does not use T, and T dead at end
- Assignment $T = x + 1$ only required on one path

Motivation

- Consider program

```
T = x+1
if (*) then M[0]=T
```

  - Assume (*) does not use T, and T dead at end
- Assignment $T = x + 1$ only required on one path
- Would like to move assignment into this path

```
if (*) then {T = x+1; M[0]=T}
```

Idea

- Delay assignments as long as possible

Idea

- Delay assignments as long as possible
- Can delay assignment `x:=e` over edge k, if
  - x is not used, nor defined by k
  - No variable of e is defined by k

Delayable Assignments Analysis

- Domain: $\{x = e \mid x \in \mathrm{Reg} \wedge e \in \mathrm{Expr}\}$, Ordering: $\sqsubseteq = \supseteq$, forward
  - I.e. forward must analysis

Delayable Assignments Analysis

- Domain: $\{x = e \mid x \in \mathrm{Reg} \wedge e \in \mathrm{Expr}\}$, Ordering: $\sqsubseteq = \supseteq$, forward
  - I.e. forward must analysis
- $d_0 = \emptyset$, no delayable assignments at program start

$$
\begin{aligned}
[\![\mathrm{Nop}]\!]^{\#} D &= D \\
[\![x = e]\!]^{\#} D &= D \setminus (\mathrm{Ass}(e) \cup \mathrm{Occ}(x)) \cup \{x = e\} \\
[\![\mathrm{Pos}(e)]\!]^{\#} D &= D \setminus \mathrm{Ass}(e) \\
[\![\mathrm{Neg}(e)]\!]^{\#} D &= D \setminus \mathrm{Ass}(e) \\
[\![x = M[e]]\!]^{\#} D &= D \setminus (\mathrm{Ass}(e) \cup \mathrm{Occ}(x)) \\
[\![M[e_1] = e_2]\!]^{\#} D &= D \setminus (\mathrm{Ass}(e_1) \cup \mathrm{Ass}(e_2))
\end{aligned}
$$

where

$$\mathrm{Ass}(e) := \{x = e' \mid x \in \mathrm{Reg}(e)\} \qquad \text{Assignments to variable in } e$$

$$\mathrm{Occ}(x) := \{x' = e \mid x = x' \vee x \in \mathrm{Reg}(e)\} \qquad \text{Assignments in which } x \text{ occurs}$$

Intuition

- $x = e \in D[u]$: On every path reaching u, the assignment $x = e$ is executed, and no edge afterwards:
  - Depends on x
  - Changes x or a variable of e

Intuition

- $x = e \in D[u]$: On every path reaching u, the assignment $x = e$ is executed, and no edge afterwards:
  - Depends on x
  - Changes x or a variable of e
- Thus, this assignment can be safely moved to u

Transformation

- Delay assignments as far as possible

Transformation

- Delay assignments as far as possible
- Do not place assignments to dead variables

Transformation

- Delay assignments as far as possible
- Do not place assignments to dead variables
- $(u, x = e, v) \mapsto (u, ss_1, w), (w, ss_2, v)$ where
  - ss_1 Assignments to live variables that cannot be delayed over action $x = e$
  - ss_2 Assignments to live variables delayable due to edge, but not at v (Other paths over v)
  - w is fresh node

Transformation

- Delay assignments as far as possible
- Do not place assignments to dead variables
- $(u, x = e, v) \mapsto (u, ss_1, w), (w, ss_2, v)$ where
  - ss_1 Assignments to live variables that cannot be delayed over action $x = e$
  - ss_2 Assignments to live variables delayable due to edge, but not at v (Other paths over v)
  - w is fresh node
  - Formally

$$ss_1 := \{x' = e' \in D[u] \setminus [\![x = e]\!]^{\#} D[u] \mid x' \in L[u]\}$$

$$ss_2 = \{x' = e' \in [\![x = e]\!]^{\#} D[u] \setminus D[v] \mid x' \in L[v]\}$$

Transformation

- Delay assignments as far as possible
- Do not place assignments to dead variables
- $(u, x = e, v) \mapsto (u, ss_1, w), (w, ss_2, v)$ where
  - ss_1 Assignments to live variables that cannot be delayed over action $x = e$
  - ss_2 Assignments to live variables delayable due to edge, but not at v (Other paths over v)
  - w is fresh node
  - Formally

$$ss_1 := \{x' = e' \in D[u] \setminus [\![x = e]\!]^\# D[u] \mid x' \in L[u]\}$$
$$ss_2 = \{x' = e' \in [\![x = e]\!]^\# D[u] \setminus D[v] \mid x' \in L[v]\}$$

- $(u, a, v) \mapsto (u, ss_1, w_1), (w_1, a, w_2), (w_2, ss_2, v)$ for a not assignment

$$ss_1 := \{x' = e' \in D[u] \setminus [\![a]\!]^\# D[u] \mid x' \in L[u]\}$$
$$ss_2 = \{x' = e' \in [\![a]\!]^\# D[u] \setminus D[v] \mid x' \in L[v]\}$$

Transformation

- Delay assignments as far as possible
- Do not place assignments to dead variables
- $(u, x = e, v) \mapsto (u, ss_1, w), (w, ss_2, v)$ where
  - ss_1 Assignments to live variables that cannot be delayed over action $x = e$
  - ss_2 Assignments to live variables delayable due to edge, but not at v (Other paths over v)
  - w is fresh node
  - Formally

$$ss_1 := \{x' = e' \in D[u] \setminus [\![x = e]\!]^{\#} D[u] \mid x' \in L[u]\}$$
$$ss_2 = \{x' = e' \in [\![x = e]\!]^{\#} D[u] \setminus D[v] \mid x' \in L[v]\}$$

- $(u, a, v) \mapsto (u, ss_1, w_1), (w_1, a, w_2), (w_2, ss_2, v)$ for a not assignment

$$ss_1 := \{x' = e' \in D[u] \setminus [\![a]\!]^{\#} D[u] \mid x' \in L[u]\}$$
$$ss_2 = \{x' = e' \in [\![a]\!]^{\#} D[u] \setminus D[v] \mid x' \in L[v]\}$$

- $v_e \in V_{\mathrm{end}} \mapsto (v_e, D[v_e], v_e')$
  - where v_e' is fresh end node, and v_e no end node any more.

Dependent actions

- Two actions a_1, a_2 are independent, iff $[\![a_1 a_2]\!] = [\![a_2 a_1]\!]$
  - Actions may be swapped

Dependent actions

- Two actions a_1, a_2 are independent, iff $[\![a_1 a_2]\!] = [\![a_2 a_1]\!]$
  - Actions may be swapped
- Assignments only delayed over independent actions

Correctness (Rough Sketch)

- First: $D[u]$ does never contain dependent assignments
  - Placement order is irrelevant

Correctness (Rough Sketch)

- First: $D[u]$ does never contain dependent assignments
  - Placement order is irrelevant
  - Proof sketch: $x = e$ only inserted by $[\![\cdot]\!]^\#$, after all dependent assignments removed

Correctness (Rough Sketch)

- First: $D[u]$ does never contain dependent assignments
  - Placement order is irrelevant
  - Proof sketch: $x = e$ only inserted by $[\![\cdot]\!]^\#$, after all dependent assignments removed
- Regard path with assignment $(u, x = e, v)$.

Correctness (Rough Sketch)

- First: $D[u]$ does never contain dependent assignments
  - Placement order is irrelevant
  - Proof sketch: $x = e$ only inserted by $[\![\cdot]\!]^\#$, after all dependent assignments removed
- Regard path with assignment $(u, x = e, v)$.
- We have $x = e \in [\![x = e]\!]^\# D[u]$. (1) Either placed here, (2) x dead, (3) or delayable at v

Correctness (Rough Sketch)

- First: $D[u]$ does never contain dependent assignments
  - Placement order is irrelevant
  - Proof sketch: $x = e$ only inserted by $[\![\cdot]\!]^\#$, after all dependent assignments removed
- Regard path with assignment $(u, x = e, v)$.
- We have $x = e \in [\![x = e]\!]^\# D[u]$. (1) Either placed here, (2) x dead, (3) or delayable at v
  - (1) No change of path

Correctness (Rough Sketch)

- First: $D[u]$ does never contain dependent assignments
  - Placement order is irrelevant
  - Proof sketch: $x = e$ only inserted by $[\![\cdot]\!]^\#$, after all dependent assignments removed
- Regard path with assignment $(u, x = e, v)$.
- We have $x = e \in [\![x = e]\!]^\# D[u]$. (1) Either placed here, (2) x dead, (3) or delayable at v
  - (1) No change of path
  - (2), not (3): Assignment dropped, but was dead anyway

Correctness (Rough Sketch)

- First: $D[u]$ does never contain dependent assignments
  - Placement order is irrelevant
  - Proof sketch: $x = e$ only inserted by $[\![\cdot]\!]^{\#}$, after all dependent assignments removed
- Regard path with assignment $(u, x = e, v)$.
- We have $x = e \in [\![x = e]\!]^{\#} D[u]$. (1) Either placed here, (2) x dead, (3) or delayable at v
  - (1) No change of path
  - (2), not (3): Assignment dropped, but was dead anyway
  - (3). Three subcases: Sketch on whiteboard!
    - (3.1) $x = e$ stops being delayable due to dependent action

    $\Longrightarrow$ Assignment placed before this action, if live
    - (3.2) $x = e$ stops being delayable at node

    $\Longrightarrow$ Assignment placed after edge to this node, if live
    - (3.3) $x = e$ delayable until end

    $\Longrightarrow$ Assignment placed at end node, if live

Example

```
1: T = x+1             D: {}        L: {x}
2: if (*) then {       D: {T=x+1}   L: {T}
3:   M[0]=T            D: {T=x+1}   L: {T}
4:   Nop               D: {}        L: {}
5: }                   D: {}        L: {}
```

Example

```
1: T = x+1              D: {}        L: {x}
2: if (*) then {        D: {T=x+1}   L: {T}
3:   M[0]=T             D: {T=x+1}   L: {T}
4:   Nop                D: {}        L: {}
5: }                    D: {}        L: {}
```

- Placement of $T = x + 1$ before edge $(3, 4)$
  - We have $T = x + 1 \in D[3] \setminus [\![M[0] = T]\!]^{\#} D[4]$, and $T \in L[3]$

Example

```
1: T = x+1            D: {}        L: {x}
2: if (*) then {      D: {T=x+1}   L: {T}
3:   M[0]=T           D: {T=x+1}   L: {T}
4:   Nop              D: {}        L: {}
5: }                  D: {}        L: {}
```

- Placement of $T = x + 1$ before edge $(3, 4)$
  - We have $T = x + 1 \in D[3] \setminus [\![M[0] = T]\!]^{\#} D[4]$, and $T \in L[3]$

```
1:
2: if (*) then {
3:   T = x+1
x:   M[0]=T
4:   Nop
5: }
```

Summary

- PDE is generalization of DAE
  - Assignment to dead variable will not be placed
  - As variable is dead on all paths leaving that assignment

Summary

- PDE is generalization of DAE
  - Assignment to dead variable will not be placed
  - As variable is dead on all paths leaving that assignment
- May also use true liveness.

Summary

- PDE is generalization of DAE
  - Assignment to dead variable will not be placed
  - As variable is dead on all paths leaving that assignment
- May also use true liveness.
- Non degradation of performance
  - Number of assignments on each path does not increase (without proof)
  - In particular: Assignments not moved into loops (Whiteboard)

Summary

- PDE is generalization of DAE
  - Assignment to dead variable will not be placed
  - As variable is dead on all paths leaving that assignment
- May also use true liveness.
- Non degradation of performance
  - Number of assignments on each path does not increase (without proof)
  - In particular: Assignments not moved into loops (Whiteboard)
- Profits from loop inversion (Whiteboard)

Conclusion

- Design of meaningful optimization is nontrivial
- Optimizations may only be useful in connection with others
- Order of optimization matters
- Some optimizations can be iterated

A meaningful ordering

LINV	Loop inversion
ALIAS	Alias analysis
AI	Constant propagation Intervals
RE CP DAE	(Simple) redundancy elimination Copy propagation Dead assignment elimination
PRE	Partial redundancy elimination
PDE	Partially dead assignment elimination

Table of Contents

Last Lecture

- Partially dead assignments
- Started semantics with procedures

Motivation

- So far:
  - Only regarded single procedure
  - But program typically has many procedures
  - Need to be pessimistic about their effect

Motivation

- So far:
  - Only regarded single procedure
  - But program typically has many procedures
  - Need to be pessimistic about their effect
- Now:
  - Analyze effects of procedures
  - Restrict to procedures without parameters/return values
  - But with local and global variables!
  - Can emulate parameters/return values!

Extending the semantics

- Each procedure f represented by control flow graph G^f. Assume these are distinct!

Extending the semantics

- Each procedure f represented by control flow graph G^f. Assume these are distinct!
- Add edge label $f()$ for call of procedure f

Extending the semantics

- Each procedure f represented by control flow graph G^f. Assume these are distinct!
- Add edge label $f()$ for call of procedure f
- Procedure main must exist

$$
\begin{aligned}
\mathrm{Conf} &= \mathrm{Stack} \times \mathrm{Globals} \times \mathrm{Store} \\
\mathrm{Globals} &= \mathrm{Glob} \to \mathrm{Val} \\
\mathrm{Store} &= \mathrm{Addr} \to \mathrm{Val} \\
\mathrm{Stack} &= \mathrm{Frame}^{+} \\
\mathrm{Frame} &= V \times \mathrm{Locals} \\
\mathrm{Locals} &= \mathrm{Loc} \to \mathrm{Val}
\end{aligned}
$$

- where Glob are global variable names, and Loc are local variable names

Execution, small-step semantics

- $[\![e]\!](\rho_l, \rho_g) : \mathrm{Val}$. Value of expression.

Execution, small-step semantics

- $[\![e]\!](\rho_l, \rho_g) : \mathrm{Val}$. Value of expression.
- $[\![a]\!](\rho_l, \rho_g, \mu) : \mathrm{Locals} \times \mathrm{Globals} \times \mathrm{Store}$. Effect of (non-call) action.

Execution, small-step semantics

- $[\![e]\!](\rho_l, \rho_g) : \mathrm{Val}$. Value of expression.
- $[\![a]\!](\rho_l, \rho_g, \mu) : \mathrm{Locals} \times \mathrm{Globals} \times \mathrm{Store}$. Effect of (non-call) action.
- Initial configuration: $([(v_0{}^{main}, \lambda x.\ 0)], \rho_g, \mu)$

Execution, small-step semantics

- $[\![e]\!](\rho_l, \rho_g) : \mathrm{Val}$. Value of expression.
- $[\![a]\!](\rho_l, \rho_g, \mu) : \mathrm{Locals} \times \mathrm{Globals} \times \mathrm{Store}$. Effect of (non-call) action.
- Initial configuration: $([(v_0{}^{main}, \lambda x.\ 0)], \rho_g, \mu)$
- $\rightarrow \subseteq \mathrm{Conf} \times \mathrm{Conf}$

$$
\begin{aligned}
&((u, \rho_l)\sigma, \rho_g, \mu) \rightarrow ((v, \rho_l')\sigma, \rho_g', \mu') && \text{(basic)}\\
&\quad \text{if } (u, a, v) \in E \wedge [\![a]\!](\rho_l, \rho_g, \mu) = (\rho_l', \rho_g', \mu') &&\\
&((u, \rho_l)\sigma, \rho_g, \mu) \rightarrow ((v_0{}^f, \lambda x.\ 0)(v, \rho_l)\sigma, \rho_g, \mu) && \text{(call)}\\
&\quad \text{if } (u, f(), v) \in E &&\\
&((u, \_)\sigma, \rho_g, \mu) \rightarrow (\sigma, \rho_g, \mu) && \text{(return)}\\
&\quad \text{if } u \in V_{\mathrm{end}} \wedge \sigma \neq \varepsilon &&
\end{aligned}
$$

Example (factorial)

```
main():
  M[0] = fac(3)

fac(x):
  if (x <= 1) return 1
  else return x * fac(x-1)
```

Example (factorial)

```
main():
  M[0] = fac(3)

fac(x):
  if (x <= 1) return 1
  else return x * fac(x-1)
```

Translation to no arguments and return values

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(m1, -)$
$Gx : -, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(m2, -)$
$Gx : 3, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f1, x : 0)$
$(m3, -)$
$Gx : 3, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f2, x : 3)$
$(m3, -)$
$Gx : 3, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f4, x : 3)$
$(m3, -)$
$Gx : 3, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f5, x : 3)$
$(m3, -)$
$Gx : 2, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f1, x : 0)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 2, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f2, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 2, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f4, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 2, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f5, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f1, x : 0)$
$(f6, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f2, x : 1)$
$(f6, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f3, x : 1)$
$(f6, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : -, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f7, x : 1)$
$(f6, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : 1, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f6, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : 1, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f7, x : 2)$
$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : 2, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f6, x : 3)$
$(m3, -)$
$Gx : 1, Gret : 2, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(f7, x : 3)$
$(m3, -)$
$Gx : 1, Gret : 6, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(m3, -)$
$Gx : 1, Gret : 6, M[0] : -$

Example (factorial)

```
main():
m1:  Gx = 3;
m2:  fac()
m3:  M[0] = Gret
m4:

fac():
 f1: x = Gx
 f2: if (x <= 1) {
 f3:   Gret = 1
     } else {
 f4:   Gx = x-1
 f5:   fac()
 f6:   Gret = x*Gret
 f7: }
```

A run:

$(m4, -)$
$Gx : 1, Gret : 6, M[0] : 6$

Realistic Call Semantic

- On real machine, procedure call involves
  - Save registers
  - Create stack frame
    - Push parameters, return address
    - Allocate stack space for local variables
  - Jump to procedure body

Realistic Call Semantic

- On real machine, procedure call involves
  - Save registers
  - Create stack frame
    - Push parameters, return address
    - Allocate stack space for local variables
  - Jump to procedure body
- Procedure return
  - Free stack frame
    - Jump to return address
    - Remove parameters from stack
  - Restore registers
  - Handle result

Realistic Call Semantic

- On real machine, procedure call involves
  - Save registers
  - Create stack frame
    - Push parameters, return address
    - Allocate stack space for local variables
  - Jump to procedure body
- Procedure return
  - Free stack frame
    - Jump to return address
    - Remove parameters from stack
  - Restore registers
  - Handle result
- Short demo: cdecl calling convention on x86

Inlining

- Procedure call is quite expensive

```
int f(int a, int b) {
  int l = a + b
  return l + l
}

int g (int a) {
  return f(a,a)
}
```

Inlining

- Procedure call is quite expensive
- Idea: Copy procedure body to call-site

```
int f(int a, int b) {
  int l = a + b
  return l + l
}

int g (int a) {
  int l = a + a
  return l + l
}
```

Problems

- Have to keep distinct local variables

Problems

- Have to keep distinct local variables
    - Our simple language has no parameters/ returns

Problems

- Have to keep distinct local variables
  - Our simple language has no parameters/ returns
- Be careful with recursion
  - Inlining optimization might not terminate

Problems

- Have to keep distinct local variables
  - Our simple language has no parameters/ returns
- Be careful with recursion
  - Inlining optimization might not terminate
- Too much inlining of (non-recursive procedures) may blow up the code

Problems

- Have to keep distinct local variables
  - Our simple language has no parameters/ returns
- Be careful with recursion
  - Inlining optimization might not terminate
- Too much inlining of (non-recursive procedures) may blow up the code
  - Exponentially!

```
void m0() {x=x+1}
void m1() {m0();m0()}
void m2() {m1();m1()}
...
void mN() {mN-1(); mN-1()}
```

Problems

- Have to keep distinct local variables
  - Our simple language has no parameters/ returns
- Be careful with recursion
  - Inlining optimization might not terminate
- Too much inlining of (non-recursive procedures) may blow up the code
  - Exponentially!

    ```
    void m0() {x=x+1}
    void m1() {m0();m0()}
    void m2() {m1();m1()}
    ...
    void mN() {mN-1(); mN-1()}
    ```

  - Inlining everything, program gets size $O(2^N)$

Call Graph

- Graph over procedures

Call Graph

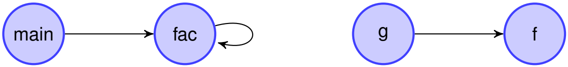
- Graph over procedures
- Edge from f to g, if body of f contains call to g

Call Graph

- Graph over procedures
- Edge from f to g, if body of f contains call to g
- In our examples

Call Graph

- Graph over procedures
- Edge from f to g, if body of f contains call to g
- In our examples



- Inline strategies

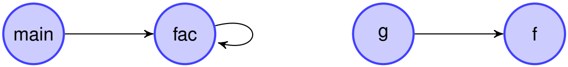
Call Graph

- Graph over procedures
- Edge from *f* to *g*, if body of *f* contains call to *g*
- In our examples

- Inline strategies
  - Leaf: Only leaf procedures

Call Graph

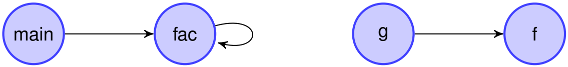
- Graph over procedures
- Edge from *f* to *g*, if body of *f* contains call to *g*
- In our examples



- Inline strategies
  - Leaf: Only leaf procedures
  - Everything: Every non-recursive procedure

Call Graph

- Graph over procedures
- Edge from f to g, if body of f contains call to g
- In our examples



- Inline strategies
  - Leaf: Only leaf procedures
  - Everything: Every non-recursive procedure
  - Real compilers use complex heuristics
    - Based on code size, register pressure, ...

Inlining transformation

- For edge $(u, f(), v)$

Inlining transformation

- For edge $(u, f(), v)$
- Make a copy of G^f, rename locals to fresh names $l_1^f, \ldots, l_n^f$

Inlining transformation

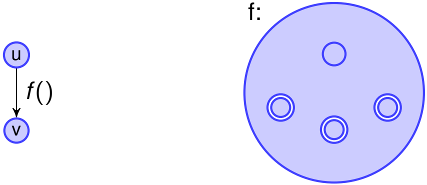
- For edge $(u, f(), v)$
- Make a copy of G^f, rename locals to fresh names $l_1^f, \ldots, l_n^f$
- Replace by edges:

Inlining transformation

- For edge $(u, f(), v)$
- Make a copy of G^f, rename locals to fresh names $l_1^f, \ldots, l_n^f$
- Replace by edges:
  - $(u, l^f = \vec{0}, v_0^f)$ (Initialize locals, goto start node of copy)

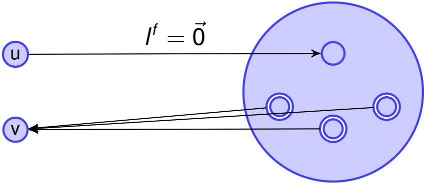
Inlining transformation

- For edge $(u, f(), v)$
- Make a copy of G^f, rename locals to fresh names $l_1^f, \dots, l_n^f$
- Replace by edges:
  - $(u, l^f = \vec{0}, v_0^f)$ (Initialize locals, goto start node of copy)
  - (v_e^f, Nop, v), for all $v_e^f \in V_{\mathrm{end}}^f$ (Link end nodes of copy with v)



Inlining transformation

- For edge $(u, f(), v)$
- Make a copy of G^f, rename locals to fresh names $l_1^f, \dots, l_n^f$
- Replace by edges:
  - $(u, l^f = \vec{0}, v_0^f)$ (Initialize locals, goto start node of copy)
  - (v_e^f, Nop, v), for all $v_e^f \in V_{\mathrm{end}}^f$ (Link end nodes of copy with v)



Tail call optimization

- Idea: If after recursive call, the procedure returns
- Re-use the procedure's stack frame, instead of allocating a new one

```
void f() {
  if (Gi < Gn-1) {
    t = a[Gi]
    Gi = Gi+1
    a[Gi]=a[Gi]+t
    f()
  }
}
```

Tail call optimization

- Idea: If after recursive call, the procedure returns
- Re-use the procedure's stack frame, instead of allocating a new one

```
void f() {
  if (Gi < Gn-1) {
    t = a[Gi]
    Gi = Gi+1
    a[Gi]=a[Gi]+t
    f()
  }
}
```

$\mapsto$

```
void f() {
  if (Gi < Gn-1) {
    t = a[Gi]
    Gi = Gi+1
    a[Gi]=a[Gi]+t
    t=0; goto f
  }
}
```

Tail call optimization

- Idea: If after recursive call, the procedure returns
- Re-use the procedure's stack frame, instead of allocating a new one

```
void f() {
  if (Gi < Gn-1) {
    t = a[Gi]
    Gi = Gi+1
    a[Gi]=a[Gi]+t
    f()
  }
}
```

$\mapsto$

```
void f() {
  if (Gi < Gn-1) {
    t = a[Gi]
    Gi = Gi+1
    a[Gi]=a[Gi]+t
    t=0; goto f
  }
}
```

- Requires no code duplication

Tail call optimization

- Idea: If after recursive call, the procedure returns
- Re-use the procedure's stack frame, instead of allocating a new one

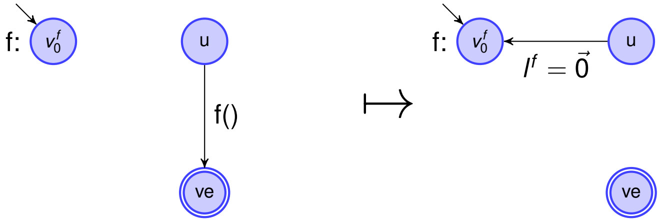
```
void f() {
  if (Gi < Gn-1) {
    t = a[Gi]
    Gi = Gi+1
    a[Gi]=a[Gi]+t
    f()
  }
}
```

$\mapsto$

```
void f() {
  if (Gi < Gn-1) {
    t = a[Gi]
    Gi = Gi+1
    a[Gi]=a[Gi]+t
    t=0; goto f
  }
}
```

- Requires no code duplication
- Have to re-initialize local variables, according to semantics
  - Target for DAE ;)

Tail-Call Transformation



Discussion

- Crucial optimization for languages without loop construct
  - E.g., functional languages

Discussion

- Crucial optimization for languages without loop construct
  - E.g., functional languages
- No duplication of code or additional local variables

Discussion

- Crucial optimization for languages without loop construct
  - E.g., functional languages
- No duplication of code or additional local variables
- The optimization may also be profitable for non-recursive calls
  - Re-use stack-space of current frame for new stack frame
  - But not expressable in our semantics (Too high-level view on locals)

Interprocedural Analysis

- Want to extend our program analysis to procedures
- For example, constant propagation

Interprocedural Analysis

- Want to extend our program analysis to procedures
- For example, constant propagation

```
main() { int t;
  t = 0;
  if (t) M[17] = 3;
  a1 = t;
  work ();
  ret = 1 - ret;
}

work() {
  if (a1) work();
  ret = a1 ;
}
```

Interprocedural Analysis

- Want to extend our program analysis to procedures
- For example, constant propagation

```
main() { int t;
  t = 0;
  if (t) M[17] = 3;
  a1 = t;
  work ();
  ret = 1 - ret;
}

work() {
  if (a1) work();
  ret = a1 ;
}
```

$\mapsto$

```
main() { int t;
  t = 0;
  //if (t) M[17] = 3;
  a1 = 0;
  work0 ();
  ret = 1;
}

work0() {
  //if (a1) work();
  ret = 0 ;
}
```

Last Lecture

- Stack-based semantics with procedures
- Inlining optimization
- Tail-call optimization
- Path-based semantics

Generalization of Paths

- Recall: Paths were sequences of actions

$$\text{path} = \varepsilon \mid \text{Act} \cdot \text{path}$$

Generalization of Paths

- Recall: Paths were sequences of actions

$$\text{path} = \varepsilon \mid \text{Act} \cdot \text{path}$$

- Now: We can call procedures. A procedure call may
  - Return on path
  - Not return on path

Generalization of Paths

- Recall: Paths were sequences of actions

$$\text{path} = \varepsilon \mid \text{Act} \cdot \text{path}$$

- Now: We can call procedures. A procedure call may
  - Return on path
  - Not return on path
  - Advantageous to make this visible in path structure

$$\begin{aligned}\text{slpath} &= \varepsilon \mid \text{Act} \cdot \text{slpath} \mid f(\text{slpath}) \cdot \text{slpath} \\ \text{path} &= \varepsilon \mid \text{Act} \cdot \text{path} \mid f(\text{slpath}) \cdot \text{path} \mid f_{<} \cdot \text{path}\end{aligned}$$

Generalization of Paths

- Recall: Paths were sequences of actions

$$\text{path} = \varepsilon \mid \text{Act} \cdot \text{path}$$

- Now: We can call procedures. A procedure call may
  - Return on path
  - Not return on path
  - Advantageous to make this visible in path structure

$$\begin{aligned} \text{slpath} &= \varepsilon \mid \text{Act} \cdot \text{slpath} \mid f(\text{slpath}) \cdot \text{slpath} \\ \text{path} &= \varepsilon \mid \text{Act} \cdot \text{path} \mid f(\text{slpath}) \cdot \text{path} \mid f_< \cdot \text{path} \end{aligned}$$

- Intuitively:
  - $f(\pi)$: Call to procedure f, which executes π and returns
  - $f_<$: Call to procedure f, which does not return
  - slpath: Same level paths, which end on same stack-level as they begin
  - Note: Inside returning call, all calls must return.

