Software Verification

Grégoire Sutre

LaBRI, University of Bordeaux, CNRS, France

Summer School on Verification Technology, Systems & Applications

September 2008

Part 1

Part I

Introduction

Outline — Introduction

1. Software Verification: Why?
2. Software Verification: How?

Outline — Introduction

1. Software Verification: Why?
2. Software Verification: How?

Ubiquity of Software in Modern Life

Once upon a time, lecturers used hand-written transparencies with an overhead projector.

- pens
- transparencies
- scissors
- sticky tape
- lamp
- lenses
- mirror
- screen

Nowadays softwares are used to design the slides and to project them

Similar evolution in many, many areas

Ubiquity of Software in Modern Life

Once upon a time, lecturers used hand-written transparencies with an overhead projector.

- pens
- transparencies
- scissors
- sticky tape
- lamp
- lenses
- mirror
- screen

Nowadays softwares are used to design the slides and to project them

Similar evolution in many, many areas

Why?

Some advantages of software over dedicated hardware components

- Reduce time to market
  - Less time to write the slides (really?)
  - Ability to re-organize the presentation
- Reduce costs
  - No pen, no transparencies
  - Re-usability of slides, ability to make minor modifications for free
- Increase functionality
  - Automatic generation of some slides (table of contents)
  - Nicer overlays (sticky tape is not required anymore!)
  - Ability to display videos

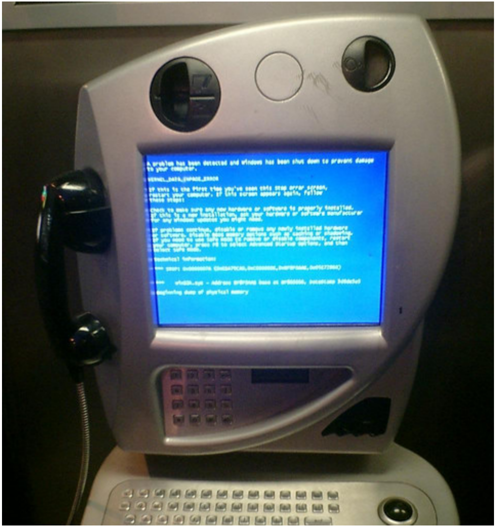
But software is not without risk. . .

Bugs are Frequent in Software

Bugs are Frequent in Software

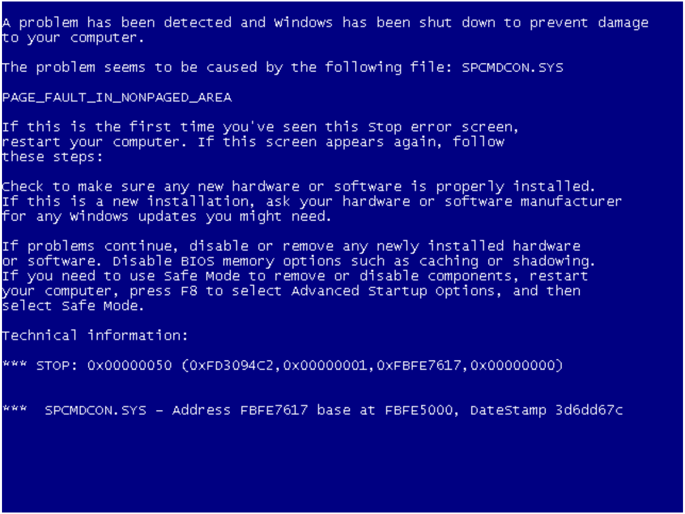
Bugs are Frequent in Software

Bugs are Frequent in Software



Bugs are Frequent in Software

Bugs are Frequent in Software



A Critical Software Bug: Ariane 5.01

« On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. »

« The failure of the Ariane 5.01 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system. »

A Critical Software Bug: Ariane 5.01

« On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. »

« The failure of the Ariane 5.01 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system. »

Software in Embedded Systems

Embedded systems in: cell phones, satellites, airplanes, cars, wireless routers, MP3 players, refrigerators, ...

Examples of Critical Systems

- attitude and orbit control systems in satellites
- *X-by-wire* control systems in airplanes and in cars (soon)

Increasing importance of software in embedded systems

- custom hardware replaced by processor + custom software
- software is a dominant factor in design time and cost (70 %)

Critical embedded systems require "exhaustive" validation

Software Complexity Grows Exponentially

As computational power grows . . .

Moore's law: *« the number of transistors on a chip doubles every two years »*

. . . software complexity grows . . .

Wirth's Law: *« software gets slower faster than hardware gets faster »*

. . . and so does the number of bugs!

Watts S. Humphrey: *« 5 – 10 bugs per 1000 lines of code after product test »*

Growing need for automatic validation techniques

Software Complexity Grows Exponentially

As computational power grows . . .

Moore's law: *« the number of transistors on a chip doubles every two years »*

. . . software complexity grows . . .

Wirth's Law: *« software gets slower faster than hardware gets faster »*

. . . and so does the number of bugs!

Watts S. Humphrey: *« 5 – 10 bugs per 1000 lines of code after product test »*

Growing need for automatic validation techniques

Software Complexity Grows Exponentially

As computational power grows . . .

Moore's law: *« the number of transistors on a chip doubles every two years »*

. . . software complexity grows . . .

Wirth's Law: *« software gets slower faster than hardware gets faster »*

. . . and so does the number of bugs!

Watts S. Humphrey: *« 5 – 10 bugs per 1000 lines of code after product test »*

Growing need for automatic validation techniques

Software Complexity Grows Exponentially

As computational power grows . . .

Moore's law: *« the number of transistors on a chip doubles every two years »*

. . . software complexity grows . . .

Wirth's Law: *« software gets slower faster than hardware gets faster »*

. . . and so does the number of bugs!

Watts S. Humphrey: *« 5 – 10 bugs per 1000 lines of code after product test »*

Growing need for automatic validation techniques

1. Software Verification: Why?
2. Software Verification: How?

Software Testing

Running the executable (obtained by compilation)

- on multiple inputs
- usually on the target platform

Testing is a widespread validation approach in the software industry

- can be (partially) automated
- can detect a lot of bugs

But

Costly and time-consuming

Not exhaustive

Software Testing

Running the executable (obtained by compilation)

- on multiple inputs
- usually on the target platform

Testing is a widespread validation approach in the software industry

- can be (partially) automated
- can detect a lot of bugs

But

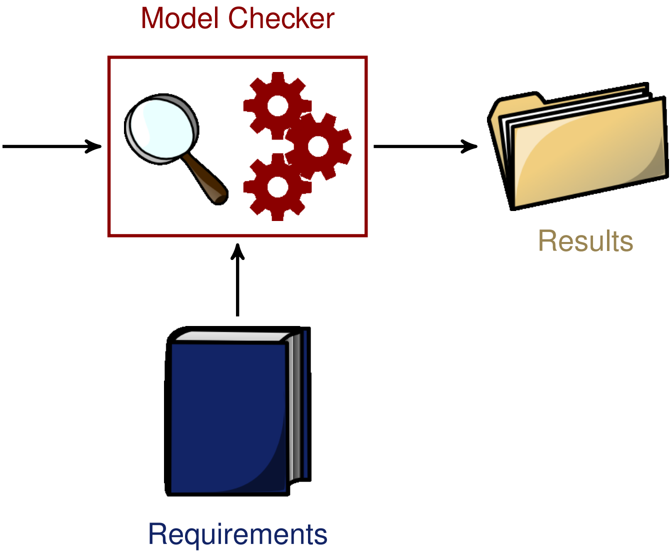
Costly and time-consuming

Not exhaustive

Dream of Software Model-Checking

```
x = 1;
if (y <= 10) {
  y = 10;
}
else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
x = y + 1;
```

Program



Fundamental Limit: Undecidability

Rice's Theorem

Any non-trivial semantic property of programs is undecidable.

Classical Example: Termination

There exists no algorithm which can solve the halting problem:

- given a description of a program as input,
- decide whether the program terminates or loops forever.

Practical Limit: Combinatorial Explosion

Implicit in Rice's Theorem is an idealized program model, where programs have access to *unbounded memory*.

In reality programs are run on a computer with *bounded memory*.

Model-checking becomes decidable for finite-state systems.

But even with bounded memory, complexity in practice is too high for finite-state model-checking:

- 1 megabyte (1 000 000 bytes) of memory $\approx$ $10^{2\,400\,000}$ states
- 1000 variables $\times$ 64 bits $\approx$ $10^{19\,200}$ states
- optimistic limit for finite-state model checkers: 10^{100} states

More Realistic Objectives for Software Verification

Incomplete Methods

Approximate Algorithms

- ☺ Always terminate
- ☹ Indefinite answer (yes / no / ?)

Exact Semi-Algorithms

- ☺ Definite answer (yes / no)
- ☹ May not terminate

Topics of the lecture

Static Analysis

Abstraction Refinement

More Realistic Objectives for Software Verification

Incomplete Methods

Approximate Algorithms

- ☺ Always terminate
- ☹ Indefinite answer (yes / no / ?)

Exact Semi-Algorithms

- ☺ Definite answer (yes / no)
- ☹ May not terminate

Topics of the lecture

Static Analysis

Abstraction Refinement

Static Analysis

Tentative Definition

Compile-time techniques to gather run-time information about programs without actually running them

Example

Detection of variables that are used before initialization

- ☺ Always terminates
- ☺ Applies to large programs
- ☹ Simple analyses (original goal was compilation)
- ☹ Indefinite answer (yes / no / ?)

In the Lecture

Data Flow Analysis   Abstract Interpretation

Static Analysis

Tentative Definition

Compile-time techniques to gather run-time information about programs without actually running them

Example

Detection of variables that are used before initialization

- ☺ Always terminates
- ☺ Applies to large programs
- ☹ Simple analyses (original goal was compilation)
- ☹ Indefinite answer (yes / no / ?)

In the Lecture

Data Flow Analysis   Abstract Interpretation

Abstraction Refinement

Tentative Definition

Analysis-time techniques to verify programs by model-checking and refinement of finite-state approximate models

Example

Verification of safety and fairness of a mutual exclusion algorithm

- ☺ Complex analyses (properties expressed in temporal logics)
- ☺ Definite answer (yes / no)
- ☹ May not terminate
- ☹ Modeling of the program into a finite-state transition system

In the Lecture

Abstract Model Refinement for Safety Properties

Abstraction Refinement

Tentative Definition

Analysis-time techniques to verify programs by model-checking and refinement of finite-state approximate models

Example

Verification of safety and fairness of a mutual exclusion algorithm

- ☺ Complex analyses (properties expressed in temporal logics)
- ☺ Definite answer (yes / no)
- ☹ May not terminate
- ☹ Modeling of the program into a finite-state transition system

In the Lecture

Abstract Model Refinement for Safety Properties

Common Ingredient: Property-Preserving Abstraction

Abstraction Process

Interpret programs according to a simplified, “abstract” semantics.

Property-Preserving Abstraction

Formally relate the “abstract” semantics with the “standard” semantics, so as to preserve relevant properties.

Preservation of Properties

Program interpretation with this abstract semantics therefore gives “correct” information about properties of real runs.

Abstract Interpretation Example: Sign Analysis

Objective of Sign Analysis

Discover for each program point the sign of possible run-time values that numerical variables can have at that point.