Generalization of Paths

- Recall: Paths between nodes

$$[empty]\,\frac{-}{u \xrightarrow{\varepsilon} u} \qquad\qquad [app]\,\frac{k = (u, a, v) \in E \quad v \xrightarrow{\pi} w}{u \xrightarrow{k\pi} w}$$

Generalization of Paths

- Recall: Paths between nodes

$$[empty]\,\frac{-}{u \xrightarrow{\varepsilon} u} \qquad\qquad [app]\,\frac{k = (u, a, v) \in E \quad v \xrightarrow{\pi} w}{u \xrightarrow{k\pi} w}$$

- Now

$$[empty]\,\frac{-}{u \xrightarrow{\varepsilon}_{\mathrm{sl}} u} \qquad [app]\,\frac{k = (u, a, v) \in E \quad v \xrightarrow{\pi}_{\mathrm{sl}} w}{u \xrightarrow{k\pi}_{\mathrm{sl}} w}$$

$$[call]\,\frac{(u, f(), v) \in E \quad {v_0}^f \xrightarrow{\pi_1}_{\mathrm{sl}} v_e^f \in V_{\mathrm{end}} \quad v \xrightarrow{\pi_2}_{\mathrm{sl}} w}{u \xrightarrow{f(\pi_1)\pi_2}_{\mathrm{sl}} w}$$

Generalization of Paths

- Recall: Paths between nodes

$$[empty]\,\frac{-}{u \xrightarrow{\varepsilon} u} \qquad [app]\,\frac{k = (u,a,v) \in E \quad v \xrightarrow{\pi} w}{u \xrightarrow{k\pi} w}$$

- Now

$$[empty]\,\frac{-}{u \xrightarrow{\varepsilon}_{\mathrm{sl}} u} \qquad [app]\,\frac{k = (u,a,v) \in E \quad v \xrightarrow{\pi}_{\mathrm{sl}} w}{u \xrightarrow{k\pi}_{\mathrm{sl}} w}$$

$$[call]\,\frac{(u, f(), v) \in E \quad {v_0}^f \xrightarrow{\pi_1}_{\mathrm{sl}} v_e^f \in V_{\mathrm{end}} \quad v \xrightarrow{\pi_2}_{\mathrm{sl}} w}{u \xrightarrow{f(\pi_1)\pi_2}_{\mathrm{sl}} w}$$

- And

$$[emp]\,\frac{-}{u \xrightarrow{\varepsilon} u} \qquad [app]\,\frac{k = (u,a,v) \in E \quad v \xrightarrow{\pi} w}{u \xrightarrow{k\pi} w}$$

$$[call]\,\frac{(u, f(), v) \in E \quad {v_0}^f \xrightarrow{\pi_1}_{\mathrm{sl}} v_e^f \in V_{\mathrm{end}} \quad v \xrightarrow{\pi_2} w}{u \xrightarrow{f(\pi_1)\pi_2} w}$$

$$[ncall]\,\frac{(u, f(), v) \in E \quad {v_0}^f \xrightarrow{\pi} w}{u \xrightarrow{f_{<}\pi} w}$$

Executions of paths

- Recall

$$[\![\varepsilon]\!]s = s \quad [\![k\pi]\!]s = [\![\pi]\!]([\![k]\!]s)$$

Executions of paths

- Recall

$$[\![\varepsilon]\!]s = s \quad [\![k\pi]\!]s = [\![\pi]\!]([\![k]\!]s)$$

- Now

$$[\![\varepsilon]\!]s = s \quad [\![k\pi]\!]s = [\![\pi]\!]([\![k]\!]s)$$

$$[\![f(\pi)]\!]s = H\ [\![\pi]\!]\ s \quad [\![f_<]\!]s = \text{enter}\ s$$

where

$$\begin{aligned} \text{enter}(\rho_l, \rho_g, \mu) &:= (\vec{0}, \rho_g, \mu) \\ \text{combine}((\rho_l, \rho_g, \mu), (\rho_l', \rho_g', \mu')) &:= (\rho_l, \rho_g', \mu') \\ H\ e\ s &:= \text{combine}(s, (e(\text{enter}\ s))) \end{aligned}$$

Executions of paths

- Recall

$$[\![\varepsilon]\!]s = s \quad [\![k\pi]\!]s = [\![\pi]\!]([\![k]\!]s)$$

- Now

$$[\![\varepsilon]\!]s = s \quad [\![k\pi]\!]s = [\![\pi]\!]([\![k]\!]s)$$

$$[\![f(\pi)]\!]s = H\ [\![\pi]\!]\ s \quad [\![f_<]\!]s = \text{enter } s$$

where

$$\begin{aligned}
\text{enter}(\rho_l, \rho_g, \mu) &:= (\vec{0}, \rho_g, \mu) \\
\text{combine}((\rho_l, \rho_g, \mu), (\rho'_l, \rho'_g, \mu')) &:= (\rho_l, \rho'_g, \mu') \\
H\ e\ s &:= \text{combine}(s, (e(\text{enter } s)))
\end{aligned}$$

- Intuition:

enter	Set up stack frame
combine	Combine procedure result with old frame

Example

```
f () {
  if x>0 then {
    x = x - 1
    f ()
    x = x + 1
  } else {
    u: Nop
  }
}

main () {
  x = 1;
  f ()
  x = 0
}
```

Example

```
f () {
  if x>0 then {
    x = x - 1
    f ()
    x = x + 1
  } else {
    u: Nop
  }
}

main () {
  x = 1;
  f ()
  x = 0
}
```

SL-path through main

```
x=1
f(
  Pos(x>0)
  x=x-1
  f(
    Neg(x>0)
    Nop
  )
  x = x + 1
)
x = 0
```

Example

```
f () {
  if x>0 then {
    x = x - 1
    f ()
    x = x + 1
  } else {
    u: Nop
  }
}

main () {
  x = 1;
  f ()
  x = 0
}
```

SL-path through main

```
x=1
f(
  Pos(x>0)
  x=x-1
  f(
    Neg(x>0)
    Nop
  )
  x = x + 1
)
x = 0
```

Path from main to u

```
x=1
f<
  Pos(x>0)
  x=x-1
  f<
    Neg(x>0)
```

Equivalence of semantics

Theorem

The stack-based and path-based semantics are equivalent:

$$(\exists \sigma.\ ([u, \rho_l], \rho_g, \mu) \to^* ([v, \rho_l']\sigma, \rho_g', \mu')) \iff (\exists \pi.\ u \xrightarrow{\pi} v \wedge [\![\pi]\!](\rho_l, \rho_g, \mu) = (\rho_l', \rho_g', \mu'))$$

Proof sketch (Whiteboard)

- Auxiliary lemma: Same-level paths

$$
\begin{aligned}
&(([u, \rho_l], \rho_g, \mu)) \rightarrow^* ([v, \rho_l'], \rho_g', \mu') \\
&\qquad \iff (\exists \pi.\ u \xrightarrow{\pi}_{\mathrm{sl}} v \wedge [\![\pi]\!](\rho_l, \rho_g, \mu) = (\rho_l', \rho_g', \mu'))
\end{aligned}
$$

Proof sketch (Whiteboard)

- Auxiliary lemma: Same-level paths

$$(([u, \rho_l], \rho_g, \mu)) \rightarrow^* ([v, \rho_l'], \rho_g', \mu')$$
$$\iff (\exists \pi.\ u \xrightarrow{\pi}_{\mathrm{sl}} v \wedge [\![\pi]\!](\rho_l, \rho_g, \mu) = (\rho_l', \rho_g', \mu'))$$

- Main ideas ($\Longrightarrow$)
  - Induction on length of execution
  - Identify non-returning calls:
    - Execution in between yields same-level paths (aux-lemma)

Proof sketch (Whiteboard)

- Auxiliary lemma: Same-level paths

$$(([u, \rho_l], \rho_g, \mu)) \rightarrow^* ([v, \rho_l'], \rho_g', \mu')$$
$$\iff (\exists \pi.\ u \xrightarrow{\pi}_{\mathrm{sl}} v \wedge [\![\pi]\!](\rho_l, \rho_g, \mu) = (\rho_l', \rho_g', \mu'))$$

- Main ideas ($\implies$)
  - Induction on length of execution
  - Identify non-returning calls:
    - Execution in between yields same-level paths (aux-lemma)
- Main ideas ($\impliedby$)
  - Induction on path structure
  - Executions can be repeated with stack extended at the bottom

$$(\sigma, \rho_g, \mu) \rightarrow^* (\sigma', \rho_g', \mu') \implies (\sigma\hat{\sigma}, \rho_g, \mu) \rightarrow^* (\sigma'\hat{\sigma}, \rho_g', \mu')$$

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^{\#} : \mathbb{D} \to \mathbb{D}$

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^{\#} : \mathbb{D} \to \mathbb{D}$
  - Actions: Nop, Test, Assign, Load, Store

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
  - Actions: Nop, Test, Assign, Load, Store
- Now: Additional actions: Returning/non-returning procedure call

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
  - Actions: Nop, Test, Assign, Load, Store
- Now: Additional actions: Returning/non-returning procedure call
- Require: Abstract effects for $f(\pi)$ and $f_<$

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
  - Actions: Nop, Test, Assign, Load, Store
- Now: Additional actions: Returning/non-returning procedure call
- Require: Abstract effects for $f(\pi)$ and $f_<$
  - Define abstract $\mathrm{enter}_f^\#$, $\mathrm{combine}_f^\#$

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^{\#} : \mathbb{D} \to \mathbb{D}$
  - Actions: Nop, Test, Assign, Load, Store
- Now: Additional actions: Returning/non-returning procedure call
- Require: Abstract effects for $f(\pi)$ and $f_<$
  - Define abstract $\mathrm{enter}_f^{\#}$, $\mathrm{combine}_f^{\#}$
  - $H_f^{\#}\ e\ d = \mathrm{combine}_f^{\#}(d, e(\mathrm{enter}_f^{\#}(d)))$

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^{\#} : \mathbb{D} \to \mathbb{D}$
  - Actions: Nop, Test, Assign, Load, Store
- Now: Additional actions: Returning/non-returning procedure call
- Require: Abstract effects for $f(\pi)$ and $f_<$
  - Define abstract $\mathrm{enter}_f^{\#}$, $\mathrm{combine}_f^{\#}$
  - $H_f^{\#}\ e\ d = \mathrm{combine}_f^{\#}(d, e(\mathrm{enter}_f^{\#}(d)))$
  - $[\![f(\pi)]\!]^{\#} d = H_f^{\#}\ [\![\pi]\!]^{\#}\ d$

Abstraction of paths

- Recall: Abstract effects of actions: $[\![a]\!]^\# : \mathbb{D} \to \mathbb{D}$
  - Actions: Nop, Test, Assign, Load, Store
- Now: Additional actions: Returning/non-returning procedure call
- Require: Abstract effects for $f(\pi)$ and $f_<$
  - Define abstract $\mathrm{enter}_f^\#$, $\mathrm{combine}_f^\#$
  - $H_f^\#\ e\ d = \mathrm{combine}_f^\#(d, e(\mathrm{enter}_f^\#(d)))$
  - $[\![f(\pi)]\!]^\# d = H_f^\#\ [\![\pi]\!]^\#\ d$
  - $[\![f_<]\!]^\# d = \mathrm{enter}_f^\#(d)$

Example: Copy constants

- Simplified constant propagation

Example: Copy constants

- Simplified constant propagation
  - Conditions not exploited

Example: Copy constants

- Simplified constant propagation
  - Conditions not exploited
  - Only assignments of form $x = y$ and $x = c, c \in \mathbb{Z}$

Example: Copy constants

- Simplified constant propagation
  - Conditions not exploited
  - Only assignments of form $x = y$ and $x = c, c \in \mathbb{Z}$
- Domain: $\mathbb{D} := \mathrm{Reg} \to \mathbb{Z}^{\top}$

Example: Copy constants

- Simplified constant propagation
  - Conditions not exploited
  - Only assignments of form $x = y$ and $x = c, c \in \mathbb{Z}$
- Domain: $\mathbb{D} := \mathrm{Reg} \to \mathbb{Z}^\top$
- Initially: $d_0\ l := 0, l \in \mathrm{Loc}$, $d_0\ g := \top, g \in \mathrm{Glob}$

Example: Copy constants

- Simplified constant propagation
  - Conditions not exploited
  - Only assignments of form $x = y$ and $x = c, c \in \mathbb{Z}$
- Domain: $\mathbb{D} := \mathrm{Reg} \to \mathbb{Z}^{\top}$
- Initially: $d_0\ l := 0, l \in \mathrm{Loc},\ d_0\ g := \top, g \in \mathrm{Glob}$
- Abstract effects

$$
\begin{aligned}
[\![x := c]\!]^{\#} d &= d(x := c) \quad \text{for } c \in \mathbb{Z} \\
[\![x := y]\!]^{\#} d &= d(x := d(y)) \quad \text{for } y \in \mathrm{Reg} \\
[\![x := e]\!]^{\#} d &= d(x := \top) \quad \text{for } e \in \mathrm{Expr} \setminus (\mathbb{Z} \cup \mathrm{Reg}) \\
[\![x := M(e)]\!]^{\#} d &= d(x := \top) \\
[\![\mathrm{Pos}(e)]\!]^{\#} d &= [\![\mathrm{Neg}(e)]\!]^{\#} d = [\![\mathrm{Nop}]\!]^{\#} d = [\![M(e_1) = e_2]\!]^{\#} d = d \\
\mathrm{enter}_f^{\#} d &= d(l := 0 \mid l \in \mathrm{Loc}) \\
\mathrm{combine}_f^{\#}\ d\ d' &= \lambda x.\ x \in \mathrm{Loc} ? d(x) : d'(x)
\end{aligned}
$$

Correctness

- Description relation $(\rho_l, \rho_g, \mu) \Delta d$
  - iff $\rho_l \Delta d|_{\text{Loc}}$ and $\rho_g \Delta d|_{\text{Glob}}$

Correctness

- Description relation $(\rho_l, \rho_g, \mu)\ \Delta\ d$
  - iff $\rho_l\ \Delta\ d|_{\mathrm{Loc}}$ and $\rho_g\ \Delta\ d|_{\mathrm{Glob}}$
- Show: $\forall \rho_g, \mu.\ [\![\pi]\!](\vec{0}, \rho_g, \mu)\ \Delta\ [\![\pi]\!]^{\#} d_0$

Correctness

- Description relation $(\rho_l, \rho_g, \mu) \; \Delta \; d$
  - iff $\rho_l \; \Delta \; d|_{\mathrm{Loc}}$ and $\rho_g \; \Delta \; d|_{\mathrm{Glob}}$
- Show: $\forall \rho_g, \mu. \; [\![\pi]\!](\vec{0}, \rho_g, \mu) \; \Delta \; [\![\pi]\!]^{\#} d_0$
  - By induction on path

Correctness

- Description relation $(\rho_l, \rho_g, \mu) \;\Delta\; d$
  - iff $\rho_l \;\Delta\; d|_{\mathrm{Loc}}$ and $\rho_g \;\Delta\; d|_{\mathrm{Glob}}$
- Show: $\forall \rho_g, \mu.\ [\![\pi]\!](\vec{0}, \rho_g, \mu) \;\Delta\; [\![\pi]\!]^{\#} d_0$
  - By induction on path
  - Then, case distinction on edges

Correctness

- Description relation $(\rho_l, \rho_g, \mu) \; \Delta \; d$
  - iff $\rho_l \; \Delta \; d|_{\mathrm{Loc}}$ and $\rho_g \; \Delta \; d|_{\mathrm{Glob}}$
- Show: $\forall \rho_g, \mu.\ [\![\pi]\!](\vec{0}, \rho_g, \mu) \; \Delta \; [\![\pi]\!]^{\#} d_0$
  - By induction on path
  - Then, case distinction on edges
  - Generalization of simulation proofs for intraprocedural case

Computing Solutions

- Interested in $\mathrm{MOP}[u] := \bigsqcup \{ [\![\pi]\!]^\# d_0 \mid v_0{}^{\mathrm{m}ain} \xrightarrow{\pi} u \}$

Computing Solutions

- Interested in $\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^\# d_0 \mid {v_0}^{\mathrm{main}} \xrightarrow{\pi} u\}$
- Idea: Constraint system for same-level effects of functions

$$
\begin{array}{rlr}
S[{v_0}^f] \sqsupseteq \mathrm{id} & & \text{(start)} \\
S[v] \sqsupseteq [\![k]\!]^\# \circ S[u] & k = (u, a, v) \in E & \text{(edge)} \\
S[v] \sqsupseteq H^\#(S[f]) \circ S[u] & k = (u, f(), v) \in E & \text{(call)} \\
S[f] \sqsupseteq S[v_e^f] & v_e^f \in V_{\mathrm{end}}^f & \text{(end)}
\end{array}
$$

Computing Solutions

- Interested in $\mathrm{MOP}[u] := \bigsqcup\{[\![\pi]\!]^{\#} d_0 \mid v_0{}^{\mathrm{main}} \xrightarrow{\pi} u\}$
- Idea: Constraint system for same-level effects of functions

$$
\begin{array}{rll}
S[{v_0}^f] \sqsupseteq \mathrm{id} & & \text{(start)} \\
S[v] \sqsupseteq [\![k]\!]^{\#} \circ S[u] & k = (u, a, v) \in E & \text{(edge)} \\
S[v] \sqsupseteq H^{\#}(S[f]) \circ S[u] & k = (u, f(), v) \in E & \text{(call)} \\
S[f] \sqsupseteq S[v_e^f] & v_e^f \in V_{\mathrm{end}}^f & \text{(end)}
\end{array}
$$

- And for effects of paths reaching u

$$
\begin{array}{rll}
R[{v_0}^{\mathrm{main}}] \sqsupseteq \mathrm{enter}^{\#}\ d_0 & & \text{(start)} \\
R[v] \sqsupseteq [\![k]\!]^{\#} R[u] & k = (u, a, v) \in E & \text{(edge)} \\
R[v] \sqsupseteq H^{\#} S[f]\ R[u] & k = (u, f(), v) \in E & \text{(call)} \\
R[{v_0}^f] \sqsupseteq \mathrm{enter}^{\#} R[u] & (u, f(), v) \in E & \text{(calln)}
\end{array}
$$

Coincidence Theorems

- Let MFP be the least solution of R, then we have

  $$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

  - For monotonic effects

Coincidence Theorems

- Let MFP be the least solution of R, then we have

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

  - For monotonic effects
- If each program point is reachable, and all effects as well as $H^{\#}$ are distributive:

$$\mathrm{MOP} = \mathrm{MFP}$$

Coincidence Theorems

- Let MFP be the least solution of R, then we have

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

  - For monotonic effects
- If each program point is reachable, and all effects as well as $H^\#$ are distributive:

$$\mathrm{MOP} = \mathrm{MFP}$$

- Generalization of corresponding intra-procedural theorems

Coincidence Theorems

- Let MFP be the least solution of R, then we have

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

  - For monotonic effects
- If each program point is reachable, and all effects as well as $H^\#$ are distributive:

$$\mathrm{MOP} = \mathrm{MFP}$$

- Generalization of corresponding intra-procedural theorems
- Intuition: Constraint system joins early

Coincidence Theorems

- Let MFP be the least solution of R, then we have

$$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

  - For monotonic effects
- If each program point is reachable, and all effects as well as $H^{\#}$ are distributive:

$$\mathrm{MOP} = \mathrm{MFP}$$

- Generalization of corresponding intra-procedural theorems
- Intuition: Constraint system joins early
  - Information from multiple incoming edges

Coincidence Theorems

- Let MFP be the least solution of R, then we have

  $$\mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

  - For monotonic effects
- If each program point is reachable, and all effects as well as $H^{\#}$ are distributive:

  $$\mathrm{MOP} = \mathrm{MFP}$$

- Generalization of corresponding intra-procedural theorems
- Intuition: Constraint system joins early
  - Information from multiple incoming edges
  - All paths through procedure on returning call

Remaining problem

- How to compute effects of call efficiently?

Remaining problem

- How to compute effects of call efficiently?
    - How to represent functions $\mathbb{D} \to \mathbb{D}$

Remaining problem

- How to compute effects of call efficiently?
  - How to represent functions $\mathbb{D} \to \mathbb{D}$
  - efficiently?

Remaining problem

- How to compute effects of call efficiently?
  - How to represent functions $\mathbb{D} \to \mathbb{D}$
  - efficiently?
- For copy constants:

Remaining problem

- How to compute effects of call efficiently?
  - How to represent functions $\mathbb{D} \to \mathbb{D}$
  - efficiently?
- For copy constants:
  - Domain is actually finite: Only need to consider constants that actually occur in the program

Remaining problem

- How to compute effects of call efficiently?
  - How to represent functions $\mathbb{D} \to \mathbb{D}$
  - efficiently?
- For copy constants:
  - Domain is actually finite: Only need to consider constants that actually occur in the program
  - But this would yield huge tables for functions

Remaining problem

- How to compute effects of call efficiently?
  - How to represent functions $\mathbb{D} \to \mathbb{D}$
  - efficiently?
- For copy constants:
  - Domain is actually finite: Only need to consider constants that actually occur in the program
  - But this would yield huge tables for functions
- Possible solutions:

Remaining problem

- How to compute effects of call efficiently?
    - How to represent functions $\mathbb{D} \to \mathbb{D}$
    - efficiently?
- For copy constants:
    - Domain is actually finite: Only need to consider constants that actually occur in the program
    - But this would yield huge tables for functions
- Possible solutions:
    - Find efficient representation for functions

Remaining problem

- How to compute effects of call efficiently?
  - How to represent functions $\mathbb{D} \to \mathbb{D}$
  - efficiently?
- For copy constants:
  - Domain is actually finite: Only need to consider constants that actually occur in the program
  - But this would yield huge tables for functions
- Possible solutions:
  - Find efficient representation for functions
  - Function actually not applied to all values $d \in \mathbb{D}$. $\implies$ compute on demand.

Efficient representation of same-level effects

- Observation: Functions $S[u] \neq \bot$ are of form $\langle m \rangle$ where

$$\langle m \rangle := \lambda D\, x.\ m_1\ x \sqcup \bigsqcup_{y \in m_2\ x} D\ y$$

  - $m_1\ x : \mathbb{Z}_\bot^\top$ - Join of constants that may be assigned to x
  - $m_2\ x : 2^{\mathrm{Reg}}$ - set of variables that may be assigned to x (non-empty)

Efficient representation of same-level effects

- Observation: Functions $S[u] \neq \bot$ are of form $\langle m \rangle$ where

$$\langle m \rangle := \lambda D\, x.\; m_1\; x \sqcup \bigsqcup_{y \in m_2\; x} D\; y$$

  - $m_1\; x : \mathbb{Z}_\bot^\top$ - Join of constants that may be assigned to x
  - $m_2\; x : 2^{\mathrm{Reg}}$ - set of variables that may be assigned to x (non-empty)
- Let $F := \{\langle m \rangle | m : \mathrm{Reg} \to \mathbb{Z}_\bot^\top \times 2^{\mathrm{Reg}}\}$ be the set of those functions

Efficient representation of same-level effects

- Observation: Functions $S[u] \neq \bot$ are of form $\langle m \rangle$ where

$$\langle m \rangle := \lambda D\, x.\; m_1\; x \sqcup \bigsqcup_{y \in m_2\; x} D\; y$$

  - $m_1\; x : \mathbb{Z}_\bot^\top$ - Join of constants that may be assigned to x
  - $m_2\; x : 2^{\mathrm{Reg}}$ - set of variables that may be assigned to x (non-empty)
- Let $F := \{\langle m \rangle | m : \mathrm{Reg} \to \mathbb{Z}_\bot^\top \times 2^{\mathrm{Reg}}\}$ be the set of those functions
- To show: id, $[\![a]\!]^\# \in F$, and F closed under $\circ$, $\sqcup$, $\mathrm{enter}^\#$, and $H^\#$

Identity and effects representable

$$\text{id} = \langle \lambda x.\ (\bot, \{x\}) \rangle$$

$$[\![x := e]\!]^\# = \begin{cases} \langle \mathrm{id}(x \mapsto (c, \emptyset)) \rangle & \text{for } e = c \in \mathbb{Z} \\ \langle \mathrm{id}(x \mapsto (\bot, \{y\})) \rangle & \text{for } e = y \in \mathrm{Reg} \\ \langle \mathrm{id}(x \mapsto (\top, \emptyset)) \rangle & \text{otherwise} \end{cases}$$

Identity and effects representable

$$\mathsf{id} = \langle \lambda x.\ (\bot, \{x\}) \rangle$$

$$[\![x := e]\!]^\# = \begin{cases} \langle \mathrm{id}(x \mapsto (c, \emptyset)) \rangle & \text{for } e = c \in \mathbb{Z} \\ \langle \mathrm{id}(x \mapsto (\bot, \{y\})) \rangle & \text{for } e = y \in \mathrm{Reg} \\ \langle \mathrm{id}(x \mapsto (\top, \emptyset)) \rangle & \text{otherwise} \end{cases}$$

- Effects of other actions similarly

Closed under function composition and join

$$\langle m \rangle \circ \langle m' \rangle = \langle \lambda x.\ (m_1\ x \sqcup \bigsqcup_{y \in m_2\ x} m'_1\ y,\ \bigcup_{y \in m_2\ x} m'_2\ y) \rangle$$

$$\langle m \rangle \sqcup \langle m' \rangle = \langle m \sqcup m' \rangle$$

Closed under function composition and join

$$\langle m \rangle \circ \langle m' \rangle = \langle \lambda x.\ (m_1\ x \sqcup \bigsqcup_{y \in m_2\ x} m_1'\ y, \bigcup_{y \in m_2\ x} m_2'\ y) \rangle$$

$$\langle m \rangle \sqcup \langle m' \rangle = \langle m \sqcup m' \rangle$$

- Intuition: Assigned constants by m_1, or by m_1', and variable goes through m_2
  - $[\![x := c; foo]\!]^\#$, or $[\![x := y; y := c]\!]^\#$
  - Note: If x not touched, we have $m_2\ x = \{x\}$
- Note: $\sqcup$ defined pointwise: $(m \sqcup m')\ x = (m_1\ x \sqcup m_1'\ x, m_2\ x \cup m_2'\ x)$

Closed under $\mathrm{enter}^{\#}$ and $H^{\#}$

$$\begin{aligned}
\mathrm{enter}^{\#} &= \langle(\lambda x.\ (0, \emptyset))|_{\mathrm{Loc}}\rangle \oplus \mathrm{id}|_{\mathrm{Glob}} \\
H^{\#}(\langle m\rangle) &= \mathrm{id}|_{\mathrm{Loc}} \oplus (\langle m\rangle \circ \mathrm{enter}^{\#})|_{\mathrm{Glob}} \\
\langle m\rangle|_{\mathrm{Loc}} \oplus \langle m'\rangle|_{\mathrm{Glob}} &:= \langle \lambda x.\ x \in \mathrm{Loc} ? m\ x : m'\ x\rangle
\end{aligned}$$

Closed under $\mathrm{enter}^{\#}$ and $H^{\#}$

$$\mathrm{enter}^{\#} = \langle(\lambda x.\ (0, \emptyset))|_{\mathrm{Loc}}\rangle \oplus \mathrm{id}|_{\mathrm{Glob}}$$

$$H^{\#}(\langle m\rangle) = \mathrm{id}|_{\mathrm{Loc}} \oplus (\langle m\rangle \circ \mathrm{enter}^{\#})|_{\mathrm{Glob}}$$

$$\langle m\rangle|_{\mathrm{Loc}} \oplus \langle m'\rangle|_{\mathrm{Glob}} := \langle\lambda x.\ x \in \mathrm{Loc} ? m\ x : m'\ x\rangle$$

- Intuition
  - Function call only affects globals
  - $\mathrm{enter}^{\#}$ is effect of entering function (set locals to 0)
  - $f_{\mathrm{Loc}} \oplus f'_{\mathrm{Glob}}$ - Use f for local variables, f' for global variables