The abstract semantics "tracks" the following information, for each variable x:

$$
\begin{aligned}
x &< 0 \\
x &\leq 0 \\
x &= 0 \\
x &\geq 0 \\
x &> 0
\end{aligned}
$$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
  y = 10;
}
else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
x = y + 1;
assert(x > 0);
```

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

$x > 0$

Abstract Interpretation Example: Sign Analysis

```
x = 1;                    x > 0
if (y ≤ 10) {             x > 0
   y = 10;                x > 0 ∧ y > 0
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

$x > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
                                  x > 0
if (y ≤ 10) {
                                  x > 0
   y = 10;
                                  x > 0 ∧ y > 0
}
else {
                                  x > 0 ∧ y > 0
   while (x < y) {
                                  x > 0 ∧ y > 0
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

$x > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
                              x > 0
if (y ≤ 10) {
                              x > 0
   y = 10;
                              x > 0 ∧ y > 0
}
else {
                              x > 0 ∧ y > 0
   while (x < y) {
                              x > 0 ∧ y > 0
      x = 2 * x;
                              x > 0 ∧ y > 0
      y = y - 1;
                              x > 0 ∧ y ≥ 0
   }
}
x = y + 1;
assert(x > 0);
```

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

$x > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0 \quad \bigvee \quad x > 0 \wedge y \geq 0 \wedge x < y$

$x > 0 \wedge y > 0$

$x > 0 \wedge y \geq 0$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$$x > 0$$

$$x > 0$$

$$x > 0 \wedge y > 0$$

$$x > 0 \wedge y > 0$$

$$x > 0 \wedge y > 0 \quad \bigvee \quad x > 0 \wedge y \geq 0 \wedge x < y$$

$$x > 0 \wedge y > 0$$

$$x > 0 \wedge y \geq 0$$

$$x > 0 \wedge y \geq 0$$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

$x > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0 \quad \bigvee \quad x > 0 \wedge y \geq 0 \wedge x < y$

$x > 0 \wedge y > 0$

$x > 0 \wedge y \geq 0$

$x > 0 \wedge y \geq 0$

$x > 0 \wedge y \geq 0 \quad (x > 0 \wedge y > 0) \bigvee (x > 0 \wedge y \geq 0)$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

$x > 0$

$x > 0 \land y > 0$

$x > 0 \land y > 0$

$x > 0 \land y > 0 \quad \bigvee \quad x > 0 \land y \geq 0 \land x < y$

$x > 0 \land y > 0$

$x > 0 \land y \geq 0$

$x > 0 \land y \geq 0$

$x > 0 \land y \geq 0 \quad (x > 0 \land y > 0) \bigvee (x > 0 \land y \geq 0)$

$x > 0 \land y \geq 0$

Abstract Interpretation Example: Sign Analysis

```
x = 1;
if (y ≤ 10) {
  y = 10;
}
else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
x = y + 1;
assert(x > 0);
```

$x > 0$

$x > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0$

$x > 0 \wedge y > 0 \quad \bigvee \quad x > 0 \wedge y \geq 0 \wedge x < y$

$x > 0 \wedge y > 0$

$x > 0 \wedge y \geq 0$

$x > 0 \wedge y \geq 0$

$x > 0 \wedge y \geq 0 \quad (x > 0 \wedge y > 0) \bigvee (x > 0 \wedge y \geq 0)$

$x > 0 \wedge y \geq 0$

Credits: Pioneers (1970's)

Iterative Data Flow Analysis

Gary Kildall

John Kam & Jeffrey Ullman

Michael Karr

. . .

Abstract Interpretation

Patrick Cousot & Radhia Cousot

Nicolas Halbwachs

. . .

And many, many more. . .

Apologies!

Outline of the Lecture

- Control Flow Automata
- Data Flow Analysis
- Abstract Interpretation
- Abstract Model Refinement

Outline of the Lecture

- Control Flow Automata
- Data Flow Analysis
- Abstract Interpretation
- Abstract Model Refinement

Static Analysis

Outline of the Lecture

- Control Flow Automata
- Data Flow Analysis
- Abstract Interpretation
- Abstract Model Refinement

Static Analysis

Abstraction Refinement

Part II

Control Flow Automata

Outline — Control Flow Automata

Outline — Control Flow Automata

3 Syntax and Semantics

4 Verification of Control Flow Automata

Short Introduction to Control Flow Automata

Requirement for verification: formal semantics of programs

Formal Semantics

Formalization as a mathematical model of the meaning of programs

Denotational Operational Axiomatic

Operational Semantics

Labeled transition system describing the possible computational steps

First Step Towards an Operational Semantics

Program text $\longrightarrow$ Graph-based representation

Short Introduction to Control Flow Automata

Requirement for verification: formal semantics of programs

Formal Semantics

Formalization as a mathematical model of the meaning of programs

Denotational   Operational   Axiomatic

Operational Semantics

Labeled transition system describing the possible computational steps

First Step Towards an Operational Semantics

Program text $\longrightarrow$ Graph-based representation

Short Introduction to Control Flow Automata

Requirement for verification: formal semantics of programs

Formal Semantics

Formalization as a mathematical model of the meaning of programs

Denotational   Operational   Axiomatic

Operational Semantics

Labeled transition system describing the possible computational steps

First Step Towards an Operational Semantics

Program text $\longrightarrow$ Graph-based representation

Short Introduction to Control Flow Automata

Requirement for verification: formal semantics of programs

Formal Semantics

Formalization as a mathematical model of the meaning of programs

Denotational Operational Axiomatic

Operational Semantics

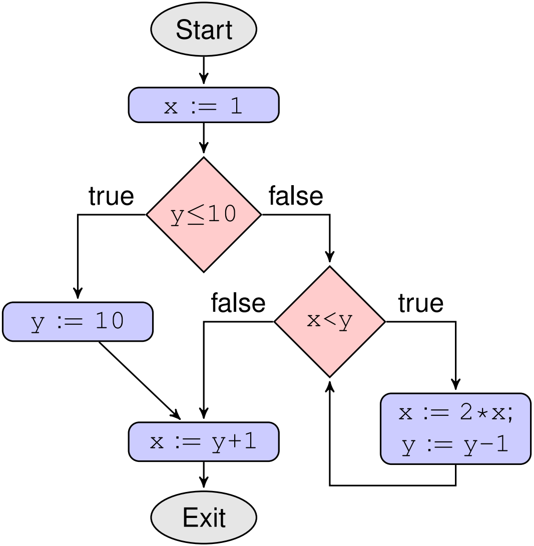
Labeled transition system describing the possible computational steps

First Step Towards an Operational Semantics

Program text $\longrightarrow$ Graph-based representation
Control flow automaton

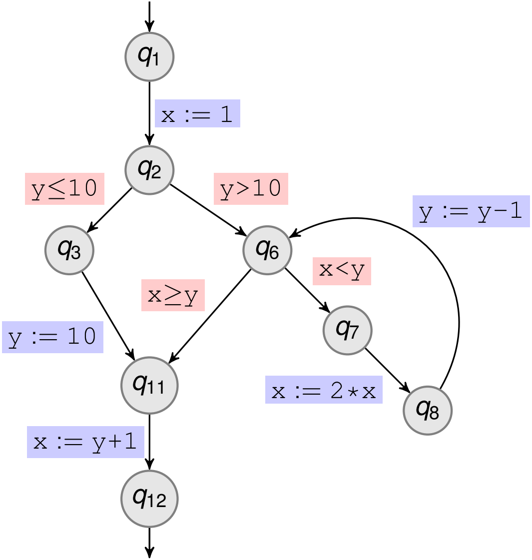
Control Flow Graph

```
x = 1;
if (y ≤ 10) {
   y = 10;
}
else {
   while (x < y) {
      x = 2 * x;
      y = y - 1;
   }
}
x = y + 1;
```



Control Flow Automaton

```
x = 1;
if (y ≤ 10) {
  y = 10;
}
else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
x = y + 1;
```



Labeled Directed Graphs

Definition

A labeled directed graph is a triple $G = \langle V, \Sigma, \rightarrow \rangle$ where:

- V is a finite set of *vertices*,
- Σ is a finite set of *labels*,
- $\rightarrow \subseteq V \times \Sigma \times V$ is a finite set of *edges*.

Notation for edges: $v \xrightarrow{\sigma} v'$ instead of $(v, \sigma, v') \in \rightarrow$

A path in G is a finite sequence $v_0 \xrightarrow{\sigma_0} v'_0, \ldots, v_k \xrightarrow{\sigma_k} v'_k$ of edges such that $v'_i = v_{i+1}$ for each $0 \leq i < k$.

Notation for paths: $v_0 \xrightarrow{\sigma_0} v_1 \cdots v_k \xrightarrow{\sigma_k} v'_k$

Labeled Directed Graphs

Definition

A labeled directed graph is a triple $G = \langle V, \Sigma, \rightarrow \rangle$ where:

- V is a finite set of *vertices*,
- Σ is a finite set of *labels*,
- $\rightarrow \subseteq V \times \Sigma \times V$ is a finite set of *edges*.

Notation for edges: $v \xrightarrow{\sigma} v'$ instead of $(v, \sigma, v') \in \rightarrow$

A path in G is a finite sequence $v_0 \xrightarrow{\sigma_0} v'_0, \ldots, v_k \xrightarrow{\sigma_k} v'_k$ of edges such that $v'_i = v_{i+1}$ for each $0 \leq i < k$.

Notation for paths: $v_0 \xrightarrow{\sigma_0} v_1 \cdots v_k \xrightarrow{\sigma_k} v'_k$

Labeled Directed Graphs

Definition

A labeled directed graph is a triple $G = \langle V, \Sigma, \rightarrow \rangle$ where:

- V is a finite set of *vertices*,
- Σ is a finite set of *labels*,
- $\rightarrow \subseteq V \times \Sigma \times V$ is a finite set of *edges*.

Notation for edges: $v \xrightarrow{\sigma} v'$ instead of $(v, \sigma, v') \in \rightarrow$

A path in G is a finite sequence $v_0 \xrightarrow{\sigma_0} v'_0, \ldots, v_k \xrightarrow{\sigma_k} v'_k$ of edges such that $v'_i = v_{i+1}$ for each $0 \leq i < k$.

Notation for paths: $v_0 \xrightarrow{\sigma_0} v_1 \cdots v_k \xrightarrow{\sigma_k} v'_k$

Control Flow Automata: Syntax

Definition

A control flow automaton is a quintuple $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$ where:

- Q is a finite set of *locations*,
- $q_{in} \in Q$ is an *initial location* and $q_{out} \in Q$ is an *exit location*,
- X is a finite set of *variables*,
- $\rightarrow \subseteq Q \times \mathrm{Op} \times Q$ is a finite set of *transitions*.

Op is the set of operations defined by:

$$
\begin{aligned}
cst &::= c \in \mathbb{Q} \\
var &::= x \in \mathrm{X} \\
expr &::= cst \mid var \mid expr \bullet expr, \text{ with } \bullet \in \{+, -, *\} \\
guard &::= expr \blacktriangleleft expr, \text{ with } \blacktriangleleft \in \{<, \leq, =, \neq, \geq, >\} \\
\mathrm{Op} &::= guard \mid var := expr
\end{aligned}
$$

Control Flow Automata: Syntax

Definition

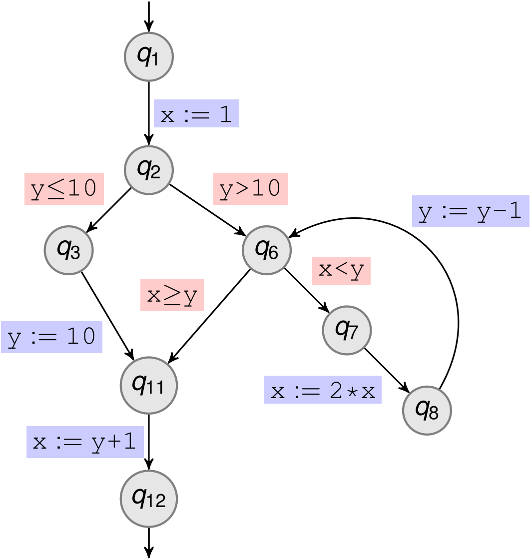
A control flow automaton is a quintuple $\langle Q, q_{in}, q_{out}, \mathtt{X}, \rightarrow \rangle$ where:

- Q is a finite set of *locations*,
- $q_{in} \in Q$ is an *initial location* and $q_{out} \in Q$ is an *exit location*,
- $\mathtt{X}$ is a finite set of *variables*,
- $\rightarrow \subseteq Q \times \mathtt{Op} \times Q$ is a finite set of *transitions*.

$\mathtt{Op}$ is the set of operations defined by:

$$
\begin{aligned}
cst &::= c \in \mathbb{Q} \\
var &::= x \in \mathtt{X} \\
expr &::= cst \mid var \mid expr \bullet expr, \text{ with } \bullet \in \{+, -, *\} \\
guard &::= expr \blacktriangleleft expr, \text{ with } \blacktriangleleft \in \{<, \leq, =, \neq, \geq, >\} \\
\mathtt{Op} &::= guard \mid var := expr
\end{aligned}
$$

Control Flow Automata: Syntax



$$Q = \left\{ \begin{array}{l} q_1, q_2, q_3, q_6, \\ q_7, q_8, q_{11}, q_{12} \end{array} \right\}$$

$$q_{in} = q_1$$

$$q_{out} = q_{12}$$

$$X = \{x, y\}$$

$$\rightarrow = \left\{ \begin{array}{l} (q_1,\ x := 1\ , q_2), \\ (q_2,\ y \leq 10\ , q_3), \\ (q_2,\ y > 10\ , q_6), \\ (q_3, y := 10, q_{11}), \\ \ldots \end{array} \right\}$$

Programs as Control Flow Automata

Control flow automata can model:

- ☺ flow of control (program points),
- ☺ numerical variables and numerical operations,
- ☺ non-determinism (uninitialized variables, boolean inputs).

Control flow automata cannot model:

- ☹ pointers
- ☹ recursion
- ☹ threads
- ☹ ...

But they are complex enough for verification... ... and for learning!

Programs as Control Flow Automata

Control flow automata can model:

- ☺ flow of control (program points),
- ☺ numerical variables and numerical operations,
- ☺ non-determinism (uninitialized variables, boolean inputs).

Control flow automata cannot model:

- ☹ pointers
- ☹ recursion
- ☹ threads
- ☹ ...

But they are complex enough for verification... ... and for learning!

Programs as Control Flow Automata

Control flow automata can model:

- ☺ flow of control (program points),
- ☺ numerical variables and numerical operations,
- ☺ non-determinism (uninitialized variables, boolean inputs).

Control flow automata cannot model:

- ☹ pointers
- ☹ recursion
- ☹ threads
- ☹ ...

Forget about these...

But they are complex enough for verification... ... and for learning!

Verification of Safety Properties

Goal

Check that "nothing bad can happen".

Bad behaviors specified e.g. as assertion violations in the original program

An assertion violation can be modeled as a location:

```
assert(x > 0)    ⟹    if (x > 0) then { BAD: }
```

Goal (refined)

Check that there is no "run" that visits a location q contained in a given set $Q_{BAD} \subseteq Q$ of bad locations.

Verification of Safety Properties

Goal

Check that "nothing bad can happen".

Bad behaviors specified e.g. as assertion violations in the original program

An assertion violation can be modeled as a location:

```
assert(x > 0)    ⟹    if (x > 0) then { BAD: }
```

Goal (refined)

Check that there is no "run" that visits a location q contained in a given set $Q_{BAD} \subseteq Q$ of bad locations.

Verification of Safety Properties

Goal

Check that "nothing bad can happen".

Bad behaviors specified e.g. as assertion violations in the original program

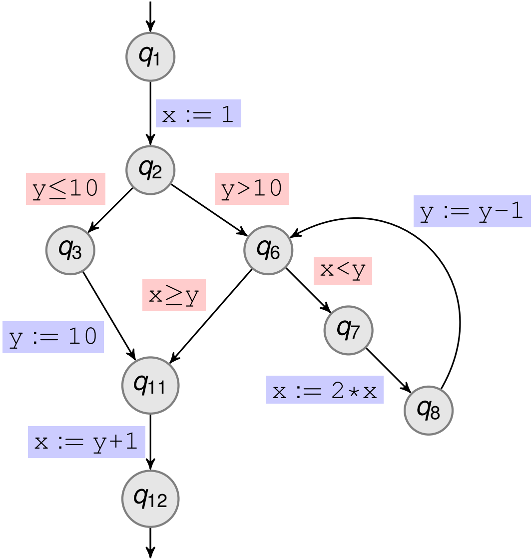
An assertion violation can be modeled as a location:

```
assert(x > 0)    ⟹    if (x > 0) then { BAD: }
```

Goal (refined)

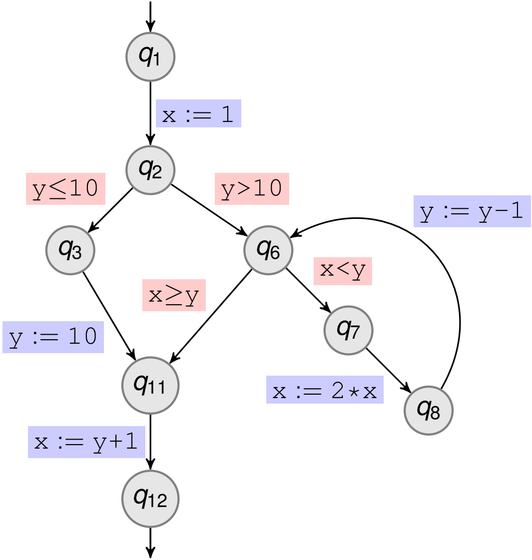
Check that there is no "run" that visits a location q contained in a given set $Q_{BAD} \subseteq Q$ of bad locations.

Runs: Examples



$$(q_1, 0, 0) \xrightarrow{\texttt{x := 1}} (q_2, 1, 0) \xrightarrow{\texttt{y}\leq\texttt{10}} (q_3, 1, 0) \xrightarrow{\texttt{y := 10}} (q_{11}, 1, 10) \xrightarrow{\texttt{x := y+1}} (q_{12}, 11, 10)$$

Runs: Examples



$(q_1, -159, 27)$

$\downarrow$ x := 1

$(q_2, 1, 27)$

$\downarrow$ y>10

$(q_6, 1, 27)$

$\downarrow$ x<y

$(q_7, 1, 27)$

$\downarrow$ x := 2*x

$(q_8, 2, 27)$

$\downarrow$ y := y-1

$(q_6, 2, 26)$

$\vdots$

Labeled Transition Systems

Definition

A labeled transition system is a quintuple $\langle C, Init, Out, \Sigma, \rightarrow \rangle$ where :

- C is a set of *configurations*
- $Init \subseteq C$ and $Out \subseteq C$ are sets of *initial* and *exit configurations*
- Σ is a finite set of *actions*
- $\rightarrow \subseteq C \times \Sigma \times C$ is a set of *transitions*

$$\mathrm{Post}(c, \sigma) = \left\{ c' \in C \,\middle|\, c \xrightarrow{\sigma} c' \right\} \qquad \mathrm{Post}(c) = \bigcup_{\sigma \in \Sigma} \mathrm{Post}(c, \sigma)$$

$$\mathrm{Post}(U, \sigma) = \bigcup_{c \in U} \mathrm{Post}(c, \sigma) \qquad \mathrm{Post}(U) = \bigcup_{c \in U} \mathrm{Post}(c)$$

Labeled Transition Systems

Definition

A labeled transition system is a quintuple $\langle C, \mathit{Init}, \mathit{Out}, \Sigma, \rightarrow \rangle$ where :

- C is a set of *configurations*
- $\mathit{Init} \subseteq C$ and $\mathit{Out} \subseteq C$ are sets of *initial* and *exit configurations*
- Σ is a finite set of *actions*
- $\rightarrow \subseteq C \times \Sigma \times C$ is a set of *transitions*

$$\mathrm{Post}(c, \sigma) = \left\{ c' \in C \;\middle|\; c \xrightarrow{\sigma} c' \right\} \qquad \mathrm{Post}(c) = \bigcup_{\sigma \in \Sigma} \mathrm{Post}(c, \sigma)$$

$$\mathrm{Post}(U, \sigma) = \bigcup_{c \in U} \mathrm{Post}(c, \sigma) \qquad \mathrm{Post}(U) = \bigcup_{c \in U} \mathrm{Post}(c)$$

Labeled Transition Systems

Definition

A labeled transition system is a quintuple $\langle C, Init, Out, \Sigma, \rightarrow \rangle$ where :

- C is a set of *configurations*
- $Init \subseteq C$ and $Out \subseteq C$ are sets of *initial* and *exit configurations*
- Σ is a finite set of *actions*
- $\rightarrow \subseteq C \times \Sigma \times C$ is a set of *transitions*

$$\mathrm{Pre}(c, \sigma) = \left\{ c' \in C \;\middle|\; c' \xrightarrow{\sigma} c \right\} \qquad \mathrm{Pre}(c) = \bigcup_{\sigma \in \Sigma} \mathrm{Pre}(c, \sigma)$$

$$\mathrm{Pre}(U, \sigma) = \bigcup_{c \in U} \mathrm{Pre}(c, \sigma) \qquad \mathrm{Pre}(U) = \bigcup_{c \in U} \mathrm{Pre}(c)$$

Semantics of Expressions and Guards

Consider a finite set $\mathbb{X}$ of variables. A valuation is a function $v : \mathbb{X} \to \mathbb{R}$.

Expressions: $[\![e]\!]_v$

$$
\begin{array}{rcll}
[\![c]\!]_v & = & c & [c \in \mathbb{Q}] \\
[\![x]\!]_v & = & v(x) & [x \in \mathbb{X}] \\
[\![e_1 + e_2]\!]_v & = & [\![e_1]\!]_v + [\![e_2]\!]_v & \\
[\![e_1 - e_2]\!]_v & = & [\![e_1]\!]_v - [\![e_2]\!]_v & \\
[\![e_1 * e_2]\!]_v & = & [\![e_1]\!]_v \times [\![e_2]\!]_v &
\end{array}
$$

Guards: $v \models g$

$$
\begin{array}{lllll}
v & \models & e_1 < e_2 & \text{if} & [\![e_1]\!]_v < [\![e_2]\!]_v \\
v & \models & e_1 \leq e_2 & \text{if} & [\![e_1]\!]_v \leq [\![e_2]\!]_v \\
v & \models & e_1 = e_2 & \text{if} & [\![e_1]\!]_v = [\![e_2]\!]_v \\
v & \models & e_1 \neq e_2 & \text{if} & [\![e_1]\!]_v \neq [\![e_2]\!]_v \\
v & \models & e_1 \geq e_2 & \text{if} & [\![e_1]\!]_v \geq [\![e_2]\!]_v \\
v & \models & e_1 > e_2 & \text{if} & [\![e_1]\!]_v > [\![e_2]\!]_v
\end{array}
$$

Semantics of Expressions and Guards

Consider a finite set $\mathbb{X}$ of variables. A valuation is a function $v : \mathbb{X} \to \mathbb{R}$.

Expressions: $[\![e]\!]_v$

$$
\begin{array}{rcll}
[\![c]\!]_v & = & c & [c \in \mathbb{Q}] \\
[\![x]\!]_v & = & v(x) & [x \in \mathbb{X}] \\
[\![e_1 + e_2]\!]_v & = & [\![e_1]\!]_v + [\![e_2]\!]_v & \\
[\![e_1 - e_2]\!]_v & = & [\![e_1]\!]_v - [\![e_2]\!]_v & \\
[\![e_1 * e_2]\!]_v & = & [\![e_1]\!]_v \times [\![e_2]\!]_v &
\end{array}
$$

Guards: $v \models g$

$$
\begin{array}{lcl}
v \models e_1 < e_2 & \textit{if} & [\![e_1]\!]_v < [\![e_2]\!]_v \\
v \models e_1 \leq e_2 & \textit{if} & [\![e_1]\!]_v \leq [\![e_2]\!]_v \\
v \models e_1 = e_2 & \textit{if} & [\![e_1]\!]_v = [\![e_2]\!]_v \\
v \models e_1 \neq e_2 & \textit{if} & [\![e_1]\!]_v \neq [\![e_2]\!]_v \\
v \models e_1 \geq e_2 & \textit{if} & [\![e_1]\!]_v \geq [\![e_2]\!]_v \\
v \models e_1 > e_2 & \textit{if} & [\![e_1]\!]_v > [\![e_2]\!]_v
\end{array}
$$

Semantics of Operations

The semantics $[\![\mathrm{op}]\!]$ of an operation op is defined as a binary relation between valuations before op and valuations after op:

$$[\![\mathrm{op}]\!] \quad \subseteq \quad (\mathrm{X} \to \mathbb{R}) \times (\mathrm{X} \to \mathbb{R})$$

Examples with $\mathrm{X} = \{\mathrm{x}, \mathrm{y}\}$

$$[\![\mathrm{x} * \mathrm{y} \leq 10]\!] \quad = \quad \{(v, v) \mid v(\mathrm{x}) \times v(\mathrm{y}) \leq 10\}$$

$$[\![\mathrm{x} := 3 * \mathrm{x}]\!] \quad = \quad \{(v, v') \mid v'(\mathrm{x}) = 3 \times v(\mathrm{x}) \ \wedge \ v'(\mathrm{y}) = v(\mathrm{y})\}$$

Operations: $[\![\mathrm{op}]\!]$

$$(v, v') \in [\![g]\!] \quad \text{if} \quad v \models g \ \text{ and } \ v' = v$$

$$(v, v') \in [\![x := e]\!] \quad \text{if} \quad \begin{cases} v'(x) = [\![e]\!]_v \\ v'(y) = v'(y) \quad \text{for all } y \neq x \end{cases}$$

Semantics of Operations

The semantics $[\![\mathrm{op}]\!]$ of an operation op is defined as a binary relation between valuations before op and valuations after op:

$$[\![\mathrm{op}]\!] \quad \subseteq \quad (\mathrm{X} \to \mathbb{R}) \times (\mathrm{X} \to \mathbb{R})$$

Examples with $\mathrm{X} = \{\mathrm{x}, \mathrm{y}\}$

$$[\![\mathrm{x}{*}\mathrm{y} \leq 10]\!] \quad = \quad \{(v, v) \mid v(\mathrm{x}) \times v(\mathrm{y}) \leq 10\}$$

$$[\![\mathrm{x} := 3{*}\mathrm{x}]\!] \quad = \quad \{(v, v') \mid v'(\mathrm{x}) = 3 \times v(\mathrm{x}) \ \wedge \ v'(\mathrm{y}) = v(\mathrm{y})\}$$

Operations: $[\![\mathrm{op}]\!]$

$$(v, v') \in [\![g]\!] \quad \textit{if} \quad v \models g \ \textit{ and } \ v' = v$$

$$(v, v') \in [\![x := e]\!] \quad \textit{if} \quad \begin{cases} v'(x) = [\![e]\!]_v \\ v'(y) = v'(y) \quad \text{for all } y \neq x \end{cases}$$

Semantics of Operations

The semantics $[\![\texttt{op}]\!]$ of an operation $\texttt{op}$ is defined as a binary relation between valuations before $\texttt{op}$ and valuations after $\texttt{op}$:

$$[\![\texttt{op}]\!] \quad \subseteq \quad (\mathtt{X} \to \mathbb{R}) \times (\mathtt{X} \to \mathbb{R})$$

Examples with $\mathtt{X} = \{\mathtt{x}, \mathtt{y}\}$

$$[\![\mathtt{x{*}y} \leq \mathtt{10}]\!] \quad = \quad \{(v, v) \mid v(\mathtt{x}) \times v(\mathtt{y}) \leq 10\}$$

$$[\![\mathtt{x} := \mathtt{3{*}x}]\!] \quad = \quad \{(v, v') \mid v'(\mathtt{x}) = 3 \times v(\mathtt{x}) \ \wedge \ v'(\mathtt{y}) = v(\mathtt{y})\}$$

Operations: $[\![\texttt{op}]\!]$

$$(v, v') \in [\![g]\!] \quad \textit{if} \quad v \models g \ \textit{ and } \ v' = v$$

$$(v, v') \in [\![x := e]\!] \quad \textit{if} \quad \begin{cases} v'(x) & = & [\![e]\!]_v \\ v'(y) & = & v'(y) \quad \text{for all } y \neq x \end{cases}$$

Operational Semantics of Control Flow Automata

Definition

The interpretation of a control flow automaton $\langle Q, q_{in}, q_{out}, \mathtt{X}, \rightarrow \rangle$ is the labeled transition system $\langle C, \mathit{Init}, \mathit{Out}, \mathtt{Op}, \rightarrow \rangle$ defined by:

- $C = Q \times (\mathtt{X} \rightarrow \mathbb{R})$
- $\mathit{Init} = \{q_{in}\} \times (\mathtt{X} \rightarrow \mathbb{R})$ and $\mathit{Out} = \{q_{out}\} \times (\mathtt{X} \rightarrow \mathbb{R})$
- $(q, v) \xrightarrow{\mathtt{op}} (q', v')$ if $q \xrightarrow{\mathtt{op}} q'$ and $(v, v') \in [\![\mathtt{op}]\!]$

Two kinds of labeled directed graphs

Control Flow Automata

Use: program source codes

- Syntactic objects
- Finite

Interpretations (LTS)

Use: program behaviors

- Semantic objects
- Uncountably infinite

Operational Semantics of Control Flow Automata

Definition

The interpretation of a control flow automaton $\langle Q, q_{in}, q_{out}, \mathtt{X}, \rightarrow \rangle$ is the labeled transition system $\langle C, \mathit{Init}, \mathit{Out}, \mathtt{Op}, \rightarrow \rangle$ defined by:

- $C = Q \times (\mathtt{X} \rightarrow \mathbb{R})$
- $\mathit{Init} = \{q_{in}\} \times (\mathtt{X} \rightarrow \mathbb{R})$ and $\mathit{Out} = \{q_{out}\} \times (\mathtt{X} \rightarrow \mathbb{R})$
- $(q, v) \xrightarrow{\mathtt{op}} (q', v')$ if $q \xrightarrow{\mathtt{op}} q'$ and $(v, v') \in [\![\mathtt{op}]\!]$

Two kinds of labeled directed graphs

Control Flow Automata

Use: program source codes

- Syntactic objects
- Finite

Interpretations (LTS)

Use: program behaviors

- Semantic objects
- Uncountably infinite

Control Paths, Execution Paths and Runs

A control path is a path in the control flow automaton:

$$q_0 \xrightarrow{\mathtt{op}_0} q_1 \cdots q_{k-1} \xrightarrow{\mathtt{op}_{k-1}} q_k$$

An execution path is a path in the labeled transition system:

$$(q_0, v_0) \xrightarrow{\mathtt{op}_0} (q_1, v_1) \cdots (q_{k-1}, v_{k-1}) \xrightarrow{\mathtt{op}_{k-1}} (q_k, v_k)$$