Recall initial example

```
main() { int t;
  t = 0;              // t=0, a1=⊤, ret=⊤
  if (t)              // t=0, a1=⊤, ret=⊤
    M[17] = 3;        // t=0, a1=⊤, ret=⊤
  a1 = t;             // t=0, a1=⊤, ret=⊤
  work ();            // t=0, a1=0, ret=⊤
  ret = 1 - ret;      // t=0, a1=0, ret=0
}                     // t=0, a1=0, ret=⊤

work() {
  if (a1) {        // id                                a1=0, ret=⊤
    work()         // id                                a1=0, ret=⊤
    Nop }          // id[ ret->(⊥,{a1}) ]            a1=0, ret=0
  ret = a1 ;       // id[ ret->(⊥,{ret,a1}) ]  a1=0, ret=⊤
}                  // id[ ret->(⊥,{a1}) ]            a1=0, ret=0
```

Discussion

- At least copy-constants can be determined interprocedurally
- For that, we had to ignore conditions and complex assignments
- However, for the reaching paths, we could have been more precise
- Extra abstractions were required as
  1. Set of abstract same-level effects must be finite
  2. and efficiently implementable

Last Lecture

- Copy-Constant propagation
- Functional approach to interprocedural analysis
  - Compute same-level effects by constraint system
  - Find efficient representation for same-level effects

Idea: Evaluation on demand

- Procedures often called only for few distinct abstract arguments
  - Observed early (Sharir/Pneuli'81, Cousot'77)
- Only analyze procedures for these
- Intuition: $[\![f, a]\!]^\#$ - effect of f if called in abstract state a
- Put up constraint system

$$
\begin{array}{rcll}
[\![{v_0}^f, a]\!]^\# & \sqsupseteq & a & \\
[\![v, a]\!]^\# & \sqsupseteq & [\![k]\!]^\#([\![u, a]\!]^\#) & \text{for basic edge } k = (u, -, v) \\
[\![v, a]\!]^\# & \sqsupseteq & \mathrm{combine}^\#([\![u, a]\!]^\#, [\![g, \mathrm{enter}^\#([\![u, a]\!]^\#)]\!]^\#) & \text{for call edge } k = (u, g(), v) \\
[\![f, a]\!]^\# & \sqsupseteq & [\![v_e^f, a]\!]^\# & \text{for } v_e^f \in V_{\mathrm{end}}^f
\end{array}
$$

- Idea: Keep track of effect for any node of procedure

Evaluation on demand

- This constraint system may be huge
- Idea: Only evaluate $[\![f, a]\!]^{\#}$ for values a that actually occur
  - Local fixed-point algorithms (not covered)
- But, we can do an example nevertheless :)

Example: Full constant propagation

```
// a1,ret | locals
main() { int t;
  t = 0;              ⊤,⊤ | 0
  if (t)              ⊤,⊤ | 0
    M[17] = 3;        ⊥
  a1 = t;             ⊤,⊤ | 0
  work ();            0,⊤ | 0
  ret = 1 - ret;      0,0 | 0
}                     0,1 | 0

work() {              [[work,(0,⊤)]]#
  if (a1)             0,⊤
    work()            ⊥
  ret = a1 ;          0,⊤
}                     0,0
```

- Only need to keep track of a_1 for calling context of `work`

Discussion

- This analysis terminates, if

Discussion

- This analysis terminates, if
  - $\mathbb{D}$ has finite height,

Discussion

- This analysis terminates, if
  - $\mathbb{D}$ has finite height,
  - and every procedure only analyzed for finitely many arguments

Discussion

- This analysis terminates, if
  - $\mathbb{D}$ has finite height,
  - and every procedure only analyzed for finitely many arguments
- Analogous algorithms have proved efficient for analysis of PROLOG

Discussion

- This analysis terminates, if
  - $\mathbb{D}$ has finite height,
  - and every procedure only analyzed for finitely many arguments
- Analogous algorithms have proved efficient for analysis of PROLOG
- Together with points-to analysis, algorithms of this kind used in the Goblint-Tool

Discussion

- This analysis terminates, if
  - $\mathbb{D}$ has finite height,
  - and every procedure only analyzed for finitely many arguments
- Analogous algorithms have proved efficient for analysis of PROLOG
- Together with points-to analysis, algorithms of this kind used in the Goblint-Tool
  - Data-race detection for C with POSIX-Threads

Crude approximation

- Start with very crude approximation:

Crude approximation

- Start with very crude approximation:
  - Just insert edges from function-call to procedure start

Crude approximation

- Start with very crude approximation:
  - Just insert edges from function-call to procedure start
  - And from return of procedure to target-node of function call

Crude approximation

- Start with very crude approximation:
  - Just insert edges from function-call to procedure start
  - And from return of procedure to target-node of function call
- I.e, for $(u, f(), v)$, generate constraints

$$D[{v_0}^f] \sqsupseteq \mathrm{enter}_f^\# D[u]$$

$$D[v] \sqsupseteq \mathrm{combine}_f^\#(D[u], D[v_e^f]) \qquad v_e^f \in V_{\mathrm{end}}^f$$

Crude approximation

- Start with very crude approximation:
  - Just insert edges from function-call to procedure start
  - And from return of procedure to target-node of function call
- I.e, for $(u, f(), v)$, generate constraints

$$D[{v_0}^f] \sqsupseteq \mathrm{enter}_f^\# D[u]$$

$$D[v] \sqsupseteq \mathrm{combine}_f^\#(D[u], D[v_e^f]) \qquad v_e^f \in V_{\mathrm{end}}^f$$

- Clearly covers all possible paths

Crude approximation

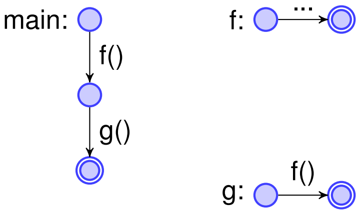
- Start with very crude approximation:
  - Just insert edges from function-call to procedure start
  - And from return of procedure to target-node of function call
- I.e, for $(u, f(), v)$, generate constraints

$$
\begin{aligned}
D[{v_0}^f] &\sqsupseteq \mathrm{enter}_f^\# D[u] \\
D[v] &\sqsupseteq \mathrm{combine}_f^\#(D[u], D[v_e^f]) && v_e^f \in V_{\mathrm{end}}^f
\end{aligned}
$$

- Clearly covers all possible paths
- But also infeasible ones

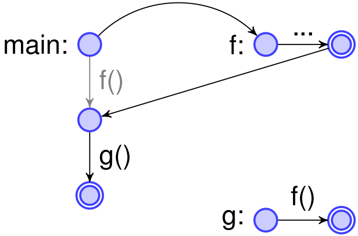
Crude approximation, example

```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



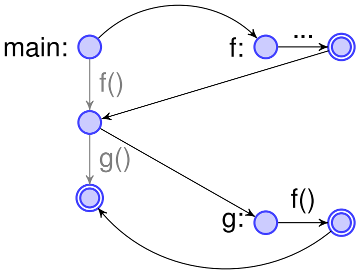
Crude approximation, example

```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



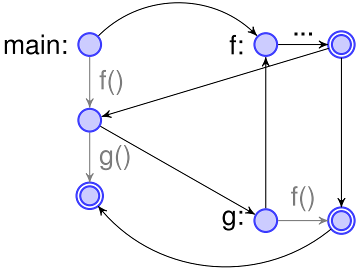
Crude approximation, example

```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



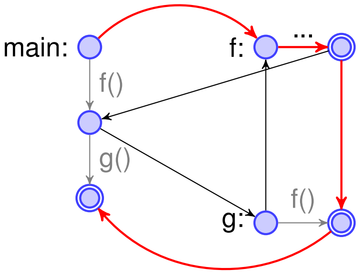
Crude approximation, example

```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



Crude approximation, example

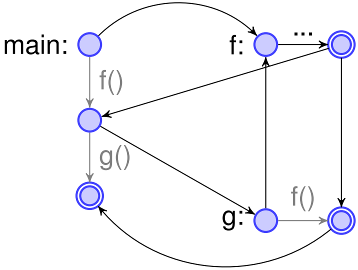
```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



Infeasible paths

Crude approximation, example

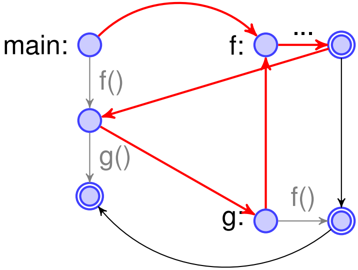
```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



Infeasible paths

Crude approximation, example

```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



Infeasible paths

Call strings

- Idea: Call string contains sequence of up to k program points

Call strings

- Idea: Call string contains sequence of up to k program points
- These are the topmost k return addresses on the stack

Call strings

- Idea: Call string contains sequence of up to *k* program points
- These are the topmost *k* return addresses on the stack
- Analyze procedures for every (feasible) call-string

Call strings

- Idea: Call string contains sequence of up to k program points
- These are the topmost k return addresses on the stack
- Analyze procedures for every (feasible) call-string
- Only create edges that match call-string

Call strings

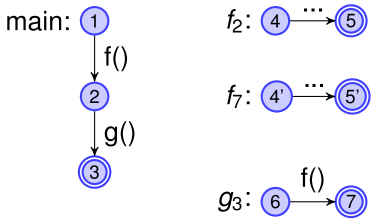
$$
\begin{array}{rll}
D[{v_0}^f, (v\omega)|_k] & \sqsupseteq \text{enter}^\#(D[u,\omega]) & (u, f(), v) \in E \\
D[v,\omega] & \sqsupseteq \text{combine}^\#(D[u,\omega], D[f, (v\omega)|_k]) & (u, f(), v) \in E \\
D[f,\omega] & \sqsupseteq D[v_e,\omega] & v_e \in V^f_{\text{end}}
\end{array}
$$

$$
\begin{array}{rll}
D[{v_0}^{\text{main}}, \varepsilon] & \sqsupseteq d_0 & \\
D[v,\omega] & \sqsupseteq [\![k]\!]^\# D[u,\omega] & k = (u, a, v) \in E
\end{array}
$$

- where $((\cdot)|_k)$ limits string size to k, cutting off nodes from the end

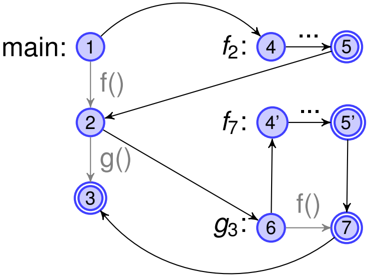
Example

```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



Example

```
f () {...}
g () { f() }

main () {
  f ();
  g ()
}
```



Discussion

- Analysis terminates if D has finite height

Discussion

- Analysis terminates if D has finite height
- Call strings with $k = 0$ matches crude approximation

Discussion

- Analysis terminates if D has finite height
- Call strings with $k = 0$ matches crude approximation
- Can increase precision by eliminating (some) infeasible paths

Discussion

- Analysis terminates if D has finite height
- Call strings with $k = 0$ matches crude approximation
- Can increase precision by eliminating (some) infeasible paths
- Cost increases exponentially with size of k

Discussion

- Analysis terminates if D has finite height
- Call strings with $k = 0$ matches crude approximation
- Can increase precision by eliminating (some) infeasible paths
- Cost increases exponentially with size of k
- In practice $k = 0$ or $k = 1$

Discussion

- Analysis terminates if D has finite height
- Call strings with $k = 0$ matches crude approximation
- Can increase precision by eliminating (some) infeasible paths
- Cost increases exponentially with size of k
- In practice $k = 0$ or $k = 1$
- Correctness proof: Simulation wrt. stack-based semantics

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based
- Analysis:

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based
- Analysis:
  - Functional: Compute same-level effects

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based
- Analysis:
  - Functional: Compute same-level effects
    - Requires efficient representation of effects

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based
- Analysis:
  - Functional: Compute same-level effects
    - Requires efficient representation of effects
  - Evaluation on demand: Same-level effects for finite number of arguments

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based
- Analysis:
  - Functional: Compute same-level effects
    - Requires efficient representation of effects
  - Evaluation on demand: Same-level effects for finite number of arguments
    - Requires finite/small number of abstract arguments for each function

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based
- Analysis:
    - Functional: Compute same-level effects
        - Requires efficient representation of effects
    - Evaluation on demand: Same-level effects for finite number of arguments
        - Requires finite/small number of abstract arguments for each function
    - Call-Strings: Limit stack-depth, add extra (stack-insensitive) paths above depth limit

Summary: Interprocedural Analysis

- Semantics: Stack-based, path-based
- Analysis:
  - Functional: Compute same-level effects
    - Requires efficient representation of effects
  - Evaluation on demand: Same-level effects for finite number of arguments
    - Requires finite/small number of abstract arguments for each function
  - Call-Strings: Limit stack-depth, add extra (stack-insensitive) paths above depth limit
    - Adds extra imprecision, exponentially cost in depth-limit

Table of Contents

Analysis of Parallel Programs

- Concurrency gets more important nowadays

Analysis of Parallel Programs

- Concurrency gets more important nowadays
- Admits new classes of bugs

Analysis of Parallel Programs

- Concurrency gets more important nowadays
- Admits new classes of bugs
  - E.g, data races

Analysis of Parallel Programs

- Concurrency gets more important nowadays
- Admits new classes of bugs
  - E.g, data races
- These are hard to find/ hard to reproduce

Analysis of Parallel Programs

- Concurrency gets more important nowadays
- Admits new classes of bugs
    - E.g, data races
- These are hard to find/ hard to reproduce
- Can program analysis help?

Data races

- Concurrent accesses to global data, one is a write

```
int g = 0;
t1 () {
  g = g + 1
}
main () {
  fork t1;
  g = g + 1
  join;
  print g
}
```

Data races

- Concurrent accesses to global data, one is a write

```
int g = 0;
t1 () {
  g = g + 1
}
main () {
  fork t1;
  g = g + 1
  join;
  print g
}
```

- What will the program print?

Data races

- Concurrent accesses to global data, one is a write

```
int g = 0;
t1 () {
  g = g + 1
}
main () {
  fork t1;
  g = g + 1
  join;
  print g
}
```

- What will the program print?
  - Assuming sequential consistency?

Data races

- Concurrent accesses to global data, one is a write

```
int g = 0;
t1 () {
  g = g + 1
}
main () {
  fork t1;
  g = g + 1
  join;
  print g
}
```

- What will the program print?
  - Assuming sequential consistency?
- Answer: In most cases: 2

Data races

- Concurrent accesses to global data, one is a write

```
int g = 0;
t1 () {
  g = g + 1
}
main () {
  fork t1;
  g = g + 1
  join;
  print g
}
```

- What will the program print?
  - Assuming sequential consistency?
- Answer: In most cases: 2
- But in very rare cases: 1

Data races

- Concurrent accesses to global data, one is a write

```
int g = 0;
t1 () {
  g = g + 1
}
main () {
  fork t1;
  g = g + 1
  join;
  print g
}
```

- What will the program print?
  - Assuming sequential consistency?
- Answer: In most cases: 2
- But in very rare cases: 1
- Depends on machine, other programs, OS, start time, ...

Locks

- Threads can acquire/release locks

```
int g = 0; lock lg;
t1 () {
  acquire(lg); g = g + 1; release(lg);
}
main () {
  fork t1;
  acquire(lg); g = g + 1; release(lg);
  join;
  print g
}
```

Locks

- Threads can acquire/release locks
- Each lock can only be acquired by one thread at the same time

```
int g = 0; lock lg;
t1 () {
  acquire(lg); g = g + 1; release(lg);
}
main () {
  fork t1;
  acquire(lg); g = g + 1; release(lg);
  join;
  print g
}
```

Locks

- Threads can acquire/release locks
- Each lock can only be acquired by one thread at the same time
- Other threads that want to acquire the lock have to wait

```
int g = 0; lock lg;
t1 () {
  acquire(lg); g = g + 1; release(lg);
}
main () {
  fork t1;
  acquire(lg); g = g + 1; release(lg);
  join;
  print g
}
```

Locks

- Threads can acquire/release locks
- Each lock can only be acquired by one thread at the same time
- Other threads that want to acquire the lock have to wait
- Used to prevent data races

```
int g = 0; lock lg;
t1 () {
  acquire(lg); g = g + 1; release(lg);
}
main () {
  fork t1;
  acquire(lg); g = g + 1; release(lg);
  join;
  print g
}
```

Demo: Goblint data race analyzer

- Program with data race
- Try to show bad reproducibility + dependence on machine load, etc.
- Show goblint-analyzer to find the race

`http://goblint.in.tum.de`

Abstract semantics with locks

- We will regard an abstract semantics with locks

Abstract semantics with locks

- We will regard an abstract semantics with locks
- I.e., it contains no state beyond the current program points and status of locks

Abstract semantics with locks

- We will regard an abstract semantics with locks
- I.e., it contains no state beyond the current program points and status of locks
- Concrete program mapped to this semantics

Abstract semantics with locks

- We will regard an abstract semantics with locks
- I.e., it contains no state beyond the current program points and status of locks
- Concrete program mapped to this semantics
  - E.g., pointer analysis to identify locks

Abstract semantics with locks

- We will regard an abstract semantics with locks
- I.e., it contains no state beyond the current program points and status of locks
- Concrete program mapped to this semantics
  - E.g., pointer analysis to identify locks
- Has more possible executions than concrete program

Abstract semantics with locks

- We will regard an abstract semantics with locks
- I.e., it contains no state beyond the current program points and status of locks
- Concrete program mapped to this semantics
    - E.g., pointer analysis to identify locks
- Has more possible executions than concrete program
- Analysis results are safe

Abstract semantics with locks

- We will regard an abstract semantics with locks
- I.e., it contains no state beyond the current program points and status of locks
- Concrete program mapped to this semantics
  - E.g., pointer analysis to identify locks
- Has more possible executions than concrete program
- Analysis results are safe
  - If we find no datarace, there is none

Abstract semantics with locks

- We will regard an abstract semantics with locks
- I.e., it contains no state beyond the current program points and status of locks
- Concrete program mapped to this semantics
  - E.g., pointer analysis to identify locks
- Has more possible executions than concrete program
- Analysis results are safe
  - If we find no datarace, there is none
  - But there may be false positives

Parallel flowgraphs with fork

- Add $\mathrm{fork}(v)$ edge label, that forks new thread starting at v

Parallel flowgraphs with fork

- Add $\mathrm{fork}(v)$ edge label, that forks new thread starting at v
  - For now, we ignore joins!

Parallel flowgraphs with fork

- Add $\mathrm{fork}(v)$ edge label, that forks new thread starting at v
  - For now, we ignore joins!
- Abstract semantics: State is multiset of nodes.

Parallel flowgraphs with fork

- Add $\mathrm{fork}(v)$ edge label, that forks new thread starting at v
  - For now, we ignore joins!
- Abstract semantics: State is multiset of nodes.
  - Initial state: $\{v_0\}$

$$
\begin{aligned}
(\{u\} \mathbin{\dot\cup} s) &\to (\{v\} \mathbin{\dot\cup} s) && (u, a, v) \in E \\
(\{u\} \mathbin{\dot\cup} s) &\to (\{v, w\} \mathbin{\dot\cup} s) && (u, \mathrm{fork}(w), v) \in E
\end{aligned}
$$

Parallel flowgraphs with fork and locks

- Additionally: Finite set of locks $\mathbb{L}$, actions $\mathrm{acq}(l)$ and $\mathrm{rel}(l)$

Parallel flowgraphs with fork and locks

- Additionally: Finite set of locks $\mathbb{L}$, actions $\mathrm{acq}(l)$ and $\mathrm{rel}(l)$
- State: Each thread together with its acquired locks

Parallel flowgraphs with fork and locks

- Additionally: Finite set of locks $\mathbb{L}$, actions $\mathrm{acq}(l)$ and $\mathrm{rel}(l)$
- State: Each thread together with its acquired locks
- Initial state: $\{(v_0, \emptyset)\}$

$$
\begin{aligned}
(\{(u,L)\} \mathbin{\dot\cup} s) &\to (\{(v,L)\} \mathbin{\dot\cup} s) & (u,a,v) &\in E \\
(\{(u,L)\} \mathbin{\dot\cup} s) &\to (\{(v,L),(w,\emptyset)\} \mathbin{\dot\cup} s) & (u,\mathrm{fork}(w),v) &\in E \\
(\{(u,L)\} \mathbin{\dot\cup} s) &\to (\{(v,L \cup \{l\})\} \mathbin{\dot\cup} s) & (u,\mathrm{acq}(l),v) &\in E \text{ and } l \notin s|_2 \\
(\{(u,L)\} \mathbin{\dot\cup} s) &\to (\{(v,L \setminus \{l\}\} \mathbin{\dot\cup} s) & (u,\mathrm{rel}(l),v) &\in E
\end{aligned}
$$

Parallel flowgraphs with fork and locks

- Additionally: Finite set of locks $\mathbb{L}$, actions $\mathrm{acq}(l)$ and $\mathrm{rel}(l)$
- State: Each thread together with its acquired locks
- Initial state: $\{(v_0, \emptyset)\}$

$$
\begin{array}{rr}
(\{(u, L)\} \,\dot{\cup}\, s) \to (\{(v, L)\} \,\dot{\cup}\, s) & (u, a, v) \in E \\
(\{(u, L)\} \,\dot{\cup}\, s) \to (\{(v, L), (w, \emptyset)\} \,\dot{\cup}\, s) & (u, \mathrm{fork}(w), v) \in E \\
(\{(u, L)\} \,\dot{\cup}\, s) \to (\{(v, L \cup \{l\})\} \,\dot{\cup}\, s) & (u, \mathrm{acq}(l), v) \in E \text{ and } l \notin s|_2 \\
(\{(u, L)\} \,\dot{\cup}\, s) \to (\{(v, L \setminus \{l\}\} \,\dot{\cup}\, s) & (u, \mathrm{rel}(l), v) \in E
\end{array}
$$

- Note: We assume that a thread only releases locks that it possesses.

Parallel flowgraphs with fork and locks

- Additionally: Finite set of locks $\mathbb{L}$, actions $\mathrm{acq}(l)$ and $\mathrm{rel}(l)$
- State: Each thread together with its acquired locks
- Initial state: $\{(v_0, \emptyset)\}$

$$
\begin{array}{rr}
(\{(u,L)\} \mathbin{\dot\cup} s) \to (\{(v,L)\} \mathbin{\dot\cup} s) & (u,a,v) \in E \\
(\{(u,L)\} \mathbin{\dot\cup} s) \to (\{(v,L),(w,\emptyset)\} \mathbin{\dot\cup} s) & (u,\mathrm{fork}(w),v) \in E \\
(\{(u,L)\} \mathbin{\dot\cup} s) \to (\{(v,L \cup \{l\})\} \mathbin{\dot\cup} s) & (u,\mathrm{acq}(l),v) \in E \text{ and } l \notin s|_2 \\
(\{(u,L)\} \mathbin{\dot\cup} s) \to (\{(v,L \setminus \{l\}\} \mathbin{\dot\cup} s) & (u,\mathrm{rel}(l),v) \in E
\end{array}
$$

- Note: We assume that a thread only releases locks that it possesses.
- We assume that a thread does not acquire a lock it already possesses.

Parallel flowgraphs with fork and locks

- Additionally: Finite set of locks $\mathbb{L}$, actions $\mathrm{acq}(l)$ and $\mathrm{rel}(l)$
- State: Each thread together with its acquired locks
- Initial state: $\{(v_0, \emptyset)\}$

$$
\begin{array}{rr}
(\{(u,L)\} \,\dot{\cup}\, s) \to (\{(v,L)\} \,\dot{\cup}\, s) & (u,a,v) \in E \\
(\{(u,L)\} \,\dot{\cup}\, s) \to (\{(v,L),(w,\emptyset)\} \,\dot{\cup}\, s) & (u,\mathrm{fork}(w),v) \in E \\
(\{(u,L)\} \,\dot{\cup}\, s) \to (\{(v,L \cup \{l\})\} \,\dot{\cup}\, s) & (u,\mathrm{acq}(l),v) \in E \text{ and } l \notin s|_2 \\
(\{(u,L)\} \,\dot{\cup}\, s) \to (\{(v,L \setminus \{l\}\} \,\dot{\cup}\, s) & (u,\mathrm{rel}(l),v) \in E
\end{array}
$$

- Note: We assume that a thread only releases locks that it possesses.
- We assume that a thread does not acquire a lock it already possesses.
- Invariant: For each reachable state, the thread's lock-sets are disjoint

$$\{(v_0,\emptyset)\} \to^* \{(u_1,L_1),(u_2,L_2)\} \,\dot{\cup}\, s \implies L_1 \cap L_2 = \emptyset$$

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel
- Lock-sets

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel
- Lock-sets
  - Sets of locks that must be allocated at program point

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel
- Lock-sets
  - Sets of locks that must be allocated at program point
  - Used to make MHP more precise

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel
- Lock-sets
  - Sets of locks that must be allocated at program point
  - Used to make MHP more precise
    - $\mathrm{MHP}(u, v)$ only if u and v have disjoint lock sets

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel
- Lock-sets
  - Sets of locks that must be allocated at program point
  - Used to make MHP more precise
    - $\mathrm{MHP}(u, v)$ only if u and v have disjoint lock sets
- Data-Races

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel
- Lock-sets
  - Sets of locks that must be allocated at program point
  - Used to make MHP more precise
    - $\mathrm{MHP}(u, v)$ only if u and v have disjoint lock sets
- Data-Races
  - Identify conflicting program points, with outgoing actions that read/write the same global variable

Analysis Plan

- Lock-insensitive may-happen in parallel (MHP)
  - Sets of program points that may be executed in parallel
- Lock-sets
  - Sets of locks that must be allocated at program point
  - Used to make MHP more precise
    - $\mathrm{MHP}(u, v)$ only if u and v have disjoint lock sets
- Data-Races
  - Identify conflicting program points, with outgoing actions that read/write the same global variable
  - Check whether they may happen in parallel

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$$
\begin{aligned}
R[u] &\supseteq \{u\} && \text{if } u \text{ interesting} && \text{(R.node)}\\
R[u] &\supseteq R[v] && \text{if } (u, \_, v) \in E && \text{(R.edge)}\\
R[u] &\supseteq R[w] && \text{if } (u, \mathrm{fork}(w), \_) \in E && \text{(R.trans)}
\end{aligned}
$$

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$R[u] \supseteq \{u\}$	if u interesting	(R.node)
$R[u] \supseteq R[v]$	if $(u, \_, v) \in E$	(R.edge)
$R[u] \supseteq R[w]$	if $(u, \mathrm{fork}(w), \_) \in E$	(R.trans)

$\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u]$	if $(u, \_, v) \in E$	(MHP.edge)
$\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.trans)
$\mathrm{MHP}[v] \supseteq R[w]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.fork1)
$\mathrm{MHP}[w] \supseteq R[v]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.fork2)

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$$
\begin{array}{lll}
R[u] \supseteq \{u\} & \text{if } u \text{ interesting} & \text{(R.node)} \\
R[u] \supseteq R[v] & \text{if } (u, \_, v) \in E & \text{(R.edge)} \\
R[u] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), \_) \in E & \text{(R.trans)}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u] & \text{if } (u, \_, v) \in E & \text{(MHP.edge)} \\
\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.trans)} \\
\mathrm{MHP}[v] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork1)} \\
\mathrm{MHP}[w] \supseteq R[v] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork2)}
\end{array}
$$

(*R.node*) Interesting node reachable from itself

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$R[u] \supseteq \{u\}$	if u interesting	(R.node)
$R[u] \supseteq R[v]$	if $(u, \_, v) \in E$	(R.edge)
$R[u] \supseteq R[w]$	if $(u, \mathrm{fork}(w), \_) \in E$	(R.trans)

$\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u]$	if $(u, \_, v) \in E$	(MHP.edge)
$\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.trans)
$\mathrm{MHP}[v] \supseteq R[w]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.fork1)
$\mathrm{MHP}[w] \supseteq R[v]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.fork2)

(*R.edge*) Propagate reachability over edge

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$$R[u] \supseteq \{u\} \quad \text{if } u \text{ interesting} \quad \text{(R.node)}$$
$$R[u] \supseteq R[v] \quad \text{if } (u, \_, v) \in E \quad \text{(R.edge)}$$
$$R[u] \supseteq R[w] \quad \text{if } (u, \mathrm{fork}(w), \_) \in E \quad \text{(R.trans)}$$

$$\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u] \quad \text{if } (u, \_, v) \in E \quad \text{(MHP.edge)}$$
$$\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u] \quad \text{if } (u, \mathrm{fork}(w), v) \in E \quad \text{(MHP.trans)}$$
$$\mathrm{MHP}[v] \supseteq R[w] \quad \text{if } (u, \mathrm{fork}(w), v) \in E \quad \text{(MHP.fork1)}$$
$$\mathrm{MHP}[w] \supseteq R[v] \quad \text{if } (u, \mathrm{fork}(w), v) \in E \quad \text{(MHP.fork2)}$$

(*R.trans*) Propagate reachability over fork

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$$
\begin{array}{lll}
R[u] \supseteq \{u\} & \text{if } u \text{ interesting} & \text{(R.node)} \\
R[u] \supseteq R[v] & \text{if } (u, \_, v) \in E & \text{(R.edge)} \\
R[u] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), \_) \in E & \text{(R.trans)}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u] & \text{if } (u, \_, v) \in E & \text{(MHP.edge)} \\
\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.trans)} \\
\mathrm{MHP}[v] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork1)} \\
\mathrm{MHP}[w] \supseteq R[v] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork2)}
\end{array}
$$

(*MHP.edge*) If this edge executed, other threads still at same positions

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$R[u] \supseteq \{u\}$	if u interesting	(R.node)
$R[u] \supseteq R[v]$	if $(u, \_, v) \in E$	(R.edge)
$R[u] \supseteq R[w]$	if $(u, \mathrm{fork}(w), \_) \in E$	(R.trans)

$\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u]$	if $(u, \_, v) \in E$	(MHP.edge)
$\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.trans)
$\mathrm{MHP}[v] \supseteq R[w]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.fork1)
$\mathrm{MHP}[w] \supseteq R[v]$	if $(u, \mathrm{fork}(w), v) \in E$	(MHP.fork2)

(*MHP.trans*) Start node of forked thread parallel to other threads

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$$
\begin{array}{lll}
R[u] \supseteq \{u\} & \text{if } u \text{ interesting} & \text{(R.node)} \\
R[u] \supseteq R[v] & \text{if } (u, \_, v) \in E & \text{(R.edge)} \\
R[u] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), \_) \in E & \text{(R.trans)}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u] & \text{if } (u, \_, v) \in E & \text{(MHP.edge)} \\
\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.trans)} \\
\mathrm{MHP}[v] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork1)} \\
\mathrm{MHP}[w] \supseteq R[v] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork2)}
\end{array}
$$

($MHP.fork1$) Forking thread parallel to everything that may be reached from forked thread

Lock-insensitive MHP

- Put up constraint system, $R[u]$: Set of (interesting) nodes reachable from u
  - Reachable also over forks

$$
\begin{array}{lll}
R[u] \supseteq \{u\} & \text{if } u \text{ interesting} & \text{(R.node)} \\
R[u] \supseteq R[v] & \text{if } (u, \_, v) \in E & \text{(R.edge)} \\
R[u] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), \_) \in E & \text{(R.trans)}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{MHP}[v] \supseteq \mathrm{MHP}[u] & \text{if } (u, \_, v) \in E & \text{(MHP.edge)} \\
\mathrm{MHP}[w] \supseteq \mathrm{MHP}[u] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.trans)} \\
\mathrm{MHP}[v] \supseteq R[w] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork1)} \\
\mathrm{MHP}[w] \supseteq R[v] & \text{if } (u, \mathrm{fork}(w), v) \in E & \text{(MHP.fork2)}
\end{array}
$$

($MHP.fork2$) Forked thread parallel to everything that may be reached from forking thread

Correctness

- For interesting nodes u and v (also u=v), we have:

$$\exists s.\ \{v_0\} \to^* \{u, v\} \,\dot\cup\, s \implies u \in \mathrm{MHP}[v]$$

Correctness

- For interesting nodes u and v (also u=v), we have:

$$\exists s.\ \{v_0\} \to^* \{u, v\} \,\dot{\cup}\, s \implies u \in \mathrm{MHP}[v]$$

- Proof sketch

Correctness

- For interesting nodes u and v (also u=v), we have:

$$\exists s.\ \{v_0\} \to^* \{u, v\} \,\dot{\cup}\, s \implies u \in \mathrm{MHP}[v]$$

- Proof sketch
  - Auxiliary: $\{u\} \to^* \{v\} \,\dot{\cup}\, s \implies v \in R[u]$

Correctness

- For interesting nodes u and v (also u=v), we have:

$$\exists s.\ \{v_0\} \to^* \{u, v\} \mathbin{\dot\cup} s \implies u \in \mathrm{MHP}[v]$$

- Proof sketch
  - Auxiliary: $\{u\} \to^* \{v\} \mathbin{\dot\cup} s \implies v \in R[u]$
  - Find the crucial fork, where u is reached from, wlog, the forked thread, and v is reached from the forking thread

Correctness

- For interesting nodes u and v (also u=v), we have:

$$\exists s.\ \{v_0\} \to^* \{u, v\} \dot{\cup} s \implies u \in \mathrm{MHP}[v]$$

- Proof sketch
  - Auxiliary: $\{u\} \to^* \{v\} \dot{\cup} s \implies v \in R[u]$
  - Find the crucial fork, where u is reached from, wlog, the forked thread, and v is reached from the forking thread
    - $\{v_0\} \to^* \{a\} \dot{\cup} \ldots$, and $(a, \mathrm{fork}(c), b) \in E$, and $\{b\} \to^* \{u\} \dot{\cup} \ldots$, and $\{c\} \to^* \{v\} \dot{\cup} \ldots$

Lock-set analysis

- Forward, must analysis (standard)

$$
\begin{array}{rcll}
LS[v_0] & \subseteq & \emptyset & \\
LS[w] & \subseteq & \emptyset & (u, \mathrm{fork}(w), v) \in E \\
LS[v] & \subseteq & LS[u] & (u, a, v) \in E,\ a \text{ no lock-action} \\
LS[v] & \subseteq & LS[u] \cup \{l\} & (u, \mathrm{acq}(l), v) \in E \\
LS[v] & \subseteq & LS[u] \setminus \{l\} & (u, \mathrm{rel}(l), v) \in E
\end{array}
$$

Lock-set analysis

- Forward, must analysis (standard)

$$\begin{array}{ll}
LS[v_0] \subseteq \emptyset & \\
LS[w] \subseteq \emptyset & (u, \mathrm{fork}(w), v) \in E \\
LS[v] \subseteq LS[u] & (u, a, v) \in E,\ a \text{ no lock-action} \\
LS[v] \subseteq LS[u] \cup \{l\} & (u, \mathrm{acq}(l), v) \in E \\
LS[v] \subseteq LS[u] \setminus \{l\} & (u, \mathrm{rel}(l), v) \in E
\end{array}$$

- Correctness:

$$l \in LS[u] \implies (\forall s.\ \{(v_0, \emptyset)\} \to^* \{(u, L)\} \mathbin{\dot\cup} s \implies l \in L)$$

Data-Race analysis

- Interesting nodes:

Data-Race analysis

- Interesting nodes:
  - Nodes with actions that read or write global variables

Data-Race analysis

- Interesting nodes:
  - Nodes with actions that read or write global variables
- For each pair (u, v) of conflicting nodes, check
  $u \in \mathrm{MHP}[v] \implies LS[u] \cap LS[v] \neq \emptyset$

Data-Race analysis

- Interesting nodes:
  - Nodes with actions that read or write global variables
- For each pair (u, v) of conflicting nodes, check
  $u \in \mathrm{MHP}[v] \implies LS[u] \cap LS[v] \neq \emptyset$
- If satisfied, report „definitely no data race”

Data-Race analysis

- Interesting nodes:
  - Nodes with actions that read or write global variables
- For each pair (u, v) of conflicting nodes, check
  $u \in \mathrm{MHP}[v] \implies LS[u] \cap LS[v] \neq \emptyset$
- If satisfied, report „definitely no data race”
- Otherwise, report possible data race

Example

```
     int g = 0; lock lg;
     t1 () {
1:     acquire(lg);          R: 2      MHP: {7,11} L: {}
2:     g = g + 1;            R: 2      MHP: {7,11} L: {lg}
3:     release(lg);          R: {}     MHP: {7,11} L: {lg}
4:   }                                 MHP: {7,11} L: {}
     main () {
5:     fork t1;              R: 2,7,11 MHP: {}     L: {}
6:     acquire(lg);          R: 7,11   MHP: {2}    L: {}
7:     g = g + 1;            R: 7,11   MHP: {2}    L: {lg}
8:     release(lg);          R: 11     MHP: {2}    L: {lg}
9:     join;                 R: 11     MHP: {2}    L: {}
10:    acquire(lg);          R: 11     MHP: {2}    L: {}
11:    print g               R: 11     MHP: {2}    L: {lg}
12:    release(lg);          R: {}     MHP: {2}    L: {lg}
13: }                        R: {}     MHP: {2}    L: {}
```

- Check lock-sets for 2/7 and 2/11

Example

```
    int g = 0; lock lg;
    t1 () {
1:    acquire(lg);      R: 2       MHP: {7,11} L: {}
2:    g = g + 1;        R: 2       MHP: {7,11} L: {lg}
3:    release(lg);      R: {}      MHP: {7,11} L: {lg}
4:  }                              MHP: {7,11} L: {}
    main () {
5:    fork t1;          R: 2,7,11  MHP: {}     L: {}
6:    acquire(lg);      R: 7,11    MHP: {2}    L: {}
7:    g = g + 1;        R: 7,11    MHP: {2}    L: {lg}
8:    release(lg);      R: 11      MHP: {2}    L: {lg}
9:    join;             R: 11      MHP: {2}    L: {}
10:   acquire(lg);      R: 11      MHP: {2}    L: {}
11:   print g           R: 11      MHP: {2}    L: {lg}
12:   release(lg);      R: {}      MHP: {2}    L: {lg}
13: }                   R: {}      MHP: {2}    L: {}
```

- Check lock-sets for 2/7 and 2/11
- Lock *lg* contained in all of them

Example

```
     int g = 0; lock lg;
     t1 () {
1:     acquire(lg);         R: 2        MHP: {7,11} L: {}
2:     g = g + 1;           R: 2        MHP: {7,11} L: {lg}
3:     release(lg);         R: {}       MHP: {7,11} L: {lg}
4:   }                                  MHP: {7,11} L: {}
     main () {
5:     fork t1;             R: 2,7,11 MHP: {}      L: {}
6:     acquire(lg);         R: 7,11     MHP: {2}    L: {}
7:     g = g + 1;           R: 7,11     MHP: {2}    L: {lg}
8:     release(lg);         R: 11       MHP: {2}    L: {lg}
9:     join;                R: 11       MHP: {2}    L: {}
10:    acquire(lg);         R: 11       MHP: {2}    L: {}
11:    print g              R: 11       MHP: {2}    L: {lg}
12:    release(lg);         R: {}       MHP: {2}    L: {lg}
13: }                       R: {}       MHP: {2}    L: {}
```

- Check lock-sets for 2/7 and 2/11
- Lock *lg* contained in all of them
- Program is safe!

Discussion

- Simple (and relatively cheap) analysis

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
    - Not handling joins

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
    - Not handling joins
    - Ignoring data completely

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
  - Not handling joins
  - Ignoring data completely
  - Not handling interaction of locks and control flow

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
  - Not handling joins
  - Ignoring data completely
  - Not handling interaction of locks and control flow
    - Fork inside lock

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
  - Not handling joins
  - Ignoring data completely
  - Not handling interaction of locks and control flow
    - Fork inside lock
    - Deadlocks

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
    - Not handling joins
    - Ignoring data completely
    - Not handling interaction of locks and control flow
        - Fork inside lock
        - Deadlocks
- Goblint:

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
    - Not handling joins
    - Ignoring data completely
    - Not handling interaction of locks and control flow
        - Fork inside lock
        - Deadlocks
- Goblint:
    - Interprocedural

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
  - Not handling joins
  - Ignoring data completely
  - Not handling interaction of locks and control flow
    - Fork inside lock
    - Deadlocks
- Goblint:
  - Interprocedural
  - Pointer-analysis

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
  - Not handling joins
  - Ignoring data completely
  - Not handling interaction of locks and control flow
    - Fork inside lock
    - Deadlocks
- Goblint:
  - Interprocedural
  - Pointer-analysis
  - Constant propagation

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
  - Not handling joins
  - Ignoring data completely
  - Not handling interaction of locks and control flow
    - Fork inside lock
    - Deadlocks
- Goblint:
  - Interprocedural
  - Pointer-analysis
  - Constant propagation
  - Equality/inequality of indexes

Discussion

- Simple (and relatively cheap) analysis
- Can prove programs data-race free
- But may return false positives, due to:
  - Not handling joins
  - Ignoring data completely
  - Not handling interaction of locks and control flow
    - Fork inside lock
    - Deadlocks
- Goblint:
  - Interprocedural
  - Pointer-analysis
  - Constant propagation
  - Equality/inequality of indexes
  - ...

Discussion

- Freedom of data races often not enough

```
int x[N];
void norm() {
  lock l; n = length(x); unlock l;
  lock l; x = 1/n * x; unlock l;
}
```

Discussion

- Freedom of data races often not enough

```
int x[N];
void norm() {
  lock l; n = length(x); unlock l;
  lock l; x = 1/n * x; unlock l;
}
```

  - Thread-safe?

Discussion

- Freedom of data races often not enough

```
int x[N];
void norm() {
  lock l; n = length(x); unlock l;
  lock l; x = 1/n * x; unlock l;
}
```

  - Thread-safe? No!

Discussion

- Freedom of data races often not enough

```
int x[N];
void norm() {
  lock l; n = length(x); unlock l;
  lock l; x = 1/n * x; unlock l;
}
```

  - Thread-safe? No!

$\Longrightarrow$ Transactionality

Discussion

- Freedom of data races often not enough

```
int x[N];
void norm() {
  lock l; n = length(x); unlock l;
  lock l; x = 1/n * x; unlock l;
}
```

  - Thread-safe? No!

$\Longrightarrow$ Transactionality

- Advanced locking patterns

Discussion

- Freedom of data races often not enough

```
int x[N];
void norm() {
  lock l; n = length(x); unlock l;
  lock l; x = 1/n * x; unlock l;
}
```

  - Thread-safe? No!

$\Longrightarrow$ Transactionality

- Advanced locking patterns
  - E.g., lock chains:

```
lock 1; lock 2; unlock 1; lock 3; unlock 2 ...
```

Discussion

- Freedom of data races often not enough

```
int x[N];
void norm() {
  lock l; n = length(x); unlock l;
  lock l; x = 1/n * x; unlock l;
}
```

  - Thread-safe? No!

$\Longrightarrow$ Transactionality

- Advanced locking patterns
  - E.g., lock chains:

    ```
    lock 1; lock 2; unlock 1; lock 3; unlock 2 ...
    ```

  - Two lock-chains executed simultaneously will never overtake

Last Lecture

- Analysis of parallel programs
  - Intraprocedural with thread creation
  - May-happen in parallel + lockset analysis = datarace analysis
- Caveats
  - Need to abstract program into model with fixed locks
    - Problematic if locks are addressed via pointers/arrays
  - Datarace freedom may no be enough
    - Transactions
    - Advanced locking patterns like lockchains

Table of Contents

Table of Contents

Motivating Example

```
for (i=l;i<r;i=i+h) {
  a=a0 + b*i
  M[a] = ...
}
```

- Initialize array in range: $[l, r[$, every hth element

Motivating Example

```
for (i=l;i<r;i=i+h) {
  a=a0 + b*i
  M[a] = ...
}
```

- Initialize array in range: $[l, r[$, every hth element
- Element size of array: b

Motivating Example

```
for (i=l;i<r;i=i+h) {
  a=a0 + b*i
  M[a] = ...
}
```

- Initialize array in range: $[l, r[$, every hth element
- Element size of array: b
- Loop requires $r - l$ multiplications

Motivating Example

```
for (i=l;i<r;i=i+h) {
  a=a0 + b*i
  M[a] = ...
}
```

- Initialize array in range: $[l, r[$, every hth element
- Element size of array: b
- Loop requires $r - l$ multiplications
- Multiplications are expensive, addition much cheaper

Motivating Example

```
for (i=l;i<r;i=i+h) {
  a=a0 + b*i
  M[a] = ...
}
```

- Initialize array in range: $[l, r[$, every hth element
- Element size of array: b
- Loop requires $r - l$ multiplications
- Multiplications are expensive, addition much cheaper
- Observation: From one iteration of the loop to the next:
  - Difference between as is constant: $(a_0 + b(i + h)) - (a_0 + bi) = bh$

Optimization

- First, loop inversion

```
i=l;
if (i<r) {
  do {
    a=a0 + b*i
    M[a] = ...
    i=i+h
  } while (i<r)
}
```

Optimization

- First, loop inversion
- Second, pre-compute difference and replace computation of *a*
  - No multiplication left in loop

```
i=l;
if (i<r) {
  delta = b*h
  a=a0 + b*i
  do {
    M[a] = ...
    i=i+h
    a=a+delta
  } while (i<r)
}
```

Optimization

- First, loop inversion
- Second, pre-compute difference and replace computation of *a*
  - No multiplication left in loop
- If
  - *i* not used elsewhere in the loop, and
  - *i* dead after loop
  - *b* not zero

```
i=l;
if (i<r) {
  delta = b*h
  a=a0 + b*i
  do {
    M[a] = ...
    i=i+h
    a=a+delta
  } while (i<r)
}
```

Optimization

- First, loop inversion
- Second, pre-compute difference and replace computation of *a*
  - No multiplication left in loop
- If
  - *i* not used elsewhere in the loop, and
  - *i* dead after loop
  - *b* not zero
  - Get rid of *i* altogether

```
if (l<r) {
  delta = b*h
  a=a0 + b*l
  N = a0 + b*r
  do {
    M[a] = ...
    a=a+delta
  } while (a<N)
}
```

In general

- Identify
  - loops
  - iteration variables
  - constants
  - Matching use structures

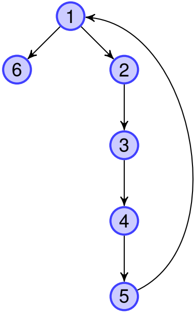
Loops

- Identify loop by node v where back-edge leads to, i.e., $(u, a, v) \in E$ with $v \Rightarrow u$
- Nodes of loop:

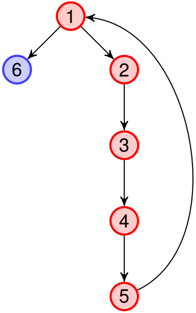
$$\mathrm{loop}[v] = \{w \mid w \to^* v \land v \Rightarrow w\}$$

  - I.e., nodes which can only be reached via v, and from which v can be reached again

Example



Example



Iteration variable

- Variable i, such that

Iteration variable

- Variable i, such that
  - All assignments to i in loop have form $i := i + h$
    - where h is loop constant

Iteration variable

- Variable i, such that
  - All assignments to i in loop have form $i := i + h$
    - where h is loop constant
- Loop constant: Plain constant, or, more sophisticated:

Iteration variable

- Variable *i*, such that
  - All assignments to *i* in loop have form $i := i + h$
    - where *h* is loop constant
- Loop constant: Plain constant, or, more sophisticated:
  - Expression that does not depend on variables modified in loop

Iteration variable

- Variable *i*, such that
  - All assignments to *i* in loop have form $i := i + h$
    - where *h* is loop constant
- Loop constant: Plain constant, or, more sophisticated:
  - Expression that does not depend on variables modified in loop
- Heuristics for application:

Iteration variable

- Variable *i*, such that
  - All assignments to *i* in loop have form $i := i + h$
    - where *h* is loop constant
- Loop constant: Plain constant, or, more sophisticated:
  - Expression that does not depend on variables modified in loop
- Heuristics for application:
  - There is an assignment to *i* in loop

Iteration variable

- Variable *i*, such that
  - All assignments to *i* in loop have form $i := i + h$
    - where *h* is loop constant
- Loop constant: Plain constant, or, more sophisticated:
  - Expression that does not depend on variables modified in loop
- Heuristics for application:
  - There is an assignment to *i* in loop
  - Assignment to *i* executed in every iteration

Strength reduction

- Strength reduction possible for expressions of the form $a_0 + b * i$, such that

Strength reduction

- Strength reduction possible for expressions of the form $a_0 + b * i$, such that
  - a_0, b are loop constants

Strength reduction

- Strength reduction possible for expressions of the form $a_0 + b * i$, such that
  - a_0, b are loop constants
  - i is iteration variable with increment h

Strength reduction

- Strength reduction possible for expressions of the form $a_0 + b * i$, such that
  - a_0, b are loop constants
  - i is iteration variable with increment h
- Introduce temporary variables a and Δ

Strength reduction

- Strength reduction possible for expressions of the form $a_0 + b * i$, such that
  - a_0, b are loop constants
  - i is iteration variable with increment h
- Introduce temporary variables a and Δ
- Initialize $a = a_0 + b * i$ and $\Delta = b * h$ right before loop
  - Note: Loop must be inverted, to avoid extra evaluations!

Strength reduction

- Strength reduction possible for expressions of the form $a_0 + b * i$, such that
  - a_0, b are loop constants
  - i is iteration variable with increment h
- Introduce temporary variables a and Δ
- Initialize $a = a_0 + b * i$ and $\Delta = b * h$ right before loop
  - Note: Loop must be inverted, to avoid extra evaluations!
- Add $a = a + \Delta$ after assignments to i

Strength reduction

- Strength reduction possible for expressions of the form $a_0 + b * i$, such that
  - a_0, b are loop constants
  - i is iteration variable with increment h
- Introduce temporary variables a and Δ
- Initialize $a = a_0 + b * i$ and $\Delta = b * h$ right before loop
  - Note: Loop must be inverted, to avoid extra evaluations!
- Add $a = a + \Delta$ after assignments to i
- Replace expression $a_0 + b * i$ by a

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:
  - $(\rho_0, \mu_0) \in I_{v_0}$
    - for states (ρ_0, μ_0) that satisfy precondition (Here: all states)

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:
  - $(\rho_0, \mu_0) \in I_{v_0}$
    - for states (ρ_0, μ_0) that satisfy precondition (Here: all states)
  - For all edges (u, a, v), we have

$$(\rho, \mu) \in I_u \cap \mathrm{dom}([\![a]\!]) \implies [\![a]\!](\rho, \mu) \in I_v$$

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:
  - $(\rho_0, \mu_0) \in I_{v_0}$
    - for states (ρ_0, μ_0) that satisfy precondition (Here: all states)
  - For all edges (u, a, v), we have

$$(\rho, \mu) \in I_u \cap \mathrm{dom}([\![a]\!]) \implies [\![a]\!](\rho, \mu) \in I_v$$

- Then, we have, for all nodes u: $[\![u]\!] \subseteq I_u$
  - Proof: Induction on paths.
    - Recall $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\ v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$
    - Intuition: All states reachable at u
    - Collecting semantics

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:
  - $(\rho_0, \mu_0) \in I_{v_0}$
    - for states (ρ_0, μ_0) that satisfy precondition (Here: all states)
  - For all edges (u, a, v), we have

$$(\rho, \mu) \in I_u \cap \mathrm{dom}([\![a]\!]) \implies [\![a]\!](\rho, \mu) \in I_v$$

- Then, we have, for all nodes u: $[\![u]\!] \subseteq I_u$
  - Proof: Induction on paths.
    - Recall $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\ v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$
    - Intuition: All states reachable at u
    - Collecting semantics
- And can use this fact to

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:
  - $(\rho_0, \mu_0) \in I_{v_0}$
    - for states (ρ_0, μ_0) that satisfy precondition (Here: all states)
  - For all edges (u, a, v), we have

$$(\rho, \mu) \in I_u \cap \mathrm{dom}([\![a]\!]) \implies [\![a]\!](\rho, \mu) \in I_v$$

- Then, we have, for all nodes u: $[\![u]\!] \subseteq I_u$
  - Proof: Induction on paths.
    - Recall $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\ v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$
    - Intuition: All states reachable at u
    - Collecting semantics
- And can use this fact to
  - Show correctness of program

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:
  - $(\rho_0, \mu_0) \in I_{v_0}$
    - for states (ρ_0, μ_0) that satisfy precondition (Here: all states)
  - For all edges (u, a, v), we have

$$(\rho, \mu) \in I_u \cap \mathrm{dom}([\![a]\!]) \implies [\![a]\!](\rho, \mu) \in I_v$$

- Then, we have, for all nodes u: $[\![u]\!] \subseteq I_u$
  - Proof: Induction on paths.
    - Recall $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\ v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$
    - Intuition: All states reachable at u
    - Collecting semantics
- And can use this fact to
  - Show correctness of program
  - Justify transformations

Excursus: Floyd-Style verification

- Establish invariants for CFG-nodes: I_u for all $u \in V$
- Invariant is set of states
  - Equivalent notation: Characteristic formula over variables/memory
  - E.g., $a = a_0 + b * i$ describes $\{(\rho, \mu) \mid \rho(a) = \rho(a_0) + b * \rho(i)\}$
- Show:
  - $(\rho_0, \mu_0) \in I_{v_0}$
    - for states (ρ_0, μ_0) that satisfy precondition (Here: all states)
  - For all edges (u, a, v), we have

$$(\rho, \mu) \in I_u \cap \mathrm{dom}([\![a]\!]) \implies [\![a]\!](\rho, \mu) \in I_v$$

- Then, we have, for all nodes u: $[\![u]\!] \subseteq I_u$
  - Proof: Induction on paths.
    - Recall $[\![u]\!] := \{(\rho, \mu) \mid \exists \rho_0, \mu_0, \pi.\ v_0 \xrightarrow{\pi} u \wedge [\![\pi]\!](\rho_0, \mu_0) = (\rho, \mu)\}$
    - Intuition: All states reachable at u
    - Collecting semantics
- And can use this fact to
  - Show correctness of program
  - Justify transformations
  - ...

Correctness

- Prove that $a = a_0 + b * i \wedge \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \wedge \Delta = b * h$

Correctness

- Prove that $a = a_0 + b * i \wedge \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \wedge \Delta = b * h$
- Proof:

Correctness

- Prove that $a = a_0 + b * i \wedge \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \wedge \Delta = b * h$
- Proof:
  - Entering loop: Have put initialization right before loop!

Correctness

- Prove that $a = a_0 + b * i \land \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \land \Delta = b * h$
- Proof:
  - Entering loop: Have put initialization right before loop!
  - Edge inside loop:

Correctness

- Prove that $a = a_0 + b * i \wedge \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \wedge \Delta = b * h$
- Proof:
  - Entering loop: Have put initialization right before loop!
  - Edge inside loop:
    - No assignments to Δ, b, and h

Correctness

- Prove that $a = a_0 + b * i \wedge \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \wedge \Delta = b * h$
- Proof:
  - Entering loop: Have put initialization right before loop!
  - Edge inside loop:
    - No assignments to Δ, b, and h
    - Assignment $i := i + h$: Check $a = a_0 + b * i \implies a = a_0 + b * (i + h - h)$.