A run is an execution path that starts with an initial configuration:

$$(q_{in}, v_{in}) \xrightarrow{\mathtt{op}_0} (q_1, v_1) \cdots (q_{k-1}, v_{k-1}) \xrightarrow{\mathtt{op}_{k-1}} (q_k, v_k)$$

Control Paths, Execution Paths and Runs

A control path is a path in the control flow automaton:

$$q_0 \xrightarrow{\mathtt{op}_0} q_1 \cdots q_{k-1} \xrightarrow{\mathtt{op}_{k-1}} q_k$$

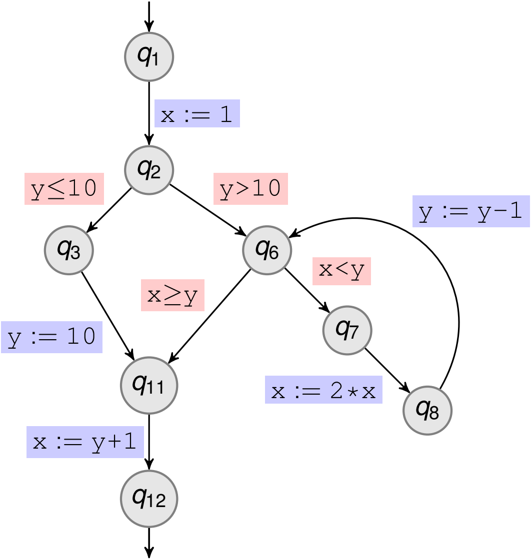
An execution path is a path in the labeled transition system:

$$(q_0, v_0) \xrightarrow{\mathtt{op}_0} (q_1, v_1) \cdots (q_{k-1}, v_{k-1}) \xrightarrow{\mathtt{op}_{k-1}} (q_k, v_k)$$

A run is an execution path that starts with an initial configuration:

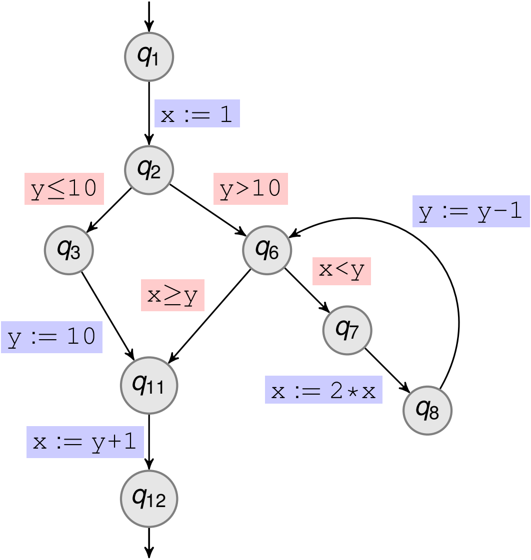
$$(q_{in}, v_{in}) \xrightarrow{\mathtt{op}_0} (q_1, v_1) \cdots (q_{k-1}, v_{k-1}) \xrightarrow{\mathtt{op}_{k-1}} (q_k, v_k)$$

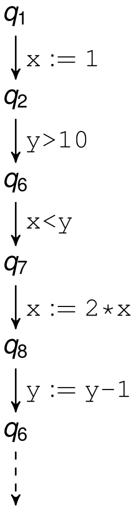
Execution Path: Example



$$(q_1, -159, 27)$$
$$\downarrow \texttt{x := 1}$$
$$(q_2, 1, 27)$$
$$\downarrow \texttt{y>10}$$
$$(q_6, 1, 27)$$
$$\downarrow \texttt{x<y}$$
$$(q_7, 1, 27)$$
$$\downarrow \texttt{x := 2*x}$$
$$(q_8, 2, 27)$$
$$\downarrow \texttt{y := y-1}$$
$$(q_6, 2, 26)$$
$$\vdots$$

Control Path: Example





Outline — Control Flow Automata

Forward Reachability Set Post*

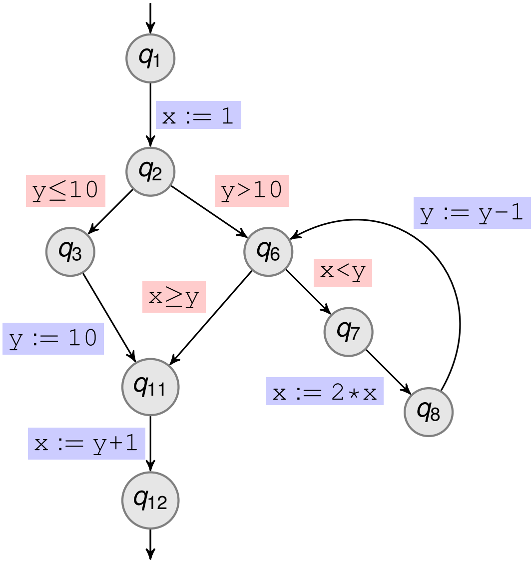
Set of all configurations that are reachable from an initial configuration

$$\mathrm{Post}^* = \bigcup_{\rho\,:\,run} \{(q, v) \mid (q, v) \text{ occurs on } \rho\}$$

$$= \bigcup_{i \in \mathbb{N}} \mathrm{Post}^i(Init)$$

$$= \bigcup_{q_{in} \xrightarrow{\mathtt{op}_0} \cdots \xrightarrow{\mathtt{op}_{k-1}} q} \{q\} \times \left(\llbracket \mathtt{op}_{k-1} \rrbracket \circ \cdots \circ \llbracket \mathtt{op}_0 \rrbracket\right) [(\mathtt{X} \to \mathbb{R})]$$

Forward Reachability Set Post* on Running Example



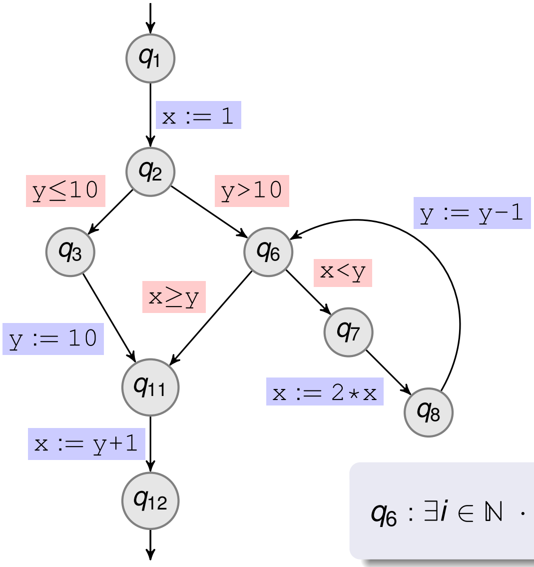
$q_1 : \mathbb{R} \times \mathbb{R}$

$q_2 : \{1\} \times \mathbb{R}$

$q_3 : \{1\} \times]-\infty, 10]$

$q_6 : \{1\} \times]10, +\infty[\ \cup$
$\{2\} \times]9, +\infty[\ \cup$
$\{4\} \times]8, +\infty[\ \cup$
$\ldots$

Forward Reachability Set Post* on Running Example



$q_1 : \mathbb{R} \times \mathbb{R}$

$q_2 : \{1\} \times \mathbb{R}$

$q_3 : \{1\} \times]-\infty, 10]$

$$q_6 : \begin{array}{l} \{1\} \times]10, +\infty[\ \cup \\ \{2\} \times]9, +\infty[\ \cup \\ \{4\} \times]8, +\infty[\ \cup \\ \ldots \end{array}$$

$$q_6 : \exists i \in \mathbb{N} \cdot \begin{cases} x = 2^i \quad \wedge \quad y + i > 10 \quad \wedge \\ i \geq 1 \implies 2^{i-1} < y + 1 \end{cases}$$

Backward Reachability Set Pre*

Set of all configurations that can reach an exit configuration

$$\mathsf{Pre}^* = \bigcup_{i \in \mathbb{N}} \mathsf{Pre}^i(Out)$$

$$= \bigcup_{q \xrightarrow{\mathrm{op}_0} \cdots \xrightarrow{\mathrm{op}_{k-1}} q_{out}} \{q\} \times \left([\![\mathrm{op}_0]\!]^{-1} \circ \cdots \circ [\![\mathrm{op}_{k-1}]\!]^{-1} \right) [(\mathrm{X} \to \mathbb{R})]$$

$$= \bigcup_{q \xrightarrow{\mathrm{op}_0} \cdots \xrightarrow{\mathrm{op}_{k-1}} q_{out}} \{q\} \times \left(([\![\mathrm{op}_{k-1}]\!] \circ \cdots \circ [\![\mathrm{op}_0]\!])^{-1} \right) [(\mathrm{X} \to \mathbb{R})]$$

Verification of Control Flow Automata

Goal (Repetition)

Check that there is no run that visits a location q contained in a given set $Q_{BAD} \subseteq Q$ of bad locations.

Define the set *Bad* of bad configurations by: $Bad = Q_{BAD} \times (\mathbb{X} \to \mathbb{R})$.

Goal (Equivalent Formulation)

Check that Post* is disjoint from *Bad*

Undecidability

The *location reachability* and *configuration reachability* problems are both undecidable for control flow automata.

Proof by reduction to location reachability in two-counters machines.

Two-Counters Machines as Control Flow Automata

Two-Counters (Minsky) Machines

Finite-state automaton extended with:

- two counters over nonnegative integers
- test for zero, increment and guarded decrement

Reachability is undecidable for this class.

Any two-counters machine can (effectively) be represented as a control flow automaton in this restricted class:

- two variables: $\mathrm{X} = \{c_1, c_2\}$
- allowed guards: $x = 0$ and $x \neq 0$ for each $x \in \mathrm{X}$
- allowed assignments: $x := x+1$ and $x := x-1$ for each $x \in \mathrm{X}$

Two-Counters Machines as Control Flow Automata

Two-Counters (Minsky) Machines

Finite-state automaton extended with:

- two counters over nonnegative integers
- test for zero, increment and guarded decrement

Reachability is undecidable for this class.

Any two-counters machine can (effectively) be represented as a control flow automaton in this restricted class:

- two variables: $\mathtt{X} = \{\mathtt{c}_1, \mathtt{c}_2\}$