Correctness

- Prove that $a = a_0 + b * i \wedge \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \wedge \Delta = b * h$
- Proof:
  - Entering loop: Have put initialization right before loop!
  - Edge inside loop:
    - No assignments to Δ, b, and h
    - Assignment $i := i + h$: Check $a = a_0 + b * i \implies a = a_0 + b * (i + h - h)$.
    - Assignment $a := a + \Delta$. Only occurs directly after assignment to i.
      Check $a = a_0 + b * (i - h) \wedge \Delta = b * h \implies a + \Delta = a_0 + b * i$

Correctness

- Prove that $a = a_0 + b * i \wedge \Delta = b * h$ is invariant for all nodes in loop
  - Except the target nodes of assignments to i
    - There, we have $a = a_0 + b * (i - h) \wedge \Delta = b * h$
- Proof:
  - Entering loop: Have put initialization right before loop!
  - Edge inside loop:
    - No assignments to Δ, b, and h
    - Assignment $i := i + h$: Check $a = a_0 + b * i \implies a = a_0 + b * (i + h - h)$.
    - Assignment $a := a + \Delta$. Only occurs directly after assignment to i.
      Check $a = a_0 + b * (i - h) \wedge \Delta = b * h \implies a + \Delta = a_0 + b * i$
    - Other edges: Do not modify variables in invariant

Table of Contents

Peephole Optimization

- Idea: Slide a small window over the code

Peephole Optimization

- Idea: Slide a small window over the code
- Optimize aggressively inside this window

Peephole Optimization

- Idea: Slide a small window over the code
- Optimize aggressively inside this window
- Examples:

Peephole Optimization

- Idea: Slide a small window over the code
- Optimize aggressively inside this window
- Examples:

$$x = x * 2 \quad \rightarrow \quad x = x + x$$

Peephole Optimization

- Idea: Slide a small window over the code
- Optimize aggressively inside this window
- Examples:

$$\begin{aligned} x = x * 2 &\quad\rightarrow\quad x = x + x \\ x = x + 1 &\quad\rightarrow\quad x{+}{+} \end{aligned}$$

Peephole Optimization

- Idea: Slide a small window over the code
- Optimize aggressively inside this window
- Examples:

$$
\begin{aligned}
x = x * 2 &\quad \rightarrow \quad x = x + x \\
x = x + 1 &\quad \rightarrow \quad x{+}{+} \\
x = 5 + a - a &\quad \rightarrow \quad x = 5
\end{aligned}
$$

Peephole Optimization

- Idea: Slide a small window over the code
- Optimize aggressively inside this window
- Examples:

$$\begin{aligned} x = x * 2 &\quad\rightarrow\quad x = x + x \\ x = x + 1 &\quad\rightarrow\quad x{+}{+} \\ x = 5 + a - a &\quad\rightarrow\quad x = 5 \\ x = x &\quad\rightarrow\quad \text{Nop} \end{aligned}$$

Peephole Optimization

- Idea: Slide a small window over the code
- Optimize aggressively inside this window
- Examples:

$$
\begin{aligned}
x = x * 2 &\quad\rightarrow\quad x = x + x \\
x = x + 1 &\quad\rightarrow\quad x{+}{+} \\
x = 5 + a - a &\quad\rightarrow\quad x = 5 \\
x = x &\quad\rightarrow\quad \text{Nop} \\
x = 0 &\quad\rightarrow\quad x = x \oplus x
\end{aligned}
$$

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation
  1. For each node:

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation
  1. For each node:
     - Follow chain of Nop-edges. (Check for loop)

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation
  1. For each node:
     - Follow chain of Nop-edges. (Check for loop)
     - Then redirect all edges on this chain to its end

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation
  1. For each node:
     - Follow chain of Nop-edges. (Check for loop)
     - Then redirect all edges on this chain to its end
  2. For each edge (u, a, v) with (v, Nop, w) and v no other outgoing nodes: Replace by (u, a, w)

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation
  1. For each node:
     - Follow chain of Nop-edges. (Check for loop)
     - Then redirect all edges on this chain to its end
  2. For each edge (u, a, v) with (v, Nop, w) and v no other outgoing nodes: Replace by (u, a, w)
  - Complexity: Linear, $O(|E|)$

Sub-Problem: Elimination of Nop

- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation
  1. For each node:
     - Follow chain of Nop-edges. (Check for loop)
     - Then redirect all edges on this chain to its end
  2. For each edge (u, a, v) with (v, Nop, w) and v no other outgoing nodes: Replace by (u, a, w)

  - Complexity: Linear, $O(|E|)$
    1. No edge redirected twice.
       (For each newly discovered edge, at most one more edge followed)

Sub-Problem: Elimination of Nop

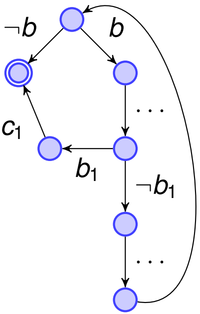
- For edge (u, Nop, v), such that u has no further outgoing edges
  - Identify u and v
  - Attention: Do not collapse Nop-loops
- Implementation
  1. For each node:
     - Follow chain of Nop-edges. (Check for loop)
     - Then redirect all edges on this chain to its end
  2. For each edge (u, a, v) with (v, Nop, w) and v no other outgoing nodes: Replace by (u, a, w)
  - Complexity: Linear, $O(|E|)$
    1. No edge redirected twice.
       (For each newly discovered edge, at most one more edge followed)
    2. For each edge, only one more edge followed

Table of Contents

Motivation

- Translate CFG to instruction list
- Need to insert jumps. No unique translation.
- Crucial for performance

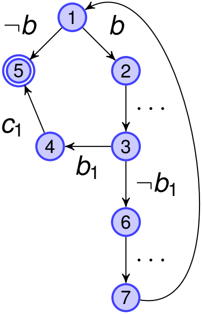
```
while (b) {
  ...
  if (b1) {
    c1;
    break;
  }
  ...
}
```



Motivation

- Translate CFG to instruction list
- Need to insert jumps. No unique translation.
- Crucial for performance

```
while (b) {
   ...
   if (b₁) {
      c₁;
      break;
   }
   ...
}
```



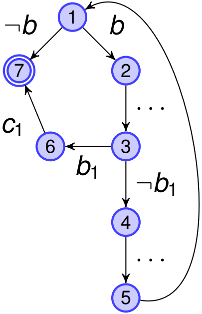

```
1:   jneg b 5
     ...
     jneg b1 6
     c1
5:   halt
6:   ...
     jmp 1
```

Bad linearization, jump in loop

Motivation

- Translate CFG to instruction list
- Need to insert jumps. No unique translation.
- Crucial for performance

```
while (b) {
  ...
  if (b1) {
    c1;
    break;
  }
  ...
}
```




```
1:   jneg b 7
     ...
     jpos b1 6
     ...
     jmp 1
6:   c1
7:   halt
```

Good linearization, jump out of loop

Heuristics

- Avoid jumps inside loops

Heuristics

- Avoid jumps inside loops
- Assign each node its loop nesting depth (temperature)

Heuristics

- Avoid jumps inside loops
- Assign each node its loop nesting depth (temperature)
    - Hotter nodes are in inner loops

Heuristics

- Avoid jumps inside loops
- Assign each node its loop nesting depth (temperature)
    - Hotter nodes are in inner loops
- If jump needs to be inserted: Jump to colder node (out of loop)

Implementation

1. Compute temperatures

Implementation

1. Compute temperatures
    - Compute predominators

Implementation

1. Compute temperatures
   - Compute predominators
   - Identify back edges

Implementation

1. Compute temperatures
   - Compute predominators
   - Identify back edges
   - For each loop head v (i.e., $(u, \_, v)$ is back edge)
     - Increase temperature of nodes in $\mathrm{loop}[v]$
     - Recall:

$$\mathrm{loop}[v] = \{w \mid w \to^* v \land v \Rightarrow w\}$$

Implementation

1. Compute temperatures
   - Compute predominators
   - Identify back edges
   - For each loop head v (i.e., $(u, \_, v)$ is back edge)
     - Increase temperature of nodes in $\mathrm{loop}[v]$
     - Recall:

$$\mathrm{loop}[v] = \{w \mid w \rightarrow^* v \wedge v \Rightarrow w\}$$

2. Linearize

Implementation

1. Compute temperatures
   - Compute predominators
   - Identify back edges
   - For each loop head v (i.e., $(u, \_, v)$ is back edge)
     - Increase temperature of nodes in $\mathrm{loop}[v]$
     - Recall:

$$\mathrm{loop}[v] = \{w \mid w \to^* v \wedge v \Rightarrow w\}$$

2. Linearize
   - Pre-order DFS to number nodes

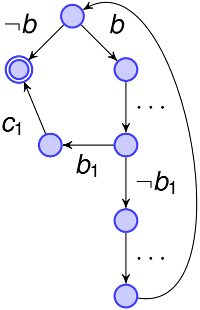
Implementation

1. Compute temperatures
   - Compute predominators
   - Identify back edges
   - For each loop head v (i.e., $(u, \_, v)$ is back edge)
     - Increase temperature of nodes in $\mathrm{loop}[v]$
     - Recall:

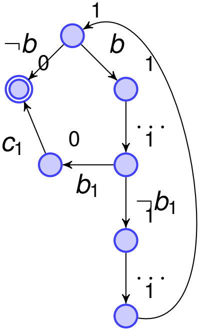
$$\mathrm{loop}[v] = \{w \mid w \to^* v \wedge v \Rightarrow w\}$$

2. Linearize
   - Pre-order DFS to number nodes
   - Visit hotter successors first

Example



Example



Example

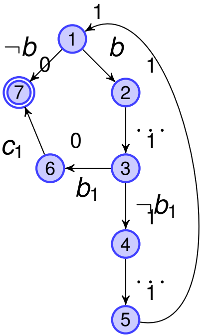


Table of Contents

Motivation

- Program needs to be compiled to specific hardware

Motivation

- Program needs to be compiled to specific hardware
- Which has some features that can be exploited for optimization, e.g.

Motivation

- Program needs to be compiled to specific hardware
- Which has some features that can be exploited for optimization, e.g.
  - Registers

Motivation

- Program needs to be compiled to specific hardware
- Which has some features that can be exploited for optimization, e.g.
  - Registers
  - Pipelines

Motivation

- Program needs to be compiled to specific hardware
- Which has some features that can be exploited for optimization, e.g.
  - Registers
  - Pipelines
  - Caches

Motivation

- Program needs to be compiled to specific hardware
- Which has some features that can be exploited for optimization, e.g.
    - Registers
    - Pipelines
    - Caches
    - Multiple Processors

Table of Contents

Nomenclature

- Variables Var, e.g. $x, y, z, \ldots$: Variables in source program (formerly also called registers)

Nomenclature

- Variables Var, e.g. $x, y, z, \ldots$: Variables in source program (formerly also called registers)
- Registers Reg, e.g. $R_1, R_2, \ldots$: Registers after register allocation

Motivation

- Processor only has limited number of registers

Motivation

- Processor only has limited number of registers
- Variables need to be mapped to those registers

Motivation

- Processor only has limited number of registers
- Variables need to be mapped to those registers
- If no more registers free: Spill to memory

Motivation

- Processor only has limited number of registers
- Variables need to be mapped to those registers
- If no more registers free: Spill to memory
  - Expensive!

Motivation

- Processor only has limited number of registers
- Variables need to be mapped to those registers
- If no more registers free: Spill to memory
  - Expensive!
- Want to map as much variables as possible to registers

Example

```
1: x=M[a]
2: y=x+1
3: if (y=0) {
4:   z=x*x
5:   M[a]=z
   } else {
7:   t=-y*y
8:   M[a]=t
9: }
```

- How many registers are needed?

Example

```
1: x=M[a]
2: y=x+1
3: if (y=0) {
4:   z=x*x
5:   M[a]=z
   } else {
7:   t=-y*y
8:   M[a]=t
9: }
```

- How many registers are needed?
  Assuming all variables dead at 9

Example

```
1: x=M[a]
2: y=x+1
3: if (y=0) {
4:    z=x*x
5:    M[a]=z
   } else {
7:    t=-y*y
8:    M[a]=t
9: }
```

- How many registers are needed?
  Assuming all variables dead at 9
- Variables: a, x, y, z, t.

Example

```
1: R1=M[R3]
2: R2=R1+1
3: if (R2=0) {
4:    R1=R1*R1
5:    M[R3]=R1
   } else {
7:    R1=-R2*R2
8:    M[R3]=R1
9: }
```

- How many registers are needed? Assuming all variables dead at 9
- Variables: a, x, y, z, t.
- Three registers suffice: $x, z, t \mapsto R_1, \quad y \mapsto R_2, \quad a \mapsto R_3$

Live Ranges

- Live range of variable x: $L[x] := \{u \mid x \in L[u]\}$

Live Ranges

- Live range of variable x: $L[x] := \{u \mid x \in L[u]\}$
  - Set of nodes where x is alive:

Live Ranges

- Live range of variable x: $L[x] := \{u \mid x \in L[u]\}$
  - Set of nodes where x is alive:
  - Analogously: True live range

Live Ranges

- Live range of variable x: $L[x] := \{u \mid x \in L[u]\}$
  - Set of nodes where x is alive:
  - Analogously: True live range
- Observation: Two variables can be mapped to same register, if their live ranges do not overlap

Example

```
                // L          a x y z t
1: x=M[a]       // {a}        1
2: y=x+1        // {a,x}      1 1
3: if (y=0) {   // {a,x,y}    1 1 1
4:   z=x*x      // {a,x}      1 1
5:   M[a]=z     // {a,z}      1     1
   } else {
7:   t=-y*y     // {a,y}      1   1
8:   M[a]=t     // {a,t}      1       1
9: }            // {}
```

Interference graph

- $I = (\mathrm{Var}, E_I)$, with $(x, y) \in E_i$ iff $x \neq y$ and $L[x] \cap L[y] \neq \emptyset$

Interference graph

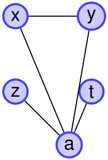
- $I = (\mathrm{Var}, E_I)$, with $(x, y) \in E_i$ iff $x \neq y$ and $L[x] \cap L[y] \neq \emptyset$
  - Graph over variables. Edge iff live ranges overlap.

Interference graph

- $I = (\mathrm{Var}, E_I)$, with $(x, y) \in E_i$ iff $x \neq y$ and $L[x] \cap L[y] \neq \emptyset$
  - Graph over variables. Edge iff live ranges overlap.
  - I is called interference graph

Interference graph

- $I = (\mathrm{Var}, E_I)$, with $(x, y) \in E_i$ iff $x \neq y$ and $L[x] \cap L[y] \neq \emptyset$
  - Graph over variables. Edge iff live ranges overlap.
  - I is called interference graph
- In our example:



Last lecture

- Peephole optimization, removal of NOP-edges
- Linearization
  - Temperature of nodes = loop nesting depth
  - Preferably jump to colder nodes
- Register allocation
  - Minimal coloring of interference graph
    - NP-hard

Background: Minimal graph coloring

- Given: Graph (V, E)

Background: Minimal graph coloring

- Given: Graph (V, E)
- Find coloring of nodes $c : V \to \mathbb{N}$, such that

Background: Minimal graph coloring

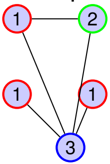
- Given: Graph (V, E)
- Find coloring of nodes $c : V \to \mathbb{N}$, such that
  - $(u, v) \in E \implies c(u) \neq c(v)$
    - I.e., adjacent nodes have different colors

Background: Minimal graph coloring

- Given: Graph (V, E)
- Find coloring of nodes $c : V \to \mathbb{N}$, such that
  - $(u, v) \in E \implies c(u) \neq c(v)$
    - I.e., adjacent nodes have different colors
  - $\max\{c(v) \mid v \in V\}$ is minimal

Background: Minimal graph coloring

- Given: Graph (V, E)
- Find coloring of nodes $c : V \to \mathbb{N}$, such that
  - $(u, v) \in E \implies c(u) \neq c(v)$
    - I.e., adjacent nodes have different colors
  - $\max\{c(v) \mid v \in V\}$ is minimal
- Example:



Complexity

- Finding a minimum graph coloring is hard
  - Precisely: NP-complete to determine whether there is a coloring with at most k colors, for $k > 2$.

Complexity

- Finding a minimum graph coloring is hard
  - Precisely: NP-complete to determine whether there is a coloring with at most k colors, for $k > 2$.
- Need heuristics

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum
  - Regard crown graph C_n, which is a complete bipartite graph over $2n$ nodes, with a perfect matching removed.

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum
  - Regard crown graph C_n, which is a complete bipartite graph over $2n$ nodes, with a perfect matching removed.
  - $C_n = (a_i, b_i \mid i \in 1 \dots n, (a_i, b_j) \mid i \neq j)$

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum
  - Regard crown graph C_n, which is a complete bipartite graph over $2n$ nodes, with a perfect matching removed.
  - $C_n = (a_i, b_i \mid i \in 1 \dots n, (a_i, b_j) \mid i \neq j)$
  - Minimal coloring uses two colors: One for the as, and one for the bs

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum
    - Regard crown graph C_n, which is a complete bipartite graph over $2n$ nodes, with a perfect matching removed.
    - $C_n = (a_i, b_i \mid i \in 1 \ldots n, (a_i, b_j) \mid i \neq j)$
    - Minimal coloring uses two colors: One for the as, and one for the bs
    - Greedy coloring with order $a_1, b_1, a_2, b_2, \ldots$ uses n colors

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum
  - Regard crown graph C_n, which is a complete bipartite graph over $2n$ nodes, with a perfect matching removed.
  - $C_n = (a_i, b_i \mid i \in 1 \dots n, (a_i, b_j) \mid i \neq j)$
  - Minimal coloring uses two colors: One for the as, and one for the bs
  - Greedy coloring with order $a_1, b_1, a_2, b_2, \dots$ uses n colors
- Node ordering heuristics

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum
  - Regard crown graph C_n, which is a complete bipartite graph over $2n$ nodes, with a perfect matching removed.
  - $C_n = (a_i, b_i \mid i \in 1 \dots n, (a_i, b_j) \mid i \neq j)$
  - Minimal coloring uses two colors: One for the as, and one for the bs
  - Greedy coloring with order $a_1, b_1, a_2, b_2, \dots$ uses n colors
- Node ordering heuristics
  - Nodes of high degree first

Greedy Heuristics

- Iterate over nodes, and assign minimum color different from already colored neighbors
- Can be implemented using DFS
- In theory, result may be arbitrarily far from optimum
  - Regard crown graph C_n, which is a complete bipartite graph over $2n$ nodes, with a perfect matching removed.
  - $C_n = (a_i, b_i \mid i \in 1 \dots n, (a_i, b_j) \mid i \neq j)$
  - Minimal coloring uses two colors: One for the *a*s, and one for the *b*s
  - Greedy coloring with order $a_1, b_1, a_2, b_2, \dots$ uses n colors
- Node ordering heuristics
  - Nodes of high degree first
  - Here: Pre-order DFS

Greedy heuristics, pseudocode

```
color(u):
  n = { v | (u,v) in E }
  c(u) = min i. i>=0 and forall v in n. i != c(v)
  for v in n
    if (c(v)==-1) color(v)

main:
  for u in V do c(u) = -1;

  for u in V do
    if c(u)==-1 then color(u)
```

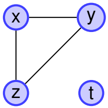
Live Range Splitting

- Consider basic block,
  - i.e., sequence of statements, no jumps in/from in between
  - $(u, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{n-1}, a_n, v)$, with no other edges touching the v_i.

Live Range Splitting

- Consider basic block,
  - i.e., sequence of statements, no jumps in/from in between
  - $(u, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{n-1}, a_n, v)$, with no other edges touching the v_i.
- Example:

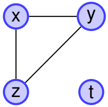
```
x=M[0] //
y=M[1] // x
t=x+y  // xy
M[2]=t //    t
x=M[4] //
z=M[5] // x
t=x+z  // x z
M[6]=t //    t
y=M[7] //
z=M[8] //  y
t=y+z  //  yz
M[9]=t //    t
```



Live Range Splitting

- Consider basic block,
  - i.e., sequence of statements, no jumps in/from in between
  - $(u, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{n-1}, a_n, v)$, with no other edges touching the v_i.
- Example:

```
x=M[0]  //
y=M[1]  //  x
t=x+y   //  xy
M[2]=t  //     t
x=M[4]  //
z=M[5]  //  x
t=x+z   //  x z
M[6]=t  //     t
y=M[7]  //
z=M[8]  //   y
t=y+z   //   yz
M[9]=t  //     t
```

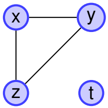


- Requires 3 registers

Live Range Splitting

- Consider basic block,
  - i.e., sequence of statements, no jumps in/from in between
  - $(u, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{n-1}, a_n, v)$, with no other edges touching the v_i.
- Example:

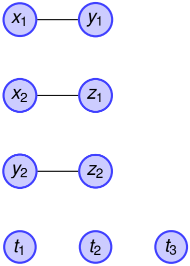
```
x=M[0] //
y=M[1] // x
t=x+y  // xy
M[2]=t //    t
x=M[4] //
z=M[5] // x
t=x+z  // x z
M[6]=t //    t
y=M[7] //
z=M[8] //  y
t=y+z  //  yz
M[9]=t //    t
```



- Requires 3 registers
- But can do same program with two registers!

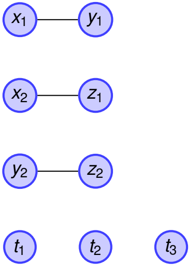
Live range splitting

$$
\begin{array}{ll}
x_1 = M[0] & // \\
y_1 = M[1] & //\ x_1 \\
t_1 = x_1 + y_1 & //\ x_1\ y_1 \\
M[2] = t_1 & //\ t_1 \\
x_2 = M[4] & // \\
z_1 = M[5] & //\ x_2 \\
t_2 = x_2 + z_1 & //\ x_2\ z_1 \\
M[6] = t_2 & //\ t_2 \\
y_2 = M[7] & // \\
z_2 = M[8] & //\ y_2 \\
t_3 = y_2 + z_2 & //\ y_2\ z_2 \\
M[9] = t_3 & //\ t_3
\end{array}
$$



Live range splitting

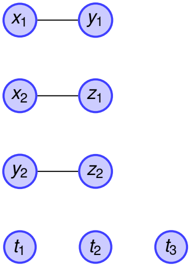
```
x1=M[0]      //
y1=M[1]      // x1
t1=x1+y1     // x1 y1
M[2]=t1      // t1
x2=M[4]      //
z1=M[5]      // x2
t2=x2+z1     // x2 z1
M[6]=t2      // t2
y2=M[7]      //
z2=M[8]      // y2
t3=y2+z2     // y2 z2
M[9]=t3      // t3
```



- In general: Rename variable if it is redefined

Live range splitting

```
x₁=M[0]     //
y₁=M[1]     // x₁
t₁=x₁+y₁    // x₁ y₁
M[2]=t₁     // t₁
x₂=M[4]     //
z₁=M[5]     // x₂
t₂=x₂+z₁    // x₂ z₁
M[6]=t₂     // t₂
y₂=M[7]     //
z₂=M[8]     // y₂
t₃=y₂+z₂    // y₂ z₂
M[9]=t₃     // t₃
```



- In general: Rename variable if it is redefined
- The interference graph forms an interval graph.

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$
  - I.e., edges between overlapping intervals

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$
  - I.e., edges between overlapping intervals
- On interval graphs, coloring can be determined efficiently

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$
  - I.e., edges between overlapping intervals
- On interval graphs, coloring can be determined efficiently
  - Use greedy algorithm, order intervals by left endpoints

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$
  - I.e., edges between overlapping intervals
- On interval graphs, coloring can be determined efficiently
  - Use greedy algorithm, order intervals by left endpoints
  - Proof idea:

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$
  - I.e., edges between overlapping intervals
- On interval graphs, coloring can be determined efficiently
  - Use greedy algorithm, order intervals by left endpoints
  - Proof idea:
    - After coloring all nodes with left endpoint i, there are exactly $o(i)$ colors allocated.

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$
  - I.e., edges between overlapping intervals
- On interval graphs, coloring can be determined efficiently
  - Use greedy algorithm, order intervals by left endpoints
  - Proof idea:
    - After coloring all nodes with left endpoint i, there are exactly $o(i)$ colors allocated.
    - Where $o(i) := |\{v \in V \mid i \in v\}|$ - number of nodes containing i.

Interval Graphs

- Nodes are intervals over the real numbers (here: natural numbers).
- Edge between $[i, j]$ and $[k, l]$, iff $[i, j] \cap [k, l] \neq \emptyset$
  - I.e., edges between overlapping intervals
- On interval graphs, coloring can be determined efficiently
  - Use greedy algorithm, order intervals by left endpoints
  - Proof idea:
    - After coloring all nodes with left endpoint i, there are exactly $o(i)$ colors allocated.
    - Where $o(i) := |\{v \in V \mid i \in v\}|$ - number of nodes containing i.
    - Obviously, there is no coloring with less than $\max\{o(i) \mid i \in \mathbb{N}\}$ colors

Wrap-up

- Heuristics required for register allocation

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient
  - Spill registers into memory (usually into stack)

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient
  - Spill registers into memory (usually into stack)
  - Preferably, hold variables from inner loops in registers

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient
  - Spill registers into memory (usually into stack)
  - Preferably, hold variables from inner loops in registers
- For basic blocks:

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient
  - Spill registers into memory (usually into stack)
  - Preferably, hold variables from inner loops in registers
- For basic blocks:
  - Efficient optimal register allocation

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient
    - Spill registers into memory (usually into stack)
    - Preferably, hold variables from inner loops in registers
- For basic blocks:
    - Efficient optimal register allocation
    - Only if live ranges are split

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient
  - Spill registers into memory (usually into stack)
  - Preferably, hold variables from inner loops in registers
- For basic blocks:
  - Efficient optimal register allocation
  - Only if live ranges are split
- Splitting live ranges for complete program

Wrap-up

- Heuristics required for register allocation
- If number of available registers not sufficient
  - Spill registers into memory (usually into stack)
  - Preferably, hold variables from inner loops in registers
- For basic blocks:
  - Efficient optimal register allocation
  - Only if live ranges are split
- Splitting live ranges for complete program

$\Longrightarrow$ Single static assignment form (SSA)

Table of Contents

Idea

- Generalize live-range splitting to programs

Idea

- Generalize live-range splitting to programs
- Proceed in two steps

Idea

- Generalize live-range splitting to programs
- Proceed in two steps
  1. Transform program such that every program point v is reached by at most one definition of variable x which is live at v.

Idea

- Generalize live-range splitting to programs
- Proceed in two steps
  1. Transform program such that every program point v is reached by at most one definition of variable x which is live at v.
  2. Introduce a separate variant x_i for each definition of x, and replace occurrences of x by the reaching variants

SSA, first transformation

- Assume that start node has no incoming edges.

SSA, first transformation

- Assume that start node has no incoming edges.
    - Otherwise, add new start node before transformation

SSA, first transformation

- Assume that start node has no incoming edges.
  - Otherwise, add new start node before transformation
- At incoming edges to join points v, i.e., nodes with > 1 incoming edges:

SSA, first transformation

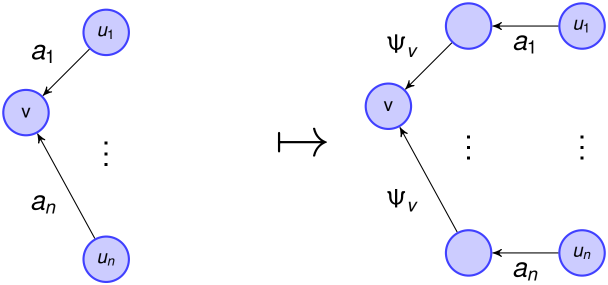
- Assume that start node has no incoming edges.
  - Otherwise, add new start node before transformation
- At incoming edges to join points v, i.e., nodes with > 1 incoming edges:
  - Introduce new edges, labeled with Ψ_v

SSA, first transformation

- Assume that start node has no incoming edges.
  - Otherwise, add new start node before transformation
- At incoming edges to join points v, i.e., nodes with > 1 incoming edges:
  - Introduce new edges, labeled with Ψ_v
  - For now: $\Psi_v := \mathrm{Nop}$

SSA, first transformation

- Assume that start node has no incoming edges.
  - Otherwise, add new start node before transformation
- At incoming edges to join points v, i.e., nodes with > 1 incoming edges:
  - Introduce new edges, labeled with Ψ_v
  - For now: $\Psi_v := \mathrm{Nop}$



Reaching definitions

- Compute reaching definitions for each variable x and program point v.