- allowed guards: $x = 0$ and $x \neq 0$ for each $x \in \mathtt{X}$
- allowed assignments: $x := x{+}1$ and $x := x{-}1$ for each $x \in \mathtt{X}$

Tentative Solution: Approximation Techniques

Definition

An invariant is any set $Inv \subseteq C$ such that $\mathrm{Post}^* \subseteq Inv$.

Idea:

1. Compute an invariant Inv (easier to compute than Post^*)
2. If Inv is disjoint from Bad then Post^* is also disjoint from Bad

Rest of the lecture:

Computation of precise enough invariants

Tentative Solution: Approximation Techniques

Definition

An invariant is any set $Inv \subseteq C$ such that $\text{Post}^* \subseteq Inv$.

Idea:

1. Compute an invariant Inv (easier to compute than Post^*)
2. If Inv is disjoint from Bad then Post^* is also disjoint from Bad

Rest of the lecture:

Computation of precise enough invariants

Summary

- Computational model for programs: control flow automata
  - syntax
  - semantics

- Undecidability in general of model-checking for control flow automata

- Tentative solution: computation of invariants

Part III

Data Flow Analysis

Outline — Data Flow Analysis

Outline — Data Flow Analysis

Short Introduction to Data Flow Analysis

Tentative Definition

Compile-time techniques to gather run-time information about data in programs without actually running them

Applications

Code optimization

- Avoid *redundant* computations (e.g. reuse available results)
- Avoid *superfluous* computations (e.g. eliminate dead code)

Code validation

- Invariant generation

Conservative approximations

Live Variables Analysis: Definition

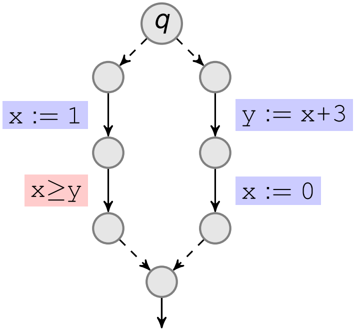
Definition

A variable *x* is live at location *q* if there exists a control path starting from *q* where *x* is used before it is modified.

Live Variables Analysis: Definition

Definition

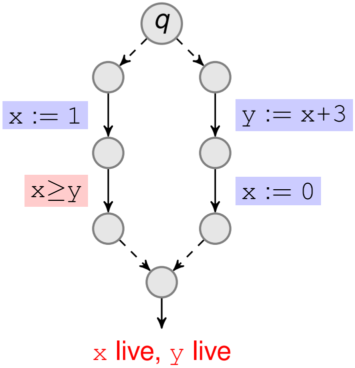
A variable *x* is live at location *q* if there exists a control path starting from *q* where *x* is used before it is modified.



Live Variables Analysis: Definition

Definition

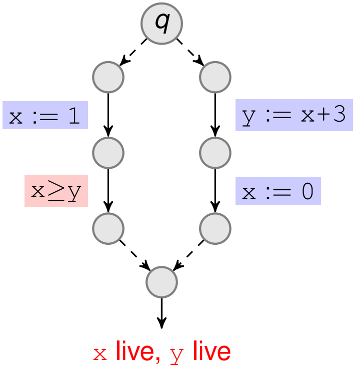
A variable *x* is live at location *q* if there exists a control path starting from *q* where *x* is used before it is modified.

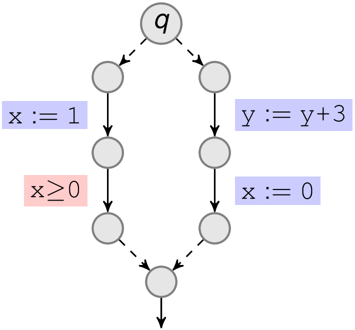


Live Variables Analysis: Definition

Definition

A variable *x* is live at location *q* if there exists a control path starting from *q* where *x* is used before it is modified.

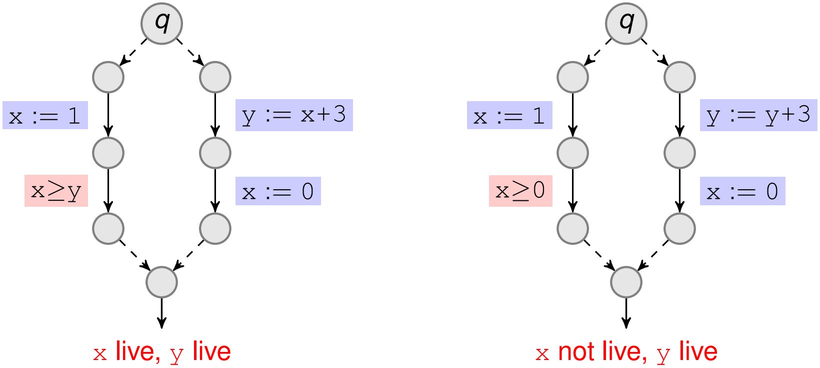




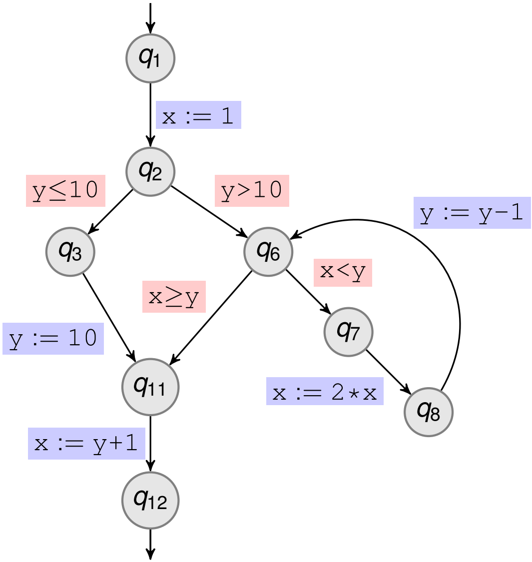
Live Variables Analysis: Definition

Definition

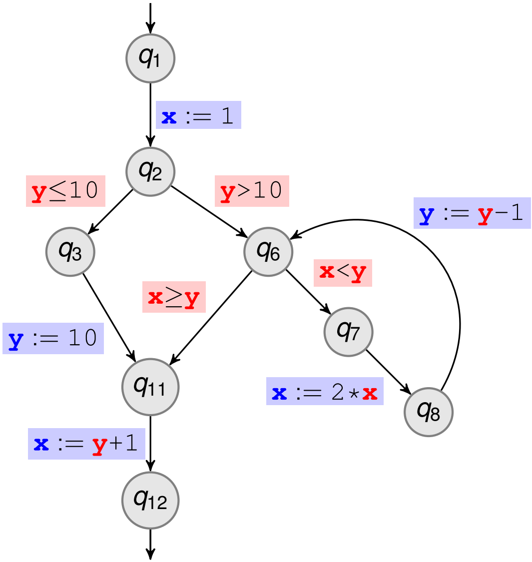
A variable x is live at location q if there exists a control path starting from q where x is used before it is modified.



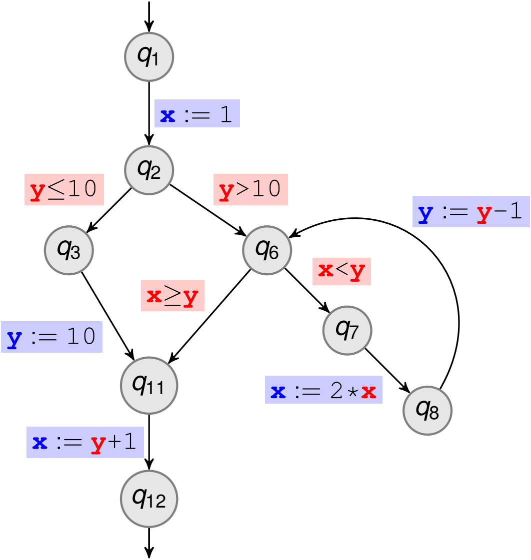
Live Variables Analysis: Running Example



Live Variables Analysis: Running Example



Live Variables Analysis: Running Example



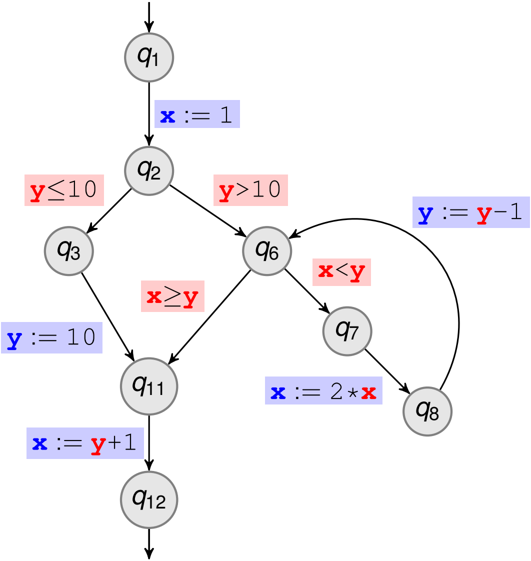
0 : Initialization

1 : Local information

2 : Propagation (←)

	x	y
q_1		
q_2		
q_3		
q_6		
q_7		
q_8		
q_{11}		
q_{12}		

Live Variables Analysis: Running Example



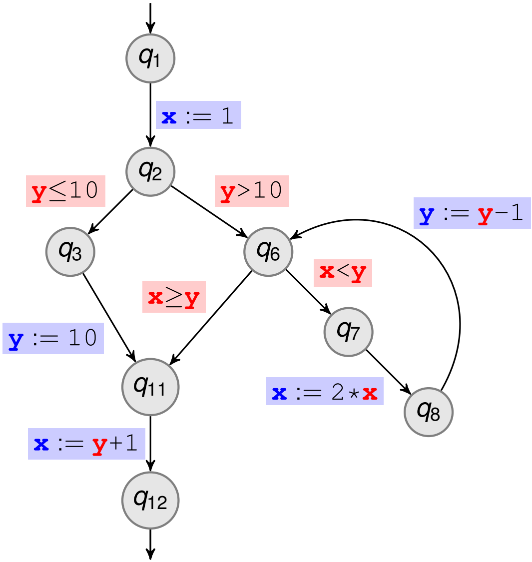
0 : Initialization

1 : Local information

2 : Propagation (⟵)

	x	y
q_1		
q_2		•
q_3		
q_6	•	•
q_7	•	
q_8		•
q_{11}		•
q_{12}		

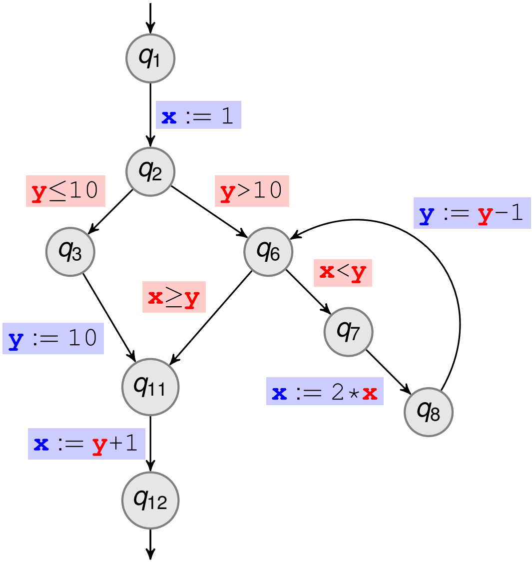
Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		
q_2	●	●
q_3		
q_6	●	●
q_7	●	
q_8		●
q_{11}		●
q_{12}		

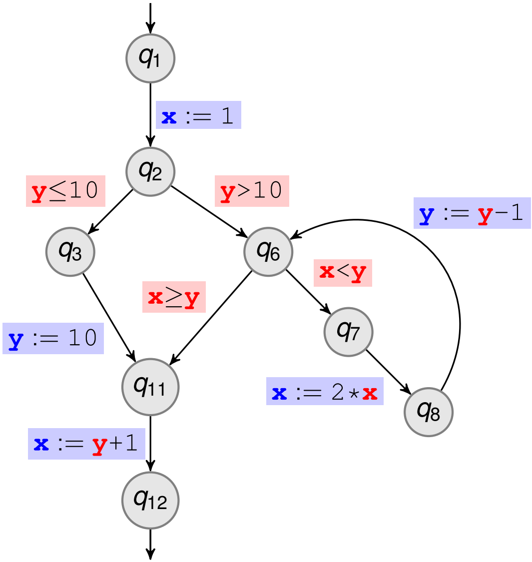
Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		
q_2	•	•
q_3		
q_6	•	•
q_7	•	
q_8		•
q_{11}		•
q_{12}		

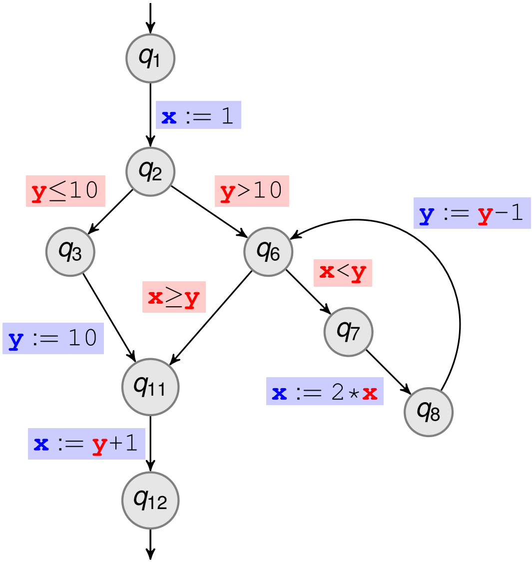
Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		●
q_2	●	●
q_3		
q_6	●	●
q_7	●	
q_8		●
q_{11}		●
q_{12}		

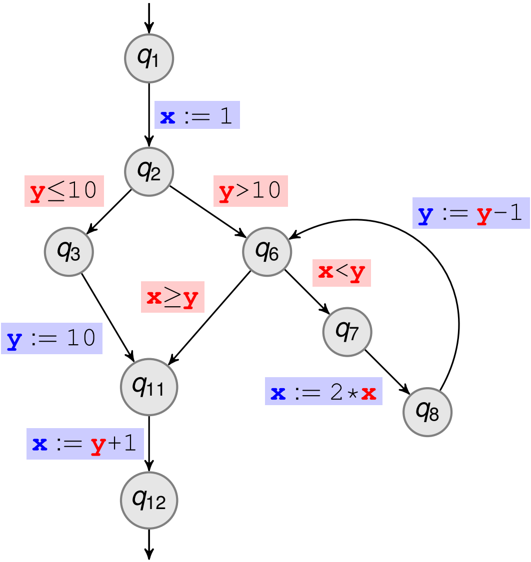
Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		•
q_2	•	•
q_3		
q_6	•	•
q_7	•	
q_8		•
q_{11}		•
q_{12}		

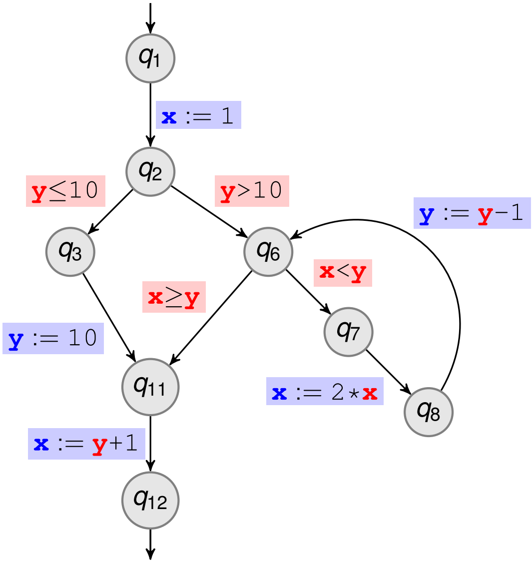
Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		•
q_2	•	•
q_3		
q_6	•	•
q_7	•	
q_8	•	•
q_{11}		•
q_{12}		

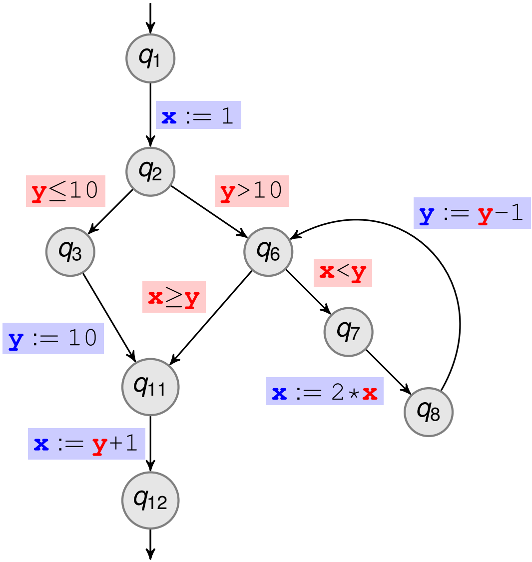
Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		•
q_2	•	•
q_3		
q_6	•	•
q_7	•	
q_8	•	•
q_{11}		•
q_{12}		

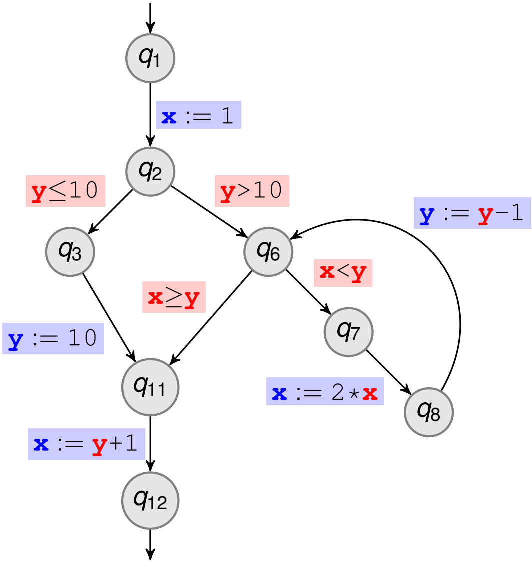
Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		●
q_2	●	●
q_3		
q_6	●	●
q_7	●	●
q_8	●	●
q_{11}		●
q_{12}		

Live Variables Analysis: Running Example



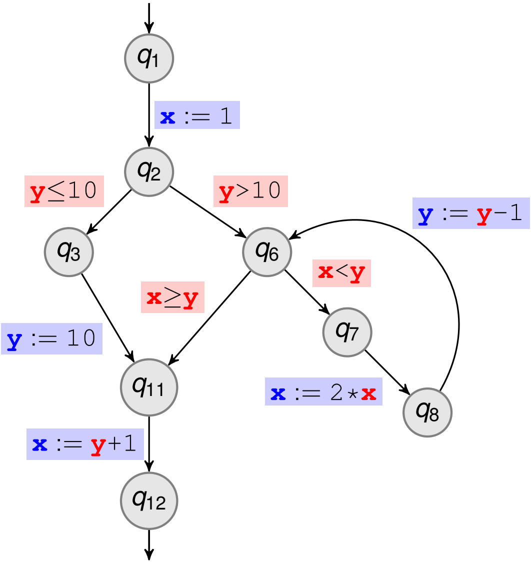
0 : Initialization

1 : Local information

2 : Propagation (←)

	x	y
q_1		•
q_2	•	•
q_3		
q_6	•	•
q_7	•	•
q_8	•	•
q_{11}		•
q_{12}		

Live Variables Analysis: Running Example



0 : Initialization
1 : Local information
2 : Propagation (←)

	x	y
q_1		•
q_2	•	•
q_3		
q_6	•	•