Reaching definitions

- Compute reaching definitions for each variable x and program point v.
  - Intuitively: The definitions that determined the value of x

Reaching definitions

- Compute reaching definitions for each variable x and program point v.
  - Intuitively: The definitions that determined the value of x
- Analyzed by forward may analysis, over domain 2^{Defs}

Reaching definitions

- Compute reaching definitions for each variable x and program point v.
  - Intuitively: The definitions that determined the value of x
- Analyzed by forward may analysis, over domain 2^{Defs}
  - where $\mathrm{Defs} = \mathrm{Var} \times V$

Reaching definitions

- Compute reaching definitions for each variable x and program point v.
  - Intuitively: The definitions that determined the value of x
- Analyzed by forward may analysis, over domain 2^{Defs}
  - where $\mathrm{Defs} = \mathrm{Var} \times V$

$$
\begin{aligned}
[\![(u, x := e, v)]\!]^{\#} R &= R \setminus \mathrm{Defs}(x) \cup \{(x, v)\} \\
[\![(u, x := M[e], v)]\!]^{\#} R &= R \setminus \mathrm{Defs}(x) \cup \{(x, v)\} \\
[\![(u, a, v)]\!]^{\#} R &= R && \text{for other edges}
\end{aligned}
$$

Reaching definitions

- Compute reaching definitions for each variable x and program point v.
  - Intuitively: The definitions that determined the value of x
- Analyzed by forward may analysis, over domain 2^{Defs}
  - where $\mathrm{Defs} = \mathrm{Var} \times V$

$$
\begin{aligned}
[\![(u, x := e, v)]\!]^{\#} R &= R \setminus \mathrm{Defs}(x) \cup \{(x, v)\} \\
[\![(u, x := M[e], v)]\!]^{\#} R &= R \setminus \mathrm{Defs}(x) \cup \{(x, v)\} \\
[\![(u, a, v)]\!]^{\#} R &= R \qquad \text{for other edges}
\end{aligned}
$$

  - Initial value: $R_0 := \{(x, v_0) \mid x \in \mathrm{Var}\}$

Reaching definitions

- Compute reaching definitions for each variable x and program point v.
  - Intuitively: The definitions that determined the value of x
- Analyzed by forward may analysis, over domain 2^{Defs}
  - where $\mathrm{Defs} = \mathrm{Var} \times V$

$$
\begin{aligned}
[\![(u, x := e, v)]\!]^{\#} R &= R \setminus \mathrm{Defs}(x) \cup \{(x, v)\} \\
[\![(u, x := M[e], v)]\!]^{\#} R &= R \setminus \mathrm{Defs}(x) \cup \{(x, v)\} \\
[\![(u, a, v)]\!]^{\#} R &= R \qquad \text{for other edges}
\end{aligned}
$$

  - Initial value: $R_0 := \{(x, v_0) \mid x \in \mathrm{Var}\}$
    - Intuitively: Interpret program start as end-point of definition for every variable

Simultaneous assignments

- At incoming edges to join points v:

Simultaneous assignments

- At incoming edges to join points v:
- Set $\Psi_v := \{x = x \mid x \in L[v] \land |R[v] \cap \mathrm{Defs}(x)| > 1\}$

Simultaneous assignments

- At incoming edges to join points v:
- Set $\Psi_v := \{x = x \mid x \in L[v] \wedge |R[v] \cap \mathrm{Defs}(x)| > 1\}$
  - Assignment $x = x$ for each live variable that has more than one reaching definition

Simultaneous assignments

- At incoming edges to join points v:
- Set $\Psi_v := \{x = x \mid x \in L[v] \wedge |R[v] \cap \mathrm{Defs}(x)| > 1\}$
  - Assignment $x = x$ for each live variable that has more than one reaching definition
  - Simultaneous assignment

Example

```
x:=M[I]
y:=1
while (x>0) {
  y=x*y
  x=x-1
}
M[R]=y
```

Example

```
1: x:=M[I]
2: y:=1
3: if not (x>0) goto 6;
4:   y=x*y
5:   x=x-1;
     goto 3
6: M[R]=y
7:
```

Example

```
1: x:=M[I]
2: y:=1
A: Nop    // Psi3
3: if not (x>0) goto 6
4:   y=x*y
5:   x=x-1
B: Nop    // Psi3
   goto 3
6: M[R]=y
7:
```

Example

```
1: x:=M[I]                  // {}    {(x,1),(y,1)}
2: y:=1                     // {x}   {(x,2),(y,1)}
A: Nop    // Psi3           // {x,y} {(x,2),(y,A)}
3: if not (x>0) goto 6;     // {x,y} {(x,2),(x,B),(y,A),(y,5)}
4:   y=x*y                  // {x,y} {(x,2),(x,B),(y,A),(y,5)}
5:   x=x-1                  // {x,y} {(x,2),(x,B),(y,5)}
B: Nop    // Psi3           // {x,y} {(x,B),(y,5)}
   goto 3
6: M[R]=y                   // {y}   {(x,2),(x,B),(y,A),(y,5)}
7:                          // {}    {(x,2),(x,B),(y,A),(y,5)}
```

Example

```
1: x:=M[I]                  // {}     {(x,1),(y,1)}
2: y:=1                     // {x}    {(x,2),(y,1)}
A: x=x|y=y                  // {x,y}  {(x,2),(y,A)}
3: if not (x>0) goto 6;     // {x,y}  {(x,2),(x,B),(y,A),(y,5)}
4:   y=x*y                  // {x,y}  {(x,2),(x,B),(y,A),(y,5)}
5:   x=x-1                  // {x,y}  {(x,2),(x,B),(y,5)}
B: x=x|y=y                  // {x,y}  {(x,B),(y,5)}
   goto 3
6: M[R]=y                   // {y}    {(x,2),(x,B),(y,A),(y,5)}
7:                          // {}     {(x,2),(x,B),(y,A),(y,5)}
```

Discussion

- This ensures that only one definition of a variable reaches each program point

Discussion

- This ensures that only one definition of a variable reaches each program point
  - Identifying the definitions by simultaneous assignments on edges to same join points

Discussion

- This ensures that only one definition of a variable reaches each program point
  - Identifying the definitions by simultaneous assignments on edges to same join points
- However, we may introduce superfluous simultaneous definitions

Discussion

- This ensures that only one definition of a variable reaches each program point
  - Identifying the definitions by simultaneous assignments on edges to same join points
- However, we may introduce superfluous simultaneous definitions
- Consider, e.g.

Discussion

- This ensures that only one definition of a variable reaches each program point
  - Identifying the definitions by simultaneous assignments on edges to same join points
- However, we may introduce superfluous simultaneous definitions
- Consider, e.g.

```
1: if (*) goto 3
2:    x=1
  goto 4
3:    x=2
4: if (*) goto 6
5:    M[0]=x
6: M[1]=x
7: HALT
```

Discussion

- This ensures that only one definition of a variable reaches each program point
  - Identifying the definitions by simultaneous assignments on edges to same join points
- However, we may introduce superfluous simultaneous definitions
- Consider, e.g.

```
1: if (*) goto 3
2:   x=1
A:   x=x
  goto 4
3:   x=2
B:   x=x
4: if (*) goto C
5:   M[0]=x
D:   x=x
6: M[1]=x
7: HALT

C: x=x
  goto 6
```

Improved Algorithm

- Introduce assignment $x = x$ before node v only if reaching definitions of x at incoming edges to v differ

Improved Algorithm

- Introduce assignment $x = x$ before node v only if reaching definitions of x at incoming edges to v differ
- Repeat until each node v is reached by exactly one definition for each variable live at v

Improved Algorithm

- Introduce assignment $x = x$ before node v only if reaching definitions of x at incoming edges to v differ
- Repeat until each node v is reached by exactly one definition for each variable live at v
  - Extend analysis for reaching definitions by $[\![(u, \{x = x \mid x \in X\}, v)]\!]^{\#} R := R \setminus \mathrm{Defs}(X) \cup X \times \{v\}$

Theorem

For a CFG with n variables, and m nodes with in-degree greater one, the above algorithm terminates after at most $n(m+1)$ *rounds.*

Improved Algorithm

- Introduce assignment $x = x$ before node v only if reaching definitions of x at incoming edges to v differ
- Repeat until each node v is reached by exactly one definition for each variable live at v
  - Extend analysis for reaching definitions by
    $[\![(u, \{x = x \mid x \in X\}, v)]\!]^\# R := R \setminus \mathrm{Defs}(X) \cup X \times \{v\}$

Theorem

For a CFG with n variables, and m nodes with in-degree greater one, the above algorithm terminates after at most $n(m+1)$ rounds.

- The efficiency depends on the number of rounds

Improved Algorithm

- Introduce assignment $x = x$ before node v only if reaching definitions of x at incoming edges to v differ
- Repeat until each node v is reached by exactly one definition for each variable live at v
  - Extend analysis for reaching definitions by $[\![(u, \{x = x \mid x \in X\}, v)]\!]^{\#} R := R \setminus \mathrm{Defs}(X) \cup X \times \{v\}$

Theorem

For a CFG with n variables, and m nodes with in-degree greater one, the above algorithm terminates after at most $n(m+1)$ rounds.

- The efficiency depends on the number of rounds
  - For well-structured CFGs, we only need one round

Improved Algorithm

- Introduce assignment $x = x$ before node v only if reaching definitions of x at incoming edges to v differ
- Repeat until each node v is reached by exactly one definition for each variable live at v
  - Extend analysis for reaching definitions by $[\![(u, \{x = x \mid x \in X\}, v)]\!]^{\#} R := R \setminus \mathrm{Defs}(X) \cup X \times \{v\}$

Theorem

For a CFG with n variables, and m nodes with in-degree greater one, the above algorithm terminates after at most $n(m+1)$ rounds.

- The efficiency depends on the number of rounds
  - For well-structured CFGs, we only need one round
    - Example where 2 rounds are required on board.

Improved Algorithm

- Introduce assignment $x = x$ before node v only if reaching definitions of x at incoming edges to v differ
- Repeat until each node v is reached by exactly one definition for each variable live at v
  - Extend analysis for reaching definitions by $[\![(u, \{x = x \mid x \in X\}, v)]\!]^{\#} R := R \setminus \mathrm{Defs}(X) \cup X \times \{v\}$

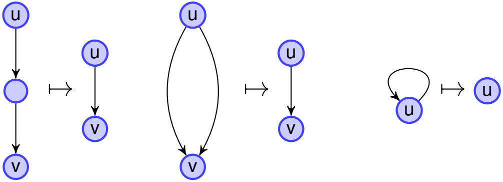
Theorem

For a CFG with n variables, and m nodes with in-degree greater one, the above algorithm terminates after at most $n(m+1)$ rounds.

- The efficiency depends on the number of rounds
  - For well-structured CFGs, we only need one round
    - Example where 2 rounds are required on board.
  - We always may terminate after k rounds by using naive algorithm

Well-structured CFGs

- A CFG is well-structured, if it can be reduced to a single edge or vertex by the following transformations



Examples

- Flowgraphs produced by only using the following control-flow commands are well-structured
  - if, while, do-while, for

Examples

- Flowgraphs produced by only using the following control-flow commands are well-structured
  - if, while, do-while, for
- Break/Continue may break well-structuredness

Examples

- Flowgraphs produced by only using the following control-flow commands are well-structured
  - if, while, do-while, for
- Break/Continue may break well-structuredness
- Some examples on board

Second phase

- Assume, each program point u is reached by exactly one definition $(x, w) \in R[u]$ for each variable x live at u

Second phase

- Assume, each program point u is reached by exactly one definition $(x, w) \in R[u]$ for each variable x live at u
- Define $\Phi_u(x) := x_w$ for the w with $(x, w) \in R[u]$

Second phase

- Assume, each program point u is reached by exactly one definition $(x, w) \in R[u]$ for each variable x live at u
- Define $\Phi_u(x) := x_w$ for the w with $(x, w) \in R[u]$
- Transform edge (u, a, v) to $(u, T_{u,v}(a), v)$, where

$$
\begin{aligned}
T_{u,v}(\mathrm{Nop}) &= \mathrm{Nop} \\
T_{u,v}(\mathrm{Neg}(e)) &= \mathrm{Neg}(\Phi_u(e)) \\
T_{u,v}(\mathrm{Pos}(e)) &= \mathrm{Pos}(\Phi_u(e)) \\
T_{u,v}(x = e) &= x_v = \Phi_u(e) \\
T_{u,v}(x = M[e]) &= x_v = M[\Phi_u(e)] \\
T_{u,v}(M[e_1] = e_2) &= M[\Phi_u(e_1)] = \Phi_u(e_2) \\
T_{u,v}(\{x = x \mid x \in X\}) &= \{x_v = \Phi_u(x) \mid x \in X\}
\end{aligned}
$$

and $\Phi_u(e)$ applies Φ_u to every variable in e

Example

```
1: x:=M[0]
2: y:=1
A: x=x|y=y
3: if not (x>0) goto 6;
4:   y=x*y
5:   x=x-1
B: x=x|y=y
   goto 3
6: M[1]=y
7:
```

Example

```
1: x_2:=M[0]
2: y_A:=1
A: x_3=x_2|y_3=y_A
3: if not (x_3>0) goto 6;
4:   y_5=x_3*y_3
5:   x_B=x_3-1
B:   x_3=x_B|y_3=y_5
   goto 3
6: M[1]=y_3
7:
```

Register Allocation for SSA form

Theorem

Assume that every program point is reachable from start and the program is in SSA form without assignments to dead variables.
Let λ denote the maximal number of simultaneously live variables and G the interference graph of the program variables. Then:

$$\lambda = \omega(G) = \chi(G)$$

where $\omega(G), \chi(G)$ are the maximal size of a clique in G and the minimal number of colors for G, respectively.
A minimal coloring of G, i.e., an optimal register allocation can be found in polynomial time.

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord
  - i.e., an edge between two nodes of the cycle that is, itself, not part of the cycle

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord
  - i.e., an edge between two nodes of the cycle that is, itself, not part of the cycle
- A graph is chordal, iff it has a perfect elimination order

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord
  - i.e., an edge between two nodes of the cycle that is, itself, not part of the cycle
- A graph is chordal, iff it has a perfect elimination order
  - I.e., an ordering of the nodes, such that each node u and all adjacent nodes $v > u$ form a clique.

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord
  - i.e., an edge between two nodes of the cycle that is, itself, not part of the cycle
- A graph is chordal, iff it has a perfect elimination order
  - I.e., an ordering of the nodes, such that each node u and all adjacent nodes $v > u$ form a clique.
- Using a reverse perfect elimination ordering as node ordering for the greedy algorithm yields a minimal coloring

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord
  - i.e., an edge between two nodes of the cycle that is, itself, not part of the cycle
- A graph is chordal, iff it has a perfect elimination order
  - I.e., an ordering of the nodes, such that each node u and all adjacent nodes $v > u$ form a clique.
- Using a reverse perfect elimination ordering as node ordering for the greedy algorithm yields a minimal coloring
- For graphs in SSA form, the dominance relation induces a perfect elimination ordering on the interference graph

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord
  - i.e., an edge between two nodes of the cycle that is, itself, not part of the cycle
- A graph is chordal, iff it has a perfect elimination order
  - I.e., an ordering of the nodes, such that each node u and all adjacent nodes $v > u$ form a clique.
- Using a reverse perfect elimination ordering as node ordering for the greedy algorithm yields a minimal coloring
- For graphs in SSA form, the dominance relation induces a perfect elimination ordering on the interference graph
  - Thus, we do not even need to construct the interference graph:

Background: Register allocation for SSA

- Interference graphs of program in SSA-form are chordal
  - I.e., every cycle of length > 3 has a chord
  - i.e., an edge between two nodes of the cycle that is, itself, not part of the cycle
- A graph is chordal, iff it has a perfect elimination order
  - I.e., an ordering of the nodes, such that each node u and all adjacent nodes $v > u$ form a clique.
- Using a reverse perfect elimination ordering as node ordering for the greedy algorithm yields a minimal coloring
- For graphs in SSA form, the dominance relation induces a perfect elimination ordering on the interference graph
  - Thus, we do not even need to construct the interference graph:
  - Just traverse CFG with pre-order DFS, and assign registers first-come first serve.

Background: Adjusting register pressure

- Via λ, we can simply estimate the amount of required registers (register pressure)

Background: Adjusting register pressure

- Via λ, we can simply estimate the amount of required registers (register pressure)
- And only perform optimizations that increase register pressure if still enough registers available

Discussion

- With SSA form, we get a cheap, optimal register allocation

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments
  - Which are meant to be executed simultaneously

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments
  - Which are meant to be executed simultaneously
  - Note: Original variables may be mapped to arbitrary registers

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments
  - Which are meant to be executed simultaneously
  - Note: Original variables may be mapped to arbitrary registers
  - I.e., $R_1 = R_2 \mid R_2 = R_1$ swaps registers R_1 and R_2

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments
  - Which are meant to be executed simultaneously
  - Note: Original variables may be mapped to arbitrary registers
  - I.e., $R_1 = R_2 \mid R_2 = R_1$ swaps registers R_1 and R_2
- We need to translate these to machine instructions

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments
  - Which are meant to be executed simultaneously
  - Note: Original variables may be mapped to arbitrary registers
  - I.e., $R_1 = R_2 \mid R_2 = R_1$ swaps registers R_1 and R_2
- We need to translate these to machine instructions
  - Use auxiliary register: $R_3 = R_1$; $R_1 = R_2$; $R_2 = R_3$

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments
  - Which are meant to be executed simultaneously
  - Note: Original variables may be mapped to arbitrary registers
  - I.e., $R_1 = R_2 \mid R_2 = R_1$ swaps registers R_1 and R_2
- We need to translate these to machine instructions
  - Use auxiliary register: $R_3 = R_1; R_1 = R_2; R_2 = R_3$
  - Use XOR-swap: $R_1 = R_1 \oplus R_2; R_2 = R_1 \oplus R_2; R_1 = R_1 \oplus R_2$

Discussion

- With SSA form, we get a cheap, optimal register allocation
- But: We still have the simultaneous assignments
  - Which are meant to be executed simultaneously
  - Note: Original variables may be mapped to arbitrary registers
  - I.e., $R_1 = R_2 \mid R_2 = R_1$ swaps registers R_1 and R_2
- We need to translate these to machine instructions
  - Use auxiliary register: $R_3 = R_1; R_1 = R_2; R_2 = R_3$
  - Use XOR-swap: $R_1 = R_1 \oplus R_2; R_2 = R_1 \oplus R_2; R_1 = R_1 \oplus R_2$
- But what about more than two registers?

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \ldots \mid R_n = R_1$

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \ldots \mid R_n = R_1$
    - Require $n - 1$ swaps: $R_1 \leftrightarrow R_2; R_2 \leftrightarrow R_3; \ldots; R_{n-1} \leftrightarrow R_n$

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \ldots \mid R_n = R_1$
  - Require $n - 1$ swaps: $R_1 \leftrightarrow R_2; R_2 \leftrightarrow R_3; \ldots; R_{n-1} \leftrightarrow R_n$
- Permutations: Consider permutation π, i.e., bijection $\{0, \ldots n\} \to \{0, \ldots n\}$

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \ldots \mid R_n = R_1$
  - Require $n - 1$ swaps: $R_1 \leftrightarrow R_2; R_2 \leftrightarrow R_3; \ldots; R_{n-1} \leftrightarrow R_n$
- Permutations: Consider permutation π, i.e., bijection $\{0, \ldots n\} \to \{0, \ldots n\}$
  - Cycle in a permutation: Sequence $p_1, \ldots, p_k$ such that $\pi(p_1) = p_2, \ldots, \pi(p_k) = p_1$, and $i \neq j \implies p_i \neq p_j$

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \ldots \mid R_n = R_1$
  - Require $n - 1$ swaps: $R_1 \leftrightarrow R_2; R_2 \leftrightarrow R_3; \ldots; R_{n-1} \leftrightarrow R_n$
- Permutations: Consider permutation π, i.e., bijection $\{0, \ldots n\} \to \{0, \ldots n\}$
  - Cycle in a permutation: Sequence $p_1, \ldots, p_k$ such that $\pi(p_1) = p_2, \ldots, \pi(p_k) = p_1$, and $i \neq j \implies p_i \neq p_j$
  - Cayley distance: $n - \#$cycles. Equals number of required swaps

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \dots \mid R_n = R_1$
  - Require $n - 1$ swaps: $R_1 \leftrightarrow R_2; R_2 \leftrightarrow R_3; \dots; R_{n-1} \leftrightarrow R_n$
- Permutations: Consider permutation π, i.e., bijection $\{0, \dots n\} \to \{0, \dots n\}$
  - Cycle in a permutation: Sequence $p_1, \dots, p_k$ such that $\pi(p_1) = p_2, \dots, \pi(p_k) = p_1$, and $i \neq j \implies p_i \neq p_j$
  - Cayley distance: $n - \#$cycles. Equals number of required swaps
    - Process each cycle separately

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \ldots \mid R_n = R_1$
  - Require $n - 1$ swaps: $R_1 \leftrightarrow R_2; R_2 \leftrightarrow R_3; \ldots; R_{n-1} \leftrightarrow R_n$
- Permutations: Consider permutation π, i.e., bijection $\{0, \ldots n\} \to \{0, \ldots n\}$
  - Cycle in a permutation: Sequence $p_1, \ldots, p_k$ such that $\pi(p_1) = p_2, \ldots, \pi(p_k) = p_1$, and $i \neq j \implies p_i \neq p_j$
  - Cayley distance: $n - \#$cycles. Equals number of required swaps
    - Process each cycle separately
- General case: Each register occurs on LHS at most once

Discussion (ctd)

- Cyclic shifts: $R_1 = R_2 \mid R_2 = R_3 \mid \ldots \mid R_n = R_1$
  - Require $n - 1$ swaps: $R_1 \leftrightarrow R_2; R_2 \leftrightarrow R_3; \ldots ; R_{n-1} \leftrightarrow R_n$
- Permutations: Consider permutation π, i.e., bijection $\{0, \ldots n\} \to \{0, \ldots n\}$
  - Cycle in a permutation: Sequence $p_1, \ldots, p_k$ such that $\pi(p_1) = p_2, \ldots, \pi(p_k) = p_1$, and $i \neq j \implies p_i \neq p_j$
  - Cayley distance: $n - \#$cycles. Equals number of required swaps
    - Process each cycle separately
- General case: Each register occurs on LHS at most once
  - Decompose into sequence of linear assignments and cyclic shifts

Interprocedural Register Allocation

- For every local variable, there is an entry in the stack frame

Interprocedural Register Allocation

- For every local variable, there is an entry in the stack frame
- Save locals to stack before call, restore after call

Interprocedural Register Allocation

- For every local variable, there is an entry in the stack frame
- Save locals to stack before call, restore after call
- Sometimes, there is hardware support for this

Interprocedural Register Allocation

- For every local variable, there is an entry in the stack frame
- Save locals to stack before call, restore after call
- Sometimes, there is hardware support for this
- Otherwise, we have to insert load and stores. We may ...

Interprocedural Register Allocation

- For every local variable, there is an entry in the stack frame
- Save locals to stack before call, restore after call
- Sometimes, there is hardware support for this
- Otherwise, we have to insert load and stores. We may ...
    - Save only registers which may actually be overwritten

Interprocedural Register Allocation

- For every local variable, there is an entry in the stack frame
- Save locals to stack before call, restore after call
- Sometimes, there is hardware support for this
- Otherwise, we have to insert load and stores. We may ...
  - Save only registers which may actually be overwritten
  - Save only registers which are live after the call

Interprocedural Register Allocation

- For every local variable, there is an entry in the stack frame
- Save locals to stack before call, restore after call
- Sometimes, there is hardware support for this
- Otherwise, we have to insert load and stores. We may ...
    - Save only registers which may actually be overwritten
    - Save only registers which are live after the call
    - May restore into different registers $\Longrightarrow$ reduction of live ranges

Table of Contents

Motivation

- Modern processors do not execute instructions one after the other

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
  - which are independent, and thus can be done in parallel for multiple instructions

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
  - which are independent, and thus can be done in parallel for multiple instructions
  - Pipelining

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
  - which are independent, and thus can be done in parallel for multiple instructions
  - Pipelining
- Hardware for executing instructions is duplicated (superscalar processors)

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
  - which are independent, and thus can be done in parallel for multiple instructions
  - Pipelining
- Hardware for executing instructions is duplicated (superscalar processors)
  - Independent instructions can be executed simultaneously

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
  - which are independent, and thus can be done in parallel for multiple instructions
  - Pipelining
- Hardware for executing instructions is duplicated (superscalar processors)
  - Independent instructions can be executed simultaneously
  - Usually combined with pipelining

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
  - which are independent, and thus can be done in parallel for multiple instructions
  - Pipelining
- Hardware for executing instructions is duplicated (superscalar processors)
  - Independent instructions can be executed simultaneously
  - Usually combined with pipelining
- Who decides what instructions to parallelize

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
  - which are independent, and thus can be done in parallel for multiple instructions
  - Pipelining
- Hardware for executing instructions is duplicated (superscalar processors)
  - Independent instructions can be executed simultaneously
  - Usually combined with pipelining
- Who decides what instructions to parallelize
  - The compiler. $\Longrightarrow$ VLIW - architectures
    - E.g., IA64, on Itanium processors

Motivation

- Modern processors do not execute instructions one after the other
- Each instruction passes multiple phases
    - which are independent, and thus can be done in parallel for multiple instructions
    - Pipelining
- Hardware for executing instructions is duplicated (superscalar processors)
    - Independent instructions can be executed simultaneously
    - Usually combined with pipelining
- Who decides what instructions to parallelize
    - The compiler. $\Longrightarrow$ VLIW - architectures
        - E.g., IA64, on Itanium processors
    - The processor (e.g. x86)
        - Compiler should arrange instructions accordingly

Pipelining

- Execute instruction in multiple phases

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor
- Idea: Keep all parts busy by having multiple instructions in the pipeline

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor
- Idea: Keep all parts busy by having multiple instructions in the pipeline
- Problem: Instructions may depend on each other

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor
- Idea: Keep all parts busy by having multiple instructions in the pipeline
- Problem: Instructions may depend on each other
  - e.g., `R2 = 0; R = R+1; R = R+R`

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor
- Idea: Keep all parts busy by having multiple instructions in the pipeline
- Problem: Instructions may depend on each other
  - e.g., `R`$_2$ `= 0; R = R+1; R = R+R`
    - execute phase of second instruction cannot start, until write-phase of first instruction completed

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor
- Idea: Keep all parts busy by having multiple instructions in the pipeline
- Problem: Instructions may depend on each other
  - e.g., `R`$_2$ `= 0; R = R+1; R = R+R`
    - execute phase of second instruction cannot start, until write-phase of first instruction completed
  - Pipeline stall.

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor
- Idea: Keep all parts busy by having multiple instructions in the pipeline
- Problem: Instructions may depend on each other
  - e.g., `R`$_2$ `= 0; R = R+1; R = R+R`
    - execute phase of second instruction cannot start, until write-phase of first instruction completed
  - Pipeline stall.
  - But compiler could have re-arranged instructions

Pipelining

- Execute instruction in multiple phases
  - e.g., fetch, decode, execute, write
  - Which are handled by different parts of the processor
- Idea: Keep all parts busy by having multiple instructions in the pipeline
- Problem: Instructions may depend on each other
  - e.g., `R`$_2$ `= 0; R = R+1; R = R+R`
    - execute phase of second instruction cannot start, until write-phase of first instruction completed
  - Pipeline stall.
  - But compiler could have re-arranged instructions
    - `R = R+1; R`$_2$`= 0; R = R+R`

Superscalar architectures

- Fetch >1 instruction per cycle.

Superscalar architectures

- Fetch >1 instruction per cycle.
- Execute them in parallel if independent

Superscalar architectures

- Fetch >1 instruction per cycle.
- Execute them in parallel if independent
- Processor checks independence

Superscalar architectures

- Fetch $>$ 1 instruction per cycle.
- Execute them in parallel if independent
- Processor checks independence
  - Out-of-order execution: Processor may re-order instructions

Superscalar architectures

- Fetch $>$ 1 instruction per cycle.
- Execute them in parallel if independent
- Processor checks independence
  - Out-of-order execution: Processor may re-order instructions
- Or compiler checks independence (VLIW)

Exam

- You may bring in two handwritten A4 sheets
- We will not ask you to write OCaml programs

Last Lecture

- Register allocation
  - by coloring interference graph
  - by going to SSA-form
- Instruction level parallelism
  - Pipelining, superscalar architectures

Observation

- These architectures are profitable if there are enough independent instructions available

Observation

- These architectures are profitable if there are enough independent instructions available
- Here:

Observation

- These architectures are profitable if there are enough independent instructions available
- Here:
  1. Re-arrange independent instructions (in basic blocks)

Observation

- These architectures are profitable if there are enough independent instructions available
- Here:
  1. Re-arrange independent instructions (in basic blocks)
  2. Increase size of basic blocks, to increase potential for parallelizing

Data dependence graph

- Consider basic block $a_1; \ldots; a_n$

Data dependence graph

- Consider basic block $a_1; \ldots; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

Data dependence graph

- Consider basic block $a_1; \ldots ; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

  read-write a_i reads register written by a_j

Data dependence graph

- Consider basic block $a_1; \ldots; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

  read-write a_i reads register written by a_j
  write-read a_i writes register read by a_j

Data dependence graph

- Consider basic block $a_1; \ldots; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

read-write	a_i reads register written by a_j
write-read	a_i writes register read by a_j
write-write	a_i and a_j both write same register

Data dependence graph

- Consider basic block $a_1; \ldots; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

  read-write a_i reads register written by a_j

  write-read a_i writes register read by a_j

  write-write a_i and a_j both write same register
- Dependence graph: Directed graph with

Data dependence graph

- Consider basic block $a_1; \ldots ; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

  read-write a_i reads register written by a_j
  write-read a_i writes register read by a_j
  write-write a_i and a_j both write same register
- Dependence graph: Directed graph with
  - $V := \{a_1, \ldots, a_n\}$

Data dependence graph

- Consider basic block $a_1; \ldots; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

  read-write a_i reads register written by a_j
  write-read a_i writes register read by a_j
  write-write a_i and a_j both write same register
- Dependence graph: Directed graph with
  - $V := \{a_1, \ldots, a_n\}$
  - $(a_i, a_j) \in E$ iff a_i and a_j are dependent

Data dependence graph

- Consider basic block $a_1; \ldots; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

  read-write a_i reads register written by a_j
  write-read a_i writes register read by a_j
  write-write a_i and a_j both write same register
- Dependence graph: Directed graph with
  - $V := \{a_1, \ldots, a_n\}$
  - $(a_i, a_j) \in E$ iff a_i and a_j are dependent
- Instructions in basic block can be reordered

Data dependence graph

- Consider basic block $a_1; \dots; a_n$
- Instructions a_i and a_j, $i < j$, are dependent, iff

  read-write a_i reads register written by a_j

  write-read a_i writes register read by a_j

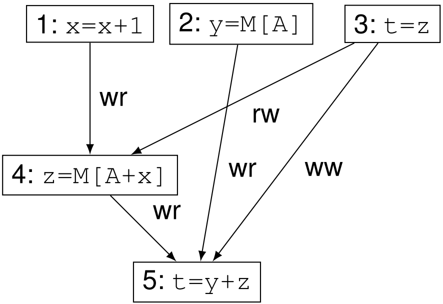
  write-write a_i and a_j both write same register
- Dependence graph: Directed graph with
  - $V := \{a_1, \dots, a_n\}$
  - $(a_i, a_j) \in E$ iff a_i and a_j are dependent
- Instructions in basic block can be reordered
  - As long as ordering respects dependence graph

Example

```
x=x+1
y=M[A]
t=z
z=M[A+x]
t=y+z
```

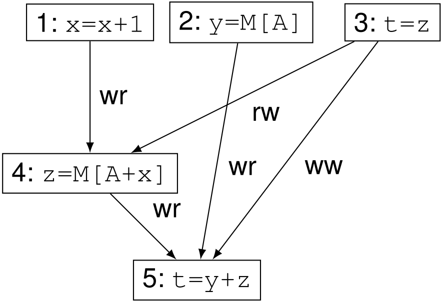
Example

```
1: x=x+1
2: y=M[A]
3: t=z
4: z=M[A+x]
5: t=y+z
```



Example

```
1: x=x+1
2: y=M[A]
3: t=z
4: z=M[A+x]
5: t=y+z
```



Possible re-ordering:

```
2: y=M[A]
1: x=x+1
3: t=z
4: z=M[A+x]
5: t=y+z
```

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture
- Assign priorities to source nodes in graph

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture
- Assign priorities to source nodes in graph
- Schedule node with highest priority first

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture
- Assign priorities to source nodes in graph
- Schedule node with highest priority first
- Heuristics for priorities

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture
- Assign priorities to source nodes in graph
- Schedule node with highest priority first
- Heuristics for priorities
  - If required resources are blocked: Lower priority

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture
- Assign priorities to source nodes in graph
- Schedule node with highest priority first
- Heuristics for priorities
  - If required resources are blocked: Lower priority
  - If dependencies not yet available: Lower priority

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture
- Assign priorities to source nodes in graph
- Schedule node with highest priority first
- Heuristics for priorities
  - If required resources are blocked: Lower priority
  - If dependencies not yet available: Lower priority
  - If node creates many new sources: Rise priority

Instruction Scheduling

- Goal: Find topological ordering that stalls pipeline as few as possible
  - Problems: Data dependencies, limited processor resources (e.g., only single floating-point unit)
  - In general: NP-hard problem
- Common heuristics: List scheduling
- While scheduling, keep track of used processor resources
  - Requires (more or less precise) model of processor architecture
- Assign priorities to source nodes in graph
- Schedule node with highest priority first
- Heuristics for priorities
  - If required resources are blocked: Lower priority
  - If dependencies not yet available: Lower priority
  - If node creates many new sources: Rise priority
  - If node lies on critical path: Rise priority

Example: Live-range splitting

- Live-range splitting helps to decrease dependencies

Example: Live-range splitting

- Live-range splitting helps to decrease dependencies
- No re-ordering possible

```
1: x=r
2: y=x+1
3: x=s
4: z=x+1
```

Example: Live-range splitting

- Live-range splitting helps to decrease dependencies
- Can be re-ordered

```
1: x₁=r
2: y=x₁+1
3: x₂=s
4: z=x₂+1
```

Example: Live-range splitting

- Live-range splitting helps to decrease dependencies
- Can be re-ordered

```
1:  x₁=r
2:  y=x₁+1
3:  x₂=s
4:  z=x₂+1
```

- Re-ordering

```
1:  x₁=r
3:  x₂=s
2:  y=x₁+1
4:  z=x₂+1
```

Example: Live-range splitting

- Live-range splitting helps to decrease dependencies
- Can be re-ordered

```
1: x₁=r
2: y=x₁+1
3: x₂=s
4: z=x₂+1
```

- Re-ordering

```
1: x₁=r
3: x₂=s
2: y=x₁+1
4: z=x₂+1
```

- Some processors do that dynamically

$\Longrightarrow$ Register renaming

Loop unrolling

- Consider the example

```
short M [...];
for (i=0;i<n;++i) {
  M[i] = 0
}
```

Loop unrolling

- Consider the example

```
short M [...];
for (i=0;i<n;++i) {
  M[i] = 0
}
```

- On 32 bit architecture: Writing 16 bit words

Loop unrolling

- Consider the example

```
short M [...];
for (i=0;i<n;++i) {
  M[i] = 0
}
```

- On 32 bit architecture: Writing 16 bit words
  - Expensive!

Loop unrolling

- Consider the example

```
short M [...];
for (i=0;i<n;++i) {
  M[i] = 0
}
```

- On 32 bit architecture: Writing 16 bit words
    - Expensive!
- Consider unrolled loop (unroll factor 2)

```
short M [...];
for (i=0;i+1<n;) {
  M[i] = 0
  i=i+1
  M[i] = 0
  i=i+1
}
if (i<n) {M[i]=0; i=i+1}  // For odd n
```

Loop unrolling

- Consider the example

```
short M [...];
for (i=0;i<n;++i) {
  M[i] = 0
}
```

- On 32 bit architecture: Writing 16 bit words
  - Expensive!
- Consider unrolled loop (unroll factor 2)

```
short M [...];
for (i=0;i+1<n;) {
  M[i] = 0
  i=i+1
  M[i] = 0
  i=i+1
}
if (i<n) {M[i]=0; i=i+1}  // For odd n
```

- Loop body can now easily be optimized, e.g., by peephole optimization

Loop unrolling

- Consider the example

```
short M [...];
for (i=0;i<n;++i) {
  M[i] = 0
}
```

- On 32 bit architecture: Writing 16 bit words
  - Expensive!
- Consider unrolled loop (unroll factor 2)

```
short M [...];
for (i=0;i+1<n;i=i+2) {
   (int)M[i] = 0
}
if (i<n) {M[i]=0; i=i+1}  // For odd n
```

- Loop body can now easily be optimized, e.g., by peephole optimization

Discussion

- Loop unrolling creates bigger basic blocks

Discussion

- Loop unrolling creates bigger basic blocks
- Which open more opportunities for parallelization

Discussion

- Loop unrolling creates bigger basic blocks
- Which open more opportunities for parallelization
- Quick demo with gcc -O2 -funroll-loops

Loop fusion

- Fuse together two successive loops

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- **for** (...) {c_1}; **for** (...) {c_2} $\mapsto$ **for** (...) {c_1;c_2}

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- `for (...) {c1}; for (...) {c2}` $\mapsto$ `for (...) {c1;c2}`
- In general:

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- `for (...) {c1}; for (...) {c2}` $\mapsto$ `for (...) {c1;c2}`
- In general:
  - ith iteration of c_1 must not read data, that is written in $<i$th iteration of c_2

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- `for (...) {c1}; for (...) {c2}` $\mapsto$ `for (...) {c1;c2}`
- In general:
  - ith iteration of c_1 must not read data, that is written in $<$ ith iteration of c_2
  - ith iteration of c_2 must not read data, that is written in $>$ ith iteration of c_1

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- `for (...) {c1}; for (...) {c2}` $\mapsto$ `for (...) {c1;c2}`
- In general:
  - ith iteration of c_1 must not read data, that is written in $<$ ith iteration of c_2
  - ith iteration of c_2 must not read data, that is written in $>$ ith iteration of c_1
- Heuristics

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- `for (...) {c1}; for (...) {c2}` $\mapsto$ `for (...) {c1;c2}`
- In general:
  - ith iteration of c_1 must not read data, that is written in $<$ ith iteration of c_2
  - ith iteration of c_2 must not read data, that is written in $>$ ith iteration of c_1
- Heuristics
  - Data written to disjoint places

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- `for (...) {c1}; for (...) {c2}` $\mapsto$ `for (...) {c1;c2}`
- In general:
  - ith iteration of c_1 must not read data, that is written in $<$ ith iteration of c_2
  - ith iteration of c_2 must not read data, that is written in $>$ ith iteration of c_1
- Heuristics
  - Data written to disjoint places
    - E.g., different, statically allocated arrays

Loop fusion

- Fuse together two successive loops
  - With the same iteration scheme
  - That are not data-dependent
- `for (...) {`c_1`}; for (...) {`c_2`}` $\mapsto$ `for (...) {`c_1`;`c_2`}`
- In general:
  - ith iteration of c_1 must not read data, that is written in $<$ ith iteration of c_2
  - ith iteration of c_2 must not read data, that is written in $>$ ith iteration of c_1
- Heuristics
  - Data written to disjoint places
    - E.g., different, statically allocated arrays
  - More sophisticated analyses, e.g., based on integer linear programming

Example

- Consider the following loop, assume A, B, C, D are guaranteed to be different

```
for (i=0;i<n;++i) C[i] = A[i] + B[i];
for (i=0;i<n;++i) D[i] = A[i] - B[i];
```

Example

- Consider the following loop, assume *A*, *B*, *C*, *D* are guaranteed to be different

```
for (i=0;i<n;++i) C[i] = A[i] + B[i];
for (i=0;i<n;++i) D[i] = A[i] - B[i];
```

- Loop fusion yields

```
for (i=0;i<n;++i) {
  C[i] = A[i] + B[i];
  D[i] = A[i] - B[i]}
```

Example

- Consider the following loop, assume *A*, *B*, *C*, *D* are guaranteed to be different

```
for (i=0;i<n;++i) C[i] = A[i] + B[i];
for (i=0;i<n;++i) D[i] = A[i] - B[i];
```

- Loop fusion yields

```
for (i=0;i<n;++i) {
  C[i] = A[i] + B[i];
  D[i] = A[i] - B[i]}
```

- Which may be further optimized to

```
for (i=0;i<n;++i) {
  R1 = A[i]; R2 = B[i];
  C[i] = R1 + R2;
  D[i] = R1 - R2}
```

Warning

- The opposite direction, loop fission, splits one loop into two

Warning

- The opposite direction, loop fission, splits one loop into two
- May be profitable for large loops

Warning

- The opposite direction, loop fission, splits one loop into two
- May be profitable for large loops
  - Smaller loops may fit into cache entirely

Warning

- The opposite direction, loop fission, splits one loop into two
- May be profitable for large loops
  - Smaller loops may fit into cache entirely
  - Accessed memory more local, better cache behavior

Table of Contents

Motivation

- Aligning of data

Motivation

- Aligning of data
- Cache-aware data access

Motivation

- Aligning of data
- Cache-aware data access
- Reduction of allocation/deallocation cost

Alignment of data

- Processor usually loads 32/64 bit words from memory

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8
  - Read from odd addresses needs to be split

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8
  - Read from odd addresses needs to be split
  - Expensive

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8
  - Read from odd addresses needs to be split
  - Expensive
- So compilers can align data in memory accordingly

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8
  - Read from odd addresses needs to be split
  - Expensive
- So compilers can align data in memory accordingly
  - Data on stack (parameters, local variables)

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8
  - Read from odd addresses needs to be split
  - Expensive
- So compilers can align data in memory accordingly
  - Data on stack (parameters, local variables)
  - Code (labels, functions, loop-heads)

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8
  - Read from odd addresses needs to be split
  - Expensive
- So compilers can align data in memory accordingly
  - Data on stack (parameters, local variables)
  - Code (labels, functions, loop-heads)
  - Layout of structures

Alignment of data

- Processor usually loads 32/64 bit words from memory
  - But only from address which is multiple of 4/8
  - Read from odd addresses needs to be split
  - Expensive
- So compilers can align data in memory accordingly
  - Data on stack (parameters, local variables)
  - Code (labels, functions, loop-heads)
  - Layout of structures
- At the cost of wasting more memory

Cache-aware data access

- Load instruction loads whole cache-line

Cache-aware data access

- Load instruction loads whole cache-line
- Subsequent loads within the same cache-line much faster

Cache-aware data access

- Load instruction loads whole cache-line
- Subsequent loads within the same cache-line much faster
- Re-arrange memory accesses accordingly

Cache-aware data access

- Load instruction loads whole cache-line
- Subsequent loads within the same cache-line much faster
- Re-arrange memory accesses accordingly
- Important case: Multi-dimensional arrays

Cache-aware data access

- Load instruction loads whole cache-line
- Subsequent loads within the same cache-line much faster
- Re-arrange memory accesses accordingly
- Important case: Multi-dimensional arrays
    - Iteration should iterate according to memory layout

Example

- Array `A[N][M]`

Example

- Array `A[N][M]`
- Assume layout: `&(A[i,j]) = i + j*N`

Example

- Array `A[N][M]`
- Assume layout: `&(A[i,j]) = i + j*N`
- `for (i=0;i<N;++i) for (j=0;j<M;++j) x=x+A[i,j]`

Example

- Array `A[N][M]`
- Assume layout: `&(A[i,j]) = i + j*N`
- `for (i=0;i<N;++i) for (j=0;j<M;++j) x=x+A[i,j]`
  - Memory accesses:
    $A+0+0N, A+0+1N, A+0+2N, ..., A+1+0N, A+1+1N, ...$

Example

- Array `A[N][M]`
- Assume layout: `&(A[i,j]) = i + j*N`
- `for (i=0;i<N;++i) for (j=0;j<M;++j) x=x+A[i,j]`
  - Memory accesses: $A + 0 + 0N, A + 0 + 1N, A + 0 + 2N, ..., A + 1 + 0N, A + 1 + 1N, ...$
  - Bad locality, when arriving at $A + 1 + 0N$, cache-line loaded on $A + 0 + 0N$ probably already overwritten

Example

- Array `A[N][M]`
- Assume layout: `&(A[i,j]) = i + j*N`
- `for (i=0;i<N;++i) for (j=0;j<M;++j) x=x+A[i,j]`
  - Memory accesses:
    $A + 0 + 0N, A + 0 + 1N, A + 0 + 2N, \ldots, A + 1 + 0N, A + 1 + 1N, \ldots$
  - Bad locality, when arriving at $A + 1 + 0N$, cache-line loaded on $A + 0 + 0N$ probably already overwritten
- Better: `for (j=0;j<M;++j) for (i=0;i<N;++i) x=x+A[i,j]`

Example

- Array `A[N][M]`
- Assume layout: `&(A[i,j]) = i + j*N`
- **`for`** `(i=0;i<N;++i)` **`for`** `(j=0;j<M;++j) x=x+A[i,j]`
  - Memory accesses: $A + 0 + 0N, A + 0 + 1N, A + 0 + 2N, ..., A + 1 + 0N, A + 1 + 1N, ...$
  - Bad locality, when arriving at $A + 1 + 0N$, cache-line loaded on $A + 0 + 0N$ probably already overwritten
- Better: **`for`** `(j=0;j<M;++j)` **`for`** `(i=0;i<N;++i) x=x+A[i,j]`
  - Memory accesses: $A + 0 + 0N, A + 1 + 0N, ..., A + 0 + 1N, A + 1 + 1N, ...$

Example

- Array `A[N][M]`
- Assume layout: `&(A[i,j]) = i + j*N`
- `for (i=0;i<N;++i) for (j=0;j<M;++j) x=x+A[i,j]`
  - Memory accesses: $A+0+0N, A+0+1N, A+0+2N, ..., A+1+0N, A+1+1N, ...$
  - Bad locality, when arriving at $A+1+0N$, cache-line loaded on $A+0+0N$ probably already overwritten
- Better: `for (j=0;j<M;++j) for (i=0;i<N;++i) x=x+A[i,j]`
  - Memory accesses: $A+0+0N, A+1+0N, ..., A+0+1N, A+1+1N, ...$
  - Good locality, $A+1+0N$ probably already in cache

Loop interchange

- Swap inner and outer loop

Loop interchange

- Swap inner and outer loop
  - If they iterate over multi-dimensional array ...

Loop interchange

- Swap inner and outer loop
  - If they iterate over multi-dimensional array ...
  - ... in wrong order

Loop interchange

- Swap inner and outer loop
  - If they iterate over multi-dimensional array ...
  - ... in wrong order
  - And loop iterations are sufficiently independent

Loop interchange

- Swap inner and outer loop
  - If they iterate over multi-dimensional array ...
  - ... in wrong order
  - And loop iterations are sufficiently independent
    - Iteration for index i, j, must only depend on iterations $\leq i, \leq j$

Loop interchange

- Swap inner and outer loop
  - If they iterate over multi-dimensional array ...
  - ... in wrong order
  - And loop iterations are sufficiently independent
    - Iteration for index i, j, must only depend on iterations $\leq i, \leq j$
    - Illustration on board!

Loop interchange

- Swap inner and outer loop
    - If they iterate over multi-dimensional array ...
    - ... in wrong order
    - And loop iterations are sufficiently independent
        - Iteration for index i, j, must only depend on iterations $\leq i, \leq j$
        - Illustration on board!
- The required dependency analysis is automatable

Loop interchange

- Swap inner and outer loop
  - If they iterate over multi-dimensional array ...
  - ... in wrong order
  - And loop iterations are sufficiently independent
    - Iteration for index i, j, must only depend on iterations $\leq i, \leq j$
    - Illustration on board!
- The required dependency analysis is automatable
  - To some extend for arrays

Loop interchange

- Swap inner and outer loop
  - If they iterate over multi-dimensional array ...
  - ... in wrong order
  - And loop iterations are sufficiently independent
    - Iteration for index i, j, must only depend on iterations $\leq i, \leq j$
    - Illustration on board!
- The required dependency analysis is automatable
  - To some extend for arrays
  - Not so much for more complex structures

Organizing data-structures block-wise

- Warning: No automation in general

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior
  - And extra memory for link-pointers

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior
  - And extra memory for link-pointers
- Alternative: Array-List

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior
  - And extra memory for link-pointers
- Alternative: Array-List
  - Keep list in array, store index of last element

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior
  - And extra memory for link-pointers
- Alternative: Array-List
  - Keep list in array, store index of last element
  - If array overflows: Double the size of the array

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior
  - And extra memory for link-pointers
- Alternative: Array-List
  - Keep list in array, store index of last element
  - If array overflows: Double the size of the array
  - If array less than quarter-full: Halve the size of the array

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior
  - And extra memory for link-pointers
- Alternative: Array-List
  - Keep list in array, store index of last element
  - If array overflows: Double the size of the array
  - If array less than quarter-full: Halve the size of the array
  - This adds amortized constant extra cost

Organizing data-structures block-wise

- Warning: No automation in general
- Example: Stack-data structure with push, pop
  - Possible implementation: Linked list
  - Disadvantage: Data items distributed over memory
  - Bad cache behavior
  - And extra memory for link-pointers
- Alternative: Array-List
  - Keep list in array, store index of last element
  - If array overflows: Double the size of the array
  - If array less than quarter-full: Halve the size of the array
  - This adds amortized constant extra cost
  - But makes cache-locality much better

Moving heap-allocated blocks to the stack

- Idea: Allocate block of memory on stack, instead of heap

Moving heap-allocated blocks to the stack

- Idea: Allocate block of memory on stack, instead of heap
  - If pointers to this block cannot escape the current stack frame

Moving heap-allocated blocks to the stack

- Idea: Allocate block of memory on stack, instead of heap
  - If pointers to this block cannot escape the current stack frame
  - Important for languages like Java, where almost everything is allocated on heap

Abstract example

```
int do_computation(...) {
  AuxData aux = new AuxData ()
  ...
  return ...
}
```

Abstract example

```
int do_computation(...) {
  AuxData aux = new AuxData ()
  ...
  return ...
}
```

- If no pointer to aux is returned or stored in global memory ...

Abstract example

```
int do_computation(...) {
  AuxData aux = new AuxData ()
  ...
  return ...
}
```

- If no pointer to aux is returned or stored in global memory ...
- ... aux can be allocated on method's stack-frame

Example

- Recall our simple pointer-language. *Ret* is global variable.

```
1: x=new()
2: y=new()
  x[A] = y
  z=x[A]
  Ret = z
```

Example

- Recall our simple pointer-language. *Ret* is global variable.

```
1: x=new()
2: y=new()
   x[A] = y
   z=x[A]
   Ret = z
```

- Allocation at 1 may not escape

Example

- Recall our simple pointer-language. *Ret* is global variable.

```
1: x=new()
2: y=new()
  x[A] = y
  z=x[A]
  Ret = z
```

- Allocation at 1 may not escape
- Thus we may do the allocation on the stack

In general

- Memory block may escape, which is

In general

- Memory block may escape, which is
    - Assigned to global variable

In general

- Memory block may escape, which is
    - Assigned to global variable
    - Reachable from global variable

In general

- Memory block may escape, which is
    - Assigned to global variable
    - Reachable from global variable
- Forward may analysis. Same as pointer-analysis

In general

- Memory block may escape, which is
  - Assigned to global variable
  - Reachable from global variable
- Forward may analysis. Same as pointer-analysis
  - Identify memory blocks with allocation sites

In general

- Memory block may escape, which is
  - Assigned to global variable
  - Reachable from global variable
- Forward may analysis. Same as pointer-analysis
  - Identify memory blocks with allocation sites
  - Analyze where variables/blocks may point to

In general

- Memory block may escape, which is
  - Assigned to global variable
  - Reachable from global variable
- Forward may analysis. Same as pointer-analysis
  - Identify memory blocks with allocation sites
  - Analyze where variables/blocks may point to
  - If global variable/unknown memory block may point to block: Possible escape

Applying the optimization, heuristics

- Only makes sense for small blocks

Applying the optimization, heuristics

- Only makes sense for small blocks
- That are allocated only once

Applying the optimization, heuristics

- Only makes sense for small blocks
- That are allocated only once
  - e.g., not inside loop

Handling procedures more precisely

- Require interprocedural points-to analysis

Handling procedures more precisely

- Require interprocedural points-to analysis
  - Expensive

Handling procedures more precisely

- Require interprocedural points-to analysis
  - Expensive
  - We do not always know whole program

Handling procedures more precisely

- Require interprocedural points-to analysis
  - Expensive
  - We do not always know whole program
    - E.g. Java loads classes at runtime

Handling procedures more precisely

- Require interprocedural points-to analysis
  - Expensive
  - We do not always know whole program
    - E.g. Java loads classes at runtime
- In worst case: Assume everything visible to called procedure may escape

Handling procedures more precisely

- Require interprocedural points-to analysis
  - Expensive
  - We do not always know whole program
    - E.g. Java loads classes at runtime
- In worst case: Assume everything visible to called procedure may escape
  - Which is consistent with parameter passing by global variables and previous analysis

Wrap-Up

- Several optimizations that exploit hardware utilization

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
    1. Restructuring of procedures/loops for better cache-behaviour

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission
     - Tail-recursion/inlining, stack-allocation

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission
     - Tail-recursion/inlining, stack-allocation
  2. Basic-block optimizations, to exploit instruction-level parallelism

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission
     - Tail-recursion/inlining, stack-allocation
  2. Basic-block optimizations, to exploit instruction-level parallelism
     - Live-range splitting

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission
     - Tail-recursion/inlining, stack-allocation
  2. Basic-block optimizations, to exploit instruction-level parallelism
     - Live-range splitting
     - Instruction scheduling

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission
     - Tail-recursion/inlining, stack-allocation
  2. Basic-block optimizations, to exploit instruction-level parallelism
     - Live-range splitting
     - Instruction scheduling
     - Loop unrolling, fusion

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission
     - Tail-recursion/inlining, stack-allocation
  2. Basic-block optimizations, to exploit instruction-level parallelism
     - Live-range splitting
     - Instruction scheduling
     - Loop unrolling, fusion
  3. Then register allocation

Wrap-Up

- Several optimizations that exploit hardware utilization
- A meaningful ordering
  1. Restructuring of procedures/loops for better cache-behaviour
     - Loop interchange, fission
     - Tail-recursion/inlining, stack-allocation
  2. Basic-block optimizations, to exploit instruction-level parallelism
     - Live-range splitting
     - Instruction scheduling
     - Loop unrolling, fusion
  3. Then register allocation
  4. And finally peephole optimization + instruction selection

Table of Contents

Last Lecture

- Optimizations to re-arrange memory access wrt. cache
  - Loop interchange
  - Lists vs. array-list
- Wrap-Up: Optimizations targeted towards features of hardware
- Started with functional languages

Functional language

- We consider simple functional language

```
prg ::= let rec f1 = e1 | ... | f_n = e_n in e
e ::= b | c | x | f_i | op | e e | fn x. e
    | let x=e in e
    | match e with p1 => e1 | ... | p_n => e_n
p ::= b | c x1 ... xn
```

Functional language

- We consider simple functional language

```
prg ::= let rec f₁ = e₁ | ... | f_n = e_n in e
e ::= b | c | x | f_i | op | e e | fn x. e
    | let x=e in e
    | match e with p₁ => e₁ | ... | p_n => e_n
p ::= b | c x₁ ... xn
```

- where
  - *b* is primitive constant
  - *c* is constructor
  - *x* is variable
  - f_i is recursive function
  - *op* is primitive operation

Table of Contents

Semantics

- Values $b, c\ v_1 \ldots v_n, \mathit{fn}\ x.\ e$ (Convention: v denotes values)

Semantics

- Values $b, c\ v_1 \ldots v_n, \mathit{fn}\ x.\ e$ (Convention: v denotes values)
- Goal of semantics: Evaluate main expression to value

Semantics

- Values $b, c\ v_1 \dots v_n$, $fn\ x.\ e$ (Convention: v denotes values)
- Goal of semantics: Evaluate main expression to value
- Done by the following rules

$$[rec]\,\frac{\texttt{let rec f\_i = e\_i}}{f_i \to e_i}$$

$$[op]\,\frac{-}{op\ b_1 \dots b_n \to [\![op]\!](b_1, \dots, b_n)}$$

$$[app1]\,\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad [app2]\,\frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

$$[\beta{-}red]\,\frac{-}{(fn\ x.\ e)\ v \to e[x \mapsto v]}$$

$$[match1]\,\frac{e \to e'}{match\ e\ with\ \dots \to match\ e'\ with\ \dots}$$

$$[match2]\,\frac{-}{match\ v\ with\ \dots \to e_i\sigma} \quad (*)$$

$$[app{-}op]\,\frac{e_k \to e_k'}{op\ v_1\ \dots\ v_{k-1}\ e_k\ \dots e_n \to op\ v_1\ \dots\ v_{k-1}\ e_k'\ \dots e_n}$$

- where $let\ x = e_1\ in\ e_2$ is syntax for $(fn\ x.\ e_2)\ e_1$
- (*): $p_i{=}{>}e_i$ is the first pattern with $p_i\sigma = v$

Semantics

- Eager evaluation

Semantics

- Eager evaluation
  - Arguments are evaluated before function is called

Semantics

- Eager evaluation
  - Arguments are evaluated before function is called
- No types: Evaluation of badly-typed program just gets stuck

Semantics

- Eager evaluation
  - Arguments are evaluated before function is called
- No types: Evaluation of badly-typed program just gets stuck
  - Example: `match 5 with True => ... | False => ...`

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

fac 2
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(fn x. ...) 2
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

match 2 with ...
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 (fac (2-1))
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 ((fn x. ...) (2-1))
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 ((fn x. ...) 1)
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 (match 1 with ...)
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 ((*) 1 (fac (1-1)))
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 ((*) 1 ((fn x. ...) (1-1)))
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 ((*) 1 ((fn x. ...) 0))
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 ((*) 1 (match 0 with ...))
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 ((*) 1 1)
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

(*) 2 1
```

Example

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in fac 2

2
```

Lazy evaluation

- Evaluate arguments only when needed, as far as needed

Lazy evaluation

- Evaluate arguments only when needed, as far as needed
  - I.e., on match or built-in function call

$$[\mathit{rec}]\frac{\texttt{let rec f\_i = e\_i}}{f_i \rightarrow e_i} \qquad [\mathit{op}]\frac{-}{\mathit{op}\ b_1 \ldots b_n \rightarrow [\![\mathit{op}]\!](b_1, \ldots, b_n)}$$

$$[\mathit{app}1]\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad [\beta\!-\!\mathit{red}]\frac{-}{(\mathit{fn}\ x.\ e_1)\ e_2 \rightarrow e_1[x \mapsto e_2]}$$

$$[\mathit{match}1]\frac{e \rightarrow e'}{\mathit{match}\ e\ \mathit{with}\ \ldots \rightarrow \mathit{match}\ e'\ \mathit{with}\ \ldots}$$

$$[\mathit{match}2]\frac{-}{\mathit{match}\ c\ \hat{e}_1 \ldots \hat{e}_k\ \mathit{with}\ \ldots \rightarrow e_i\sigma} \quad (*)$$

$$[\mathit{match}3]\frac{-}{\mathit{match}\ b\ \mathit{with}\ \ldots \rightarrow e_i\sigma} \quad (*)$$

$$[\mathit{app}\!-\!\mathit{op}]\frac{e_k \rightarrow e_k'}{\mathit{op}\ v_1\ \ldots\ v_{k-1}\ e_k\ \ldots e_n \rightarrow \mathit{op}\ v_1\ \ldots\ v_{k-1}\ e_k'\ \ldots e_n}$$

Lazy evaluation

- Evaluate arguments only when needed, as far as needed
  - I.e., on match or built-in function call

$$[rec]\,\frac{\texttt{let rec f\_i = e\_i}}{f_i \to e_i} \qquad [op]\,\frac{-}{op\ b_1 \dots b_n \to [\![op]\!](b_1, \dots, b_n)}$$

$$[app1]\,\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad [\beta{-}red]\,\frac{-}{(fn\ x.\ e_1)\ e_2 \to e_1[x \mapsto e_2]}$$

$$[match1]\,\frac{e \to e'}{match\ e\ with\ \dots \to match\ e'\ with\ \dots}$$

$$[match2]\,\frac{-}{match\ c\ \hat{e}_1 \dots \hat{e}_k\ with\ \dots \to e_i\sigma} \quad (*)$$

$$[match3]\,\frac{-}{match\ b\ with\ \dots \to e_i\sigma} \quad (*)$$

$$[app{-}op]\,\frac{e_k \to e_k'}{op\ v_1\ \dots\ v_{k-1}\ e_k\ \dots e_n \to op\ v_1\ \dots\ v_{k-1}\ e_k'\ \dots e_n}$$

- Note: Only simple patterns allowed in match

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)

(fac 2)
```

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)

((fn x. ...) 2)
```

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)

(match 2 with ...)
```

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)

((*) 2 (fac (2-1)))
```

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)

((*) 2 ((fn x. ...) (2-1)))
```

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)

((*) 2 (match (2-1) with ...))
```

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)
```

```
((*) 2 (match 1 with ...))
```

Example (lazy)

```
let
  rec fac = fn x. match x with
    0 => 1
  | x => x * fac (x-1)
in (fac 2)

and so on ...
```

Eager vs. Lazy

- Eager: Argument evaluated before function call
- Lazy: Function call before argument
    - Argument of match only until constructor is at top
        - Weak head normal form
    - Arguments of primitive operator: Completely

Optimization Plan

- Optimize on functional level

Optimization Plan

- Optimize on functional level
- Translate to imperative language/IR

Optimization Plan

- Optimize on functional level
- Translate to imperative language/IR
- Use optimizations for imperative code

Optimization Plan

- Optimize on functional level
- Translate to imperative language/IR
- Use optimizations for imperative code
- Now: Optimizations on functional level

Table of Contents

Simple optimizations

- Idea: Move some evaluation from run-time to compile-time

Simple optimizations

- Idea: Move some evaluation from run-time to compile-time
- Function-application to let

$\texttt{(fn x. e}_1\texttt{) e}_2\texttt{ --> let x=e}_2\texttt{ in e}_1$

Simple optimizations

- Idea: Move some evaluation from run-time to compile-time
- Function-application to let

  ```
  (fn x. e₁) e₂ --> let x=e₂ in e₁
  ```

- Matches, where part of the pattern is already known

  ```
  match c e₁ ... e_n with ... c x₁ ... xₙ => e
  > let x₁=e₁; ...; xₙ=e_n in e
  ```

Simple optimizations

- Idea: Move some evaluation from run-time to compile-time
- Function-application to let

  ```
  (fn x. e₁) e₂ --> let x=e₂ in e₁
  ```

- Matches, where part of the pattern is already known

  ```
  match c e₁ ... e_n with ... c x₁ ... xₙ => e
  > let x₁=e₁; ...; xₙ=e_n in e
  ```

- Let-reduction

  ```
  let x=e₁ in e --> e[x↦e₁]
  ```

Substitution

- Beware of name-capture

```
let x = 1 in
let f = fn y. x+y in
let x = 4 in
  f x
```

Substitution

- Beware of name-capture

```
let x = 1 in
let f = fn y. x+y in
let x = 4 in
  f x
```

  - Consider reduction of $f = \ldots$.

Substitution

- Beware of name-capture

```
let x = 1 in
let f = fn y. x+y in
let x = 4 in
  f x
```

  - Consider reduction of $f = \ldots$.
- α-conversion: (Consistent) renaming of (bound) variables does not change meaning of program

Substitution

- Beware of name-capture

```
let x = 1 in
let f = fn y. x+y in
let x = 4 in
  f x
```

  - Consider reduction of $f = \ldots$.
- α-conversion: (Consistent) renaming of (bound) variables does not change meaning of program
- Convention: Substitution uses α-conversion to avoid name-capture

Substitution

- Beware of name-capture

```
let x = 1 in
let f = fn y. x+y in
let x = 4 in
  f x
```

  - Consider reduction of $f = \dots$.
- α-conversion: (Consistent) renaming of (bound) variables does not change meaning of program
- Convention: Substitution uses α-conversion to avoid name-capture
  - Here: Convert `let x=4 in f x` to `let x₁=4 in f x₁`

Termination issues

- Let-reduction may change semantics

```
let rec f = fn x. 1 + f x in
let _ = f 0 in
  42
```

Termination issues

- Let-reduction may change semantics

```
let rec f = fn x. 1 + f x in
let _ = f 0 in
  42
```

  - This program does not terminate

Termination issues

- Let-reduction may change semantics

```
let rec f = fn x. 1 + f x in
let _ = f 0 in
  42
```

  - This program does not terminate
  - But, applying let-reduction, we get

```
let rec f = fn x. 1 + f x in
  42
```

Termination issues

- Let-reduction may change semantics

```
let rec f = fn x. 1 + f x in
let _ = f 0 in
  42
```

  - This program does not terminate
  - But, applying let-reduction, we get

```
let rec f = fn x. 1 + f x in
  42
```

  - which returns 42

Termination issues

- Let-reduction may change semantics

```
let rec f = fn x. 1 + f x in
let _ = f 0 in
  42
```

  - This program does not terminate
  - But, applying let-reduction, we get

    ```
    let rec f = fn x. 1 + f x in
      42
    ```

  - which returns 42
- For eager evaluation, non-terminating programs may be transformed to terminating ones

Termination issues

- Let-reduction may change semantics

```
let rec f = fn x. 1 + f x in
let _ = f 0 in
  42
```

  - This program does not terminate
  - But, applying let-reduction, we get

```
let rec f = fn x. 1 + f x in
  42
```

  - which returns 42
- For eager evaluation, non-terminating programs may be transformed to terminating ones
- For lazy evaluation, semantics is preserved

Side-effects

- Languages like SML/OCaml/F# have side-effects

Side-effects

- Languages like SML/OCaml/F# have side-effects
- Side-effecting expressions must not be let-reduced

```
let _ = print "Hello"
in ()
```

Application of let-reduction

- May make program less efficient

Application of let-reduction

- May make program less efficient
  - Re-computing values instead of storing them in variable
    ```
    let x=expensive-op in x+x
    ```

Application of let-reduction

- May make program less efficient
  - Re-computing values instead of storing them in variable
    `let x=expensive-op in x+x`
- May blow up program code exponentially
  `let x = x+x in let x = x+x in ... in x`

Application of let-reduction

- May make program less efficient
  - Re-computing values instead of storing them in variable
    `let x=expensive-op in x+x`
- May blow up program code exponentially
  `let x = x+x in let x = x+x in ... in x`
- Heuristics for application: reduce `let` x_1=e_1`in e`

Application of let-reduction

- May make program less efficient
  - Re-computing values instead of storing them in variable
    `let x=expensive-op in x+x`
- May blow up program code exponentially
  `let x = x+x in let x = x+x in ... in x`
- Heuristics for application: reduce `let` x_1=e_1`in e`
  - if e_1 is a variable (or constant)

Application of let-reduction

- May make program less efficient
  - Re-computing values instead of storing them in variable
    `let x=expensive-op in x+x`
- May blow up program code exponentially
  `let x = x+x in let x = x+x in ... in x`
- Heuristics for application: reduce `let x₁=e₁in e`
  - if e_1 is a variable (or constant)
  - if x_1 does not occur in e

Application of let-reduction

- May make program less efficient
  - Re-computing values instead of storing them in variable
    `let x=expensive-op in x+x`
- May blow up program code exponentially
  `let x = x+x in let x = x+x in ... in x`
- Heuristics for application: reduce `let` x_1=e_1`in e`
  - if e_1 is a variable (or constant)
  - if x_1 does not occur in e
  - if x_1 occurs exactly once in e

More transformations

- Valid for programs (fragments) with no side-effects

```
(let x=e in e1) e2 --> let x=e in e1 e2
  // Renaming x to avoid name capture

let x1=e1 in let x2=e2 in e
--> let x2=e2 in let x1=e1 in e
  // If x1 not free in e2
  // Renaming x2 to avoid name capture

let x1 = (let x2=e2 in e1) in e
--> let x2=e2 in let x1=e1 in e
  // Renaming x2 to avoid name capture
```

More transformations

- Valid for programs (fragments) with no side-effects

```
(let x=e in e1) e2 --> let x=e in e1 e2
  // Renaming x to avoid name capture

let x1=e1 in let x2=e2 in e
--> let x2=e2 in let x1=e1 in e
  // If x1 not free in e2
  // Renaming x2 to avoid name capture

let x1 = (let x2=e2 in e1) in e
--> let x2=e2 in let x1=e1 in e
  // Renaming x2 to avoid name capture
```

- May open potential for other optimizations

Inlining

- Consider program `let f=fn x.` e_1 `in e`

Inlining

- Consider program `let f=fn x.` e_1 `in e`
- Inside *e*, replace `f` e_2 by `let x=`e_2`in` e_1

Inlining

- Consider program `let f=fn x.` e_1 `in e`
- Inside *e*, replace `f` e_2 by `let x=`e_2`in` e_1
  - Goal: Save overhead for function call

Inlining

- Consider program `let f=fn x.` e_1 `in e`
- Inside *e*, replace `f` e_2 by `let x=`e_2`in` e_1
  - Goal: Save overhead for function call
  - Warning: May blow up the code

Example

```
let fmax = fn f. fn x. fn y.
  if x>y then f x else f y in
let max = fmax (fn x. x) in
  ...
```

Example

```
let fmax = fn f. fn x. fn y.
  if x>y then f x else f y in
let max = (let f = (fn x. x) in
  fn x. fn y. if x>y then f x else f y) in
  ...
```

(inlined fmax)

Example

```
let fmax = fn f. fn x. fn y.
  if x>y then f x else f y in
let max = (let f = (fn x. x) in
  fn x. fn y. if x>y then let x=x in x else let x=y in x) i
  ...
```

(inlined f)

Example

```
let fmax = fn f. fn x. fn y.
  if x>y then f x else f y in
let max = (
  fn x. fn y. if x>y then x else y) in
  ...
```

(Let-reduction for single-var expressions and unused variables)

Note

- Inlining can be seen as special case of let-reduction

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects
  - Only inlining terms of form `fn x. e`, which are not evaluated, unless applied to an argument

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects
  - Only inlining terms of form `fn x. e`, which are not evaluated, unless applied to an argument
- In untyped languages (e.g., LISP), the inlining optimization may not terminate

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects
  - Only inlining terms of form `fn x. e`, which are not evaluated, unless applied to an argument
- In untyped languages (e.g., LISP), the inlining optimization may not terminate

```
let w = fn f. fn y. f (y f y) in
let fix = fn f. w f w
```

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects
  - Only inlining terms of form `fn x. e`, which are not evaluated, unless applied to an argument
- In untyped languages (e.g., LISP), the inlining optimization may not terminate

```
let w = fn f. fn y. f (y f y) in
let fix = fn f. let f=f in let y=w in f (y f y)
```

(Inlined w)

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects
  - Only inlining terms of form `fn x. e`, which are not evaluated, unless applied to an argument
- In untyped languages (e.g., LISP), the inlining optimization may not terminate

```
let w = fn f. fn y. f (y f y) in
let fix = fn f. f (w f w)
```

((Safe) let-reduction (copy variables))

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects
    - Only inlining terms of form `fn x. e`, which are not evaluated, unless applied to an argument
- In untyped languages (e.g., LISP), the inlining optimization may not terminate

```
let w = fn f. fn y. f (y f y) in
let fix = fn f. f (f (f (... f (w f w))))
```

(...)

Note

- Inlining can be seen as special case of let-reduction
- However: Does not change termination behavior or side-effects
  - Only inlining terms of form `fn x. e`, which are not evaluated, unless applied to an argument
- In untyped languages (e.g., LISP), the inlining optimization may not terminate
- In typed languages like OCaml or Haskell, however, we have
  - Inlining always terminates

Table of Contents

Specialization of recursive functions

- Function to square all elements of a list
  - Note: Dropping the restriction that let-rec occurs outermost

```
let rec map = fn f. fn l.
  match l with
    [] => []
  | x#l => f x # map f l
in
let f = fn x. x*x in
let sqrl = map f in ...
```

Specialization of recursive functions

- Function to square all elements of a list
  - Note: Dropping the restriction that let-rec occurs outermost
- Requires many function calls to *f*

```
let rec map = fn f. fn l.
  match l with
    [] => []
  | x#l => f x # map f l
in
let f = fn x. x*x in
let sqrl = map f in ...
```

Specialization of recursive functions

- Function to square all elements of a list
  - Note: Dropping the restriction that let-rec occurs outermost
- Requires many function calls to *f*
- Idea: Replace `map f` by new function `mapf`

```
let rec map = fn f. fn l.
  match l with
    [] => []
  | x#l => f x # map f l
in
let f = fn x. x*x in
let sqrl = map f in ...
```

Specialization of recursive functions

- Function to square all elements of a list
  - Note: Dropping the restriction that let-rec occurs outermost
- Requires many function calls to *f*
- Idea: Replace `map f` by new function `mapf`
- Specialization of *map* for argument *f*

```
let rec map = fn f. fn l.
  match l with
    [] => []
  | x#l => f x # map f l
in
let f = fn x. x*x in
let sqrl = map f in ...
```

Specialization of recursive functions

- Function to square all elements of a list
  - Note: Dropping the restriction that let-rec occurs outermost
- Requires many function calls to *f*
- Idea: Replace `map f` by new function `mapf`
- Specialization of *map* for argument *f*

```
let rec map = fn f. fn l.
  match l with
    [] => []
  | x#l => f x # map f l
in
let f = fn x. x*x in
let rec mapf = fn l.
  match l with
    [] => []
  | x#l => f x # mapf l
in
let sqrl = mapf in ...
```

(Specialization)

Specialization of recursive functions

- Function to square all elements of a list
  - Note: Dropping the restriction that let-rec occurs outermost
- Requires many function calls to *f*
- Idea: Replace `map f` by new function `mapf`
- Specialization of *map* for argument *f*

```
let rec map = fn f. fn l.
  match l with
    [] => []
  | x#l => f x # map f l
in
let f = fn x. x*x in
let rec mapf = fn l.
  match l with
    [] => []
  | x#l => x*x # mapf l
in
let sqrl = mapf in ...
```

(Inlining)

Function folding

- When specializing function `f a` to `fa`,

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*
  - Beware of name-captures!

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*
  - Beware of name-captures!
- If recursive function calls alter the specialized argument:

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*
  - Beware of name-captures!
- If recursive function calls alter the specialized argument:
  - Potential for new specializations may be created

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*
  - Beware of name-captures!
- If recursive function calls alter the specialized argument:
  - Potential for new specializations may be created
  - Infinitely often ...

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*
  - Beware of name-captures!
- If recursive function calls alter the specialized argument:
  - Potential for new specializations may be created
  - Infinitely often ...
  - `let rec f = fn g. fn l. ... f (fn x. g (g x)) ...`

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*
  - Beware of name-captures!
- If recursive function calls alter the specialized argument:
  - Potential for new specializations may be created
  - Infinitely often ...
  - `let rec f = fn g. fn l. ... f (fn x. g (g x)) ...`
- Safe and simple heuristics:

Function folding

- When specializing function `f a` to `fa`,
  - we may replace `f a` by `fa` in definition of *fa*
  - Beware of name-captures!
- If recursive function calls alter the specialized argument:
  - Potential for new specializations may be created
  - Infinitely often ...
  - `let rec f = fn g. fn l. ... f (fn x. g (g x)) ...`
- Safe and simple heuristics:
  - Only specialize functions of the form

    ```
    let rec f = fn x. e
    ```

    such that recursive occurrences of *f* in *e* have the form *f x*

Table of Contents

Deforestation

Deforestation

- Idea: Often, lists are used as intermediate data structures

Deforestation

- Idea: Often, lists are used as intermediate data structures
- Standard list functions

```
let rec map = fn f. fn l. match l with
  [] => []
| x#xs => f x # map f xs

let rec filter = fn P. fn l. match l with
  [] => []
| x#xs => if P x then x#filter P xs else filter P xs

let rec foldl = fn f. fn a. fn l. match l with
  [] => []
| x#xs => foldl f (f a x) xs
```

Deforestation

- Examples of derived functions

```
let sum = foldl (+) 0

let length = sum o map (fn x. 1)

let der = fn l.
  let n = length l in
  let mean = sum l / n in
  let s2 = (
      sum
    o map (fn x. x*x)
    o map (fn x. x-mean)) l
  in
    s2 / n
```

Idea

- Avoid intermediate list structures

Idea

- Avoid intermediate list structures
- E.g., we could define

```
length = foldl (fn a. fn _. a+1) 0
```

Idea

- Avoid intermediate list structures
- E.g., we could define

```
length = foldl (fn a. fn _. a+1) 0
```

- In general, we can define rules for combinations of the basic list functions like fold, map, filter, ...

```
map f o map g = map (f o g)
foldl f a o map g = foldl (fn a. f a o g) a
filter P o filter Q = filter (fn x. P x & Q x)
...
```

Idea

- Avoid intermediate list structures
- E.g., we could define

```
length = foldl (fn a. fn _. a+1) 0
```

- In general, we can define rules for combinations of the basic list functions like fold, map, filter, ...

```
map f o map g = map (f o g)
foldl f a o map g = foldl (fn a. f a o g) a
filter P o filter Q = filter (fn x. P x & Q x)
...
```

- We may also need versions of these rules in first-order form, e.g. `map f (map g l) = ...`

Example

```
let der = fn l.
  let n = length l in
  let mean = sum l / n in
  let s2 = (
      sum
    o map (fn x. x*x)
    o map (fn x. x-mean)) l
  in
    s2 / n
```

Example

```
let der = fn l.
  let n = length l in
  let mean = sum l / length l in
  let s2 = (
      foldl (+) 0
    o map (fn x. x*x)
    o map (fn x. x-mean)) l
  in
    s2 / n
```

Let-optimization/ inlining

Example

```
let der = fn l.
  let n = length l in
  let mean = sum l / length l in
  let s2 = (
      foldl (+) 0
    o map ((fn x. x*x) o (fn x. x-mean))) l
  in
    s2 / n
```

map-map rule

Example

```
let der = fn l.
  let n = length l in
  let mean = sum l / length l in
  let s2 = foldl (
      fn a. (+) a o (fn x. x*x) o (fn x. x-mean)
    ) 0 l
  in
    s2 / n
```

fold-map rule

Example

```
let der = fn l.
  let n = length l in
  let mean = sum l / length l in
  let s2 = foldl (
      fn a. fn x. let x=x-mean in let x=x*x in a+x
    ) 0 l
  in
    s2 / n
```

function-application, unfolding of *o*, let-optimization.

Discussion

- Beware of side-effects!

Discussion

- Beware of side-effects!
- Need rules for many combinations of functions.

Discussion

- Beware of side-effects!
- Need rules for many combinations of functions.
    - Does not scale

Discussion

- Beware of side-effects!
- Need rules for many combinations of functions.
    - Does not scale
- Only works for built-in functions

Discussion

- Beware of side-effects!
- Need rules for many combinations of functions.
  - Does not scale
- Only works for built-in functions
  - Could try to automatically recognize user-defined functions

Discussion

- Beware of side-effects!
- Need rules for many combinations of functions.
  - Does not scale
- Only works for built-in functions
  - Could try to automatically recognize user-defined functions
- Can be extended to algebraic datatypes in general

Discussion

- Beware of side-effects!
- Need rules for many combinations of functions.
  - Does not scale
- Only works for built-in functions
  - Could try to automatically recognize user-defined functions
- Can be extended to algebraic datatypes in general
  - They all have standard map and fold functions

Reducing the number of required rules

- Try to find standard representation

Reducing the number of required rules

- Try to find standard representation
- `foldr` seems to be a good candidate:

```
foldr f a [] = a
foldr f a (x#xs) = f x (foldr f a xs)
```

Reducing the number of required rules

- Try to find standard representation
- `foldr` seems to be a good candidate:

```
foldr f a [] = a
foldr f a (x#xs) = f x (foldr f a xs)
```

- We can represent *map*, *filter*, *sum*, ...

Reducing the number of required rules

- Try to find standard representation
- `foldr` seems to be a good candidate:

  ```
  foldr f a [] = a
  foldr f a (x#xs) = f x (foldr f a xs)
  ```

- We can represent *map*, *filter*, *sum*, ...
  - But no list-reversal, as *foldl* can

Reducing the number of required rules

- Try to find standard representation
- `foldr` seems to be a good candidate:

```
foldr f a [] = a
foldr f a (x#xs) = f x (foldr f a xs)
```

- We can represent *map*, *filter*, *sum*, ...
  - But no list-reversal, as *foldl* can
- Problem: How to compose two foldr-calls?

Reducing the number of required rules

- Try to find standard representation
- `foldr` seems to be a good candidate:

```
foldr f a [] = a
foldr f a (x#xs) = f x (foldr f a xs)
```

- We can represent *map*, *filter*, *sum*, ...
  - But no list-reversal, as *foldl* can
- Problem: How to compose two foldr-calls?
  - `foldr f1 a1 (foldr f2 a2 l) = ???`

Composition of *foldr*

- Idea: Abstract over constructors

```
map f l = foldr (fn l. fn x. f x#l) [] l

map' f l = fn c. fn n.
  foldr (fn l. fn x. c (f x) l) n l
build g = g (#) []
map f l = build (map' f l)
```

Composition of *foldr*

- Idea: Abstract over constructors

```
map f l = foldr (fn l. fn x. f x#l) [] l

map' f l = fn c. fn n.
  foldr (fn l. fn x. c (f x) l) n l
build g = g (#) []
map f l = build (map' f l)
```

- Have

```
foldr f a (build g) = g f a
```

Composition of *foldr*

- Idea: Abstract over constructors

```
map f l = foldr (fn l. fn x. f x#l) [] l

map' f l = fn c. fn n.
  foldr (fn l. fn x. c (f x) l) n l
build g = g (#) []
map f l = build (map' f l)
```

- Have

```
foldr f a (build g) = g f a
```

  - If abstraction over list inside *g* done properly
  - I.e., *g* actually produces list using its arguments

Example

```
map f (map g l)
```

Example

```
map f (map g l)

= build (map' f (build (map' g l)))
```

Example

```
map f (map g l)

= build (map' f (build (map' g l)))

= build (fn c. fn n.
    foldr (fn l. fn x. c (f x) l) n (build (map' g l)))
```

Example

```
map f (map g l)

= build (map' f (build (map' g l)))

= build (fn c. fn n.
    foldr (fn l. fn x. c (f x) l) n (build (map' g l)))

= build (fn c. fn n. map' g l (fn l. fn x. c (f x) l) n)
```

Intuition

- Functions may consume lists (*foldr*), produce lists (*build*), or both

Intuition

- Functions may consume lists (*foldr*), produce lists (*build*), or both
- Applying a chain of functions: (*build foldr*) (*build foldr*) ... (*build foldr*)

Intuition

- Functions may consume lists (*foldr*), produce lists (*build*), or both
- Applying a chain of functions: (*build foldr*) (*build foldr*) ... (*build foldr*)
- Can be re-bracketed to *build* (*foldr build*) ... (*foldr build*) *foldr*

Intuition

- Functions may consume lists (*foldr*), produce lists (*build*), or both
- Applying a chain of functions: (*build foldr*) (*build foldr*) ... (*build foldr*)
- Can be re-bracketed to *build* (*foldr build*) ... (*foldr build*) *foldr*
- And the inner pairs cancel out, leaving a single *build foldr*

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`
  - Only correct if *g* is abstracted over list correctly

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`
  - Only correct if *g* is abstracted over list correctly
  - Consider, e.g., `foldr f a (build (fn _. fn _. [True]))`

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`
  - Only correct if *g* is abstracted over list correctly
  - Consider, e.g., `foldr f a (build (fn _. fn _. [True]))`
    - Which is, in general, not the same as `(fn _. fn _. [True]) f a`

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`
  - Only correct if *g* is abstracted over list correctly
  - Consider, e.g., `foldr f a (build (fn _. fn _. [True]))`
    - Which is, in general, not the same as `(fn _. fn _. [True]) f a`
- If language is parametric, can be enforced via type:

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`
  - Only correct if *g* is abstracted over list correctly
  - Consider, e.g., `foldr f a (build (fn _. fn _. [True]))`
    - Which is, in general, not the same as `(fn _. fn _. [True]) f a`
- If language is parametric, can be enforced via type:
  - If *g* has type $\forall\beta.(A \to \beta \to \beta) \to \beta \to \beta$

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`
  - Only correct if *g* is abstracted over list correctly
  - Consider, e.g., `foldr f a (build (fn _. fn _. [True]))`
    - Which is, in general, not the same as `(fn _. fn _. [True]) f a`
- If language is parametric, can be enforced via type:
  - If *g* has type $\forall\beta.(A \to \beta \to \beta) \to \beta \to \beta$
  - It can only produce its result of type β by using its arguments

Discussion

- Single rule for deforestation: `foldr f a (build g) = g f a`
  - Only correct if *g* is abstracted over list correctly
  - Consider, e.g., `foldr f a (build (fn _. fn _. [True]))`
    - Which is, in general, not the same as `(fn _. fn _. [True]) f a`
- If language is parametric, can be enforced via type:
  - If *g* has type $\forall\beta.(A \to \beta \to \beta) \to \beta \to \beta$
  - It can only produce its result of type β by using its arguments
  - Which is exactly the required abstraction over the list constructors

Wrap-up

- Transformations for functional programs

Wrap-up

- Transformations for functional programs
  - Let-optimization

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining
  - Specialization

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining
  - Specialization
  - Deforestation

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining
  - Specialization
  - Deforestation
  - ...

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining
  - Specialization
  - Deforestation
  - ...
- Aim at reducing complexity before translation to IR

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining
  - Specialization
  - Deforestation
  - ...
- Aim at reducing complexity before translation to IR
- On (imperative) IR, all former optimizations of this lecture can be done

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining
  - Specialization
  - Deforestation
  - ...
- Aim at reducing complexity before translation to IR
- On (imperative) IR, all former optimizations of this lecture can be done
  - Important one: Tail-call optimization

Wrap-up

- Transformations for functional programs
  - Let-optimization
  - Inlining
  - Specialization
  - Deforestation
  - ...
- Aim at reducing complexity before translation to IR
- On (imperative) IR, all former optimizations of this lecture can be done
  - Important one: Tail-call optimization
    - There are no loops in functional languages

That's it!

Questions?