q_7	•	•
q_8	•	•
q_{11}		•
q_{12}		

Live Variables Analysis: Formulation

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

System of equations: variables L_q for $q \in Q$, with $L_q \subseteq \mathrm{X}$

$$L_q \;=\; \bigcup_{q \xrightarrow{\mathrm{op}} q'} Gen_{\mathrm{op}} \cup (L_{q'} \setminus Kill_{\mathrm{op}}) \qquad\qquad L(q_{out}) \;=\; \emptyset$$

$$Gen_{\mathrm{op}} = \begin{cases} Var(g) & \text{if } \mathrm{op} = g \\ Var(e) & \text{if } \mathrm{op} = x := e \end{cases} \qquad Kill_{\mathrm{op}} = \begin{cases} \emptyset & \text{if } \mathrm{op} = g \\ \{x\} & \text{if } \mathrm{op} = x := e \end{cases}$$

$$f_{\mathrm{op}}(X) \;=\; Gen_{\mathrm{op}} \cup (X \setminus Kill_{\mathrm{op}})$$

Live Variables Analysis: Formulation

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

System of equations: variables L_q for $q \in Q$, with $L_q \subseteq \mathrm{X}$

$$L_q = \bigcup_{q \xrightarrow{\mathrm{op}} q'} f_{\mathrm{op}}(L_{q'}) \qquad L(q_{out}) = \emptyset$$

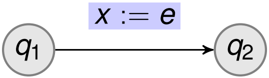
$$Gen_{\mathrm{op}} = \begin{cases} Var(g) & \text{if } \mathrm{op} = g \\ Var(e) & \text{if } \mathrm{op} = x := e \end{cases} \qquad Kill_{\mathrm{op}} = \begin{cases} \emptyset & \text{if } \mathrm{op} = g \\ \{x\} & \text{if } \mathrm{op} = x := e \end{cases}$$

$$f_{\mathrm{op}}(X) = Gen_{\mathrm{op}} \cup (X \setminus Kill_{\mathrm{op}})$$

Live Variables Analysis: Applications

Code Optimization

Dead code elimination



If x is not live at location q_2 then we may remove the assignment $x := e$ on the edge from q_1 to q_2.

This is sound since the analysis is conservative

Available Expressions Analysis: Definition

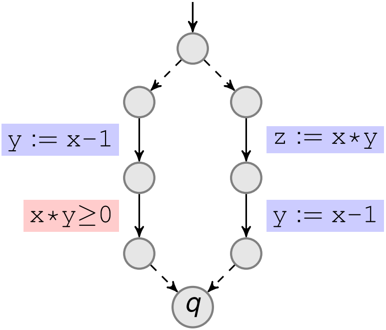
Definition

A expression e is available at location q if every control path from q_{in} to q contains an evaluation of e which is not followed by an assignment of any variable x occurring in e.

Available Expressions Analysis: Definition

Definition

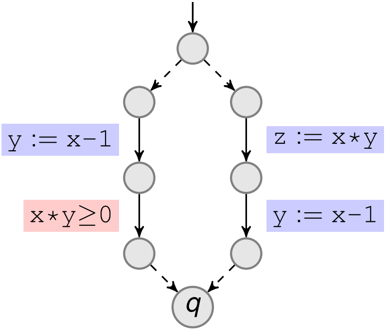
A expression *e* is available at location *q* if every control path from q_{in} to *q* contains an evaluation of *e* which is not followed by an assignment of any variable *x* occurring in *e*.



Available Expressions Analysis: Definition

Definition

A expression *e* is available at location *q* if every control path from q_{in} to *q* contains an evaluation of *e* which is not followed by an assignment of any variable *x* occurring in *e*.

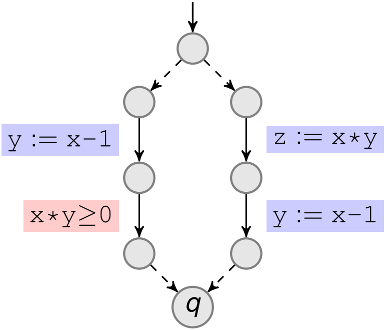


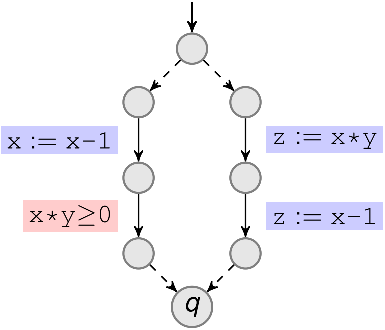
x−1 available, x*y not available

Available Expressions Analysis: Definition

Definition

A expression e is available at location q if every control path from q_{in} to q contains an evaluation of e which is not followed by an assignment of any variable x occurring in e.



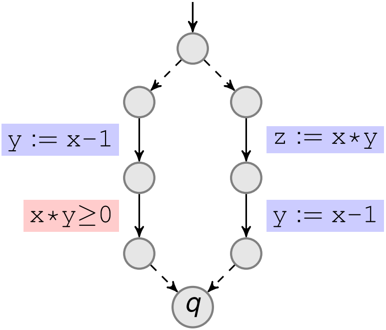


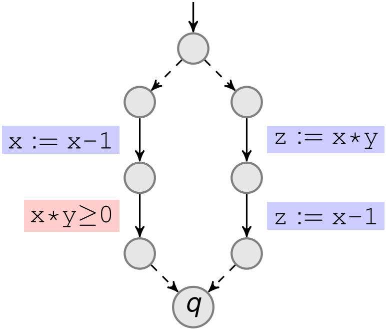
x−1 available, x∗y not available

Available Expressions Analysis: Definition

Definition

A expression e is available at location q if every control path from q_{in} to q contains an evaluation of e which is not followed by an assignment of any variable x occurring in e.

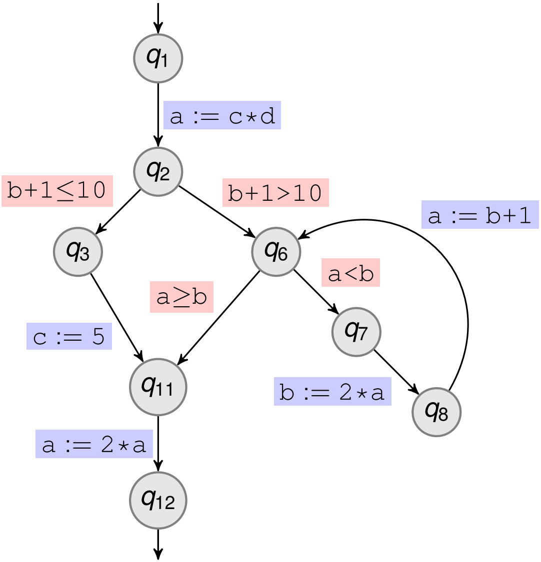




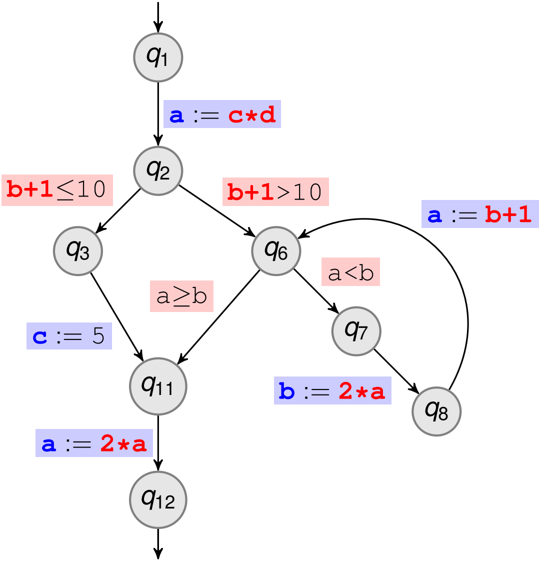
`x−1` available, `x∗y` not available

`x−1` not available, `x∗y` available

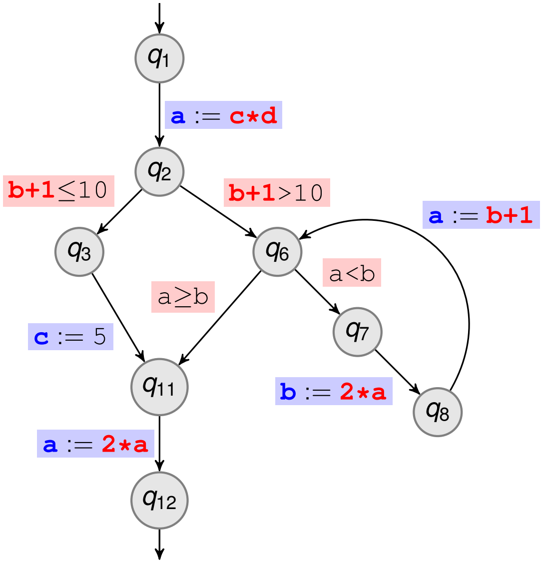
Available Expressions Analysis: Other Example



Available Expressions Analysis: Other Example



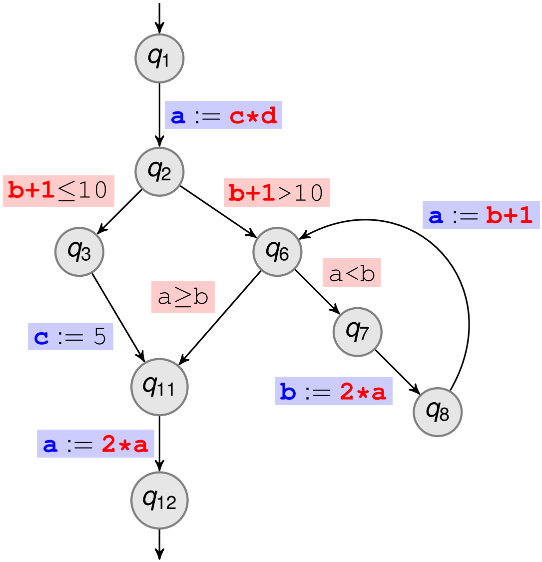
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	•	•	•
q_3	•	•	•
q_6	•	•	•
q_7	•	•	•
q_8	•	•	•
q_{11}	•	•	•
q_{12}	•	•	•

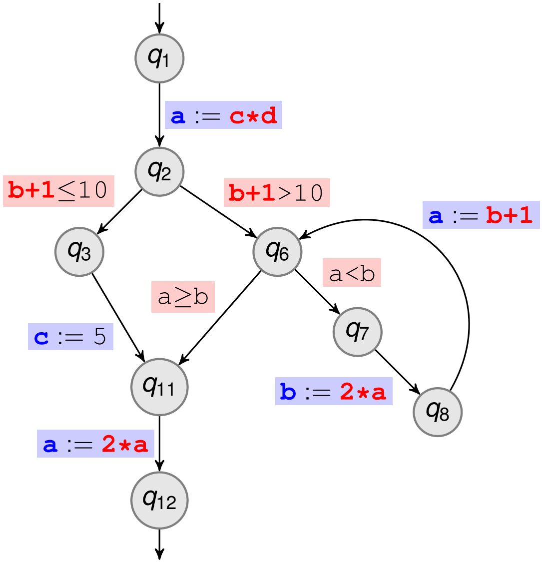
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	●	●	
q_3	●	●	●
q_6	●	●	
q_7	●	●	●
q_8	●		●
q_{11}		●	●
q_{12}	●	●	

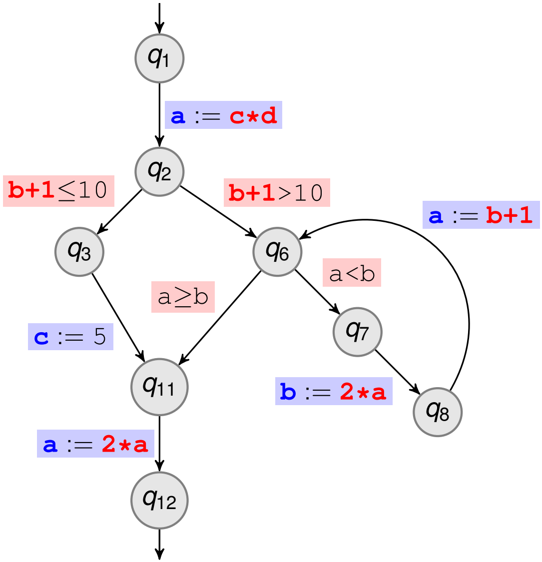
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation ($\rightarrow$)

	c*d	b+1	2*a
q_1			
q_2	●	●	
q_3	●	●	●
q_6	●	●	
q_7	●	●	●
q_8	●		●
q_{11}		●	●
q_{12}	●	●	

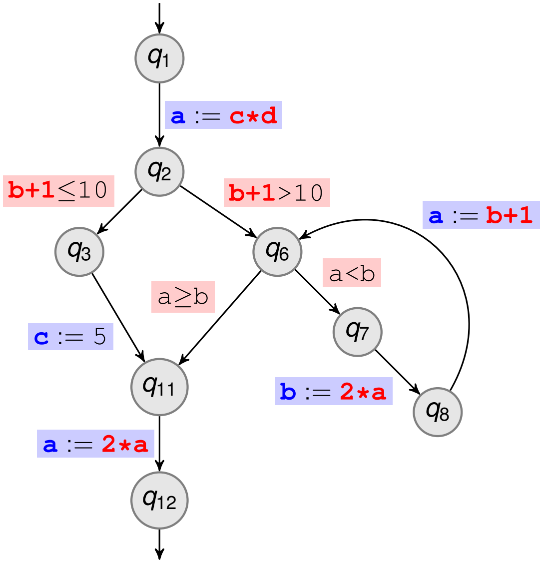
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	•		
q_3	•	•	•
q_6	•	•	
q_7	•	•	•
q_8	•		•
q_{11}		•	•
q_{12}	•	•	

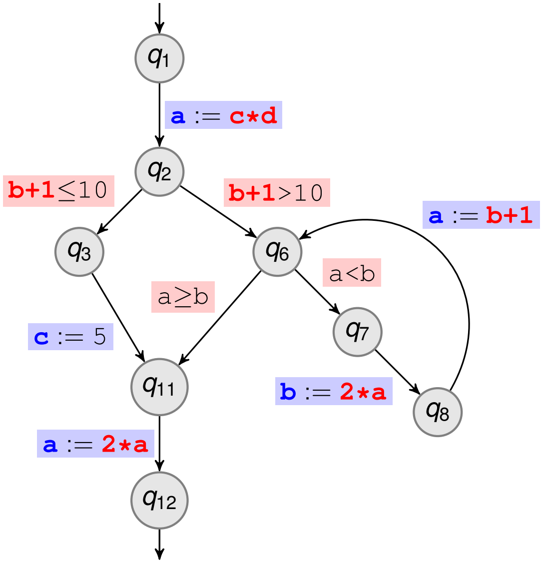
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	●		
q_3	●	●	●
q_6	●	●	
q_7	●	●	●
q_8	●		●
q_{11}		●	●
q_{12}	●	●	

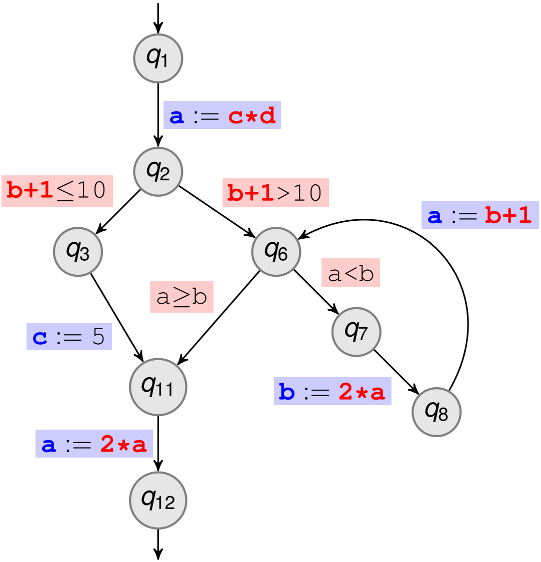
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

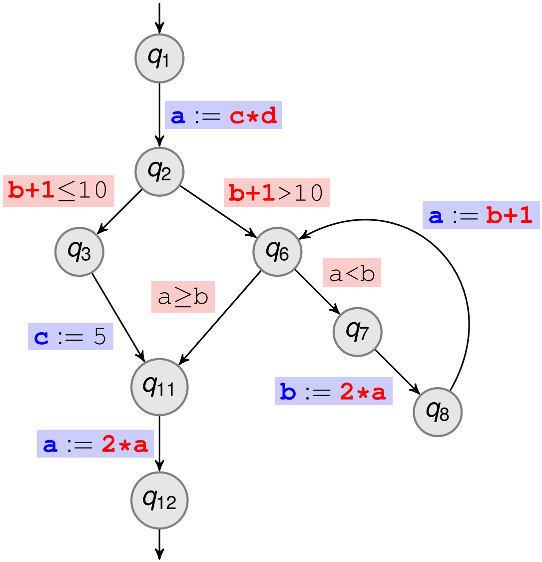
	c*d	b+1	2*a
q_1			
q_2	•		
q_3	•	•	
q_6	•	•	
q_7	•	•	•
q_8	•		•
q_{11}		•	•
q_{12}	•	•	

Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

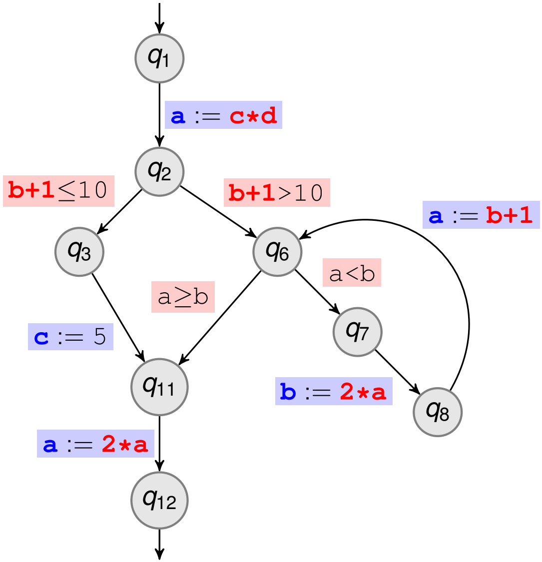
	c*d	b+1	2*a
q_1			
q_2	●		
q_3	●	●	
q_6	●	●	
q_7	●	●	●
q_8	●		●
q_{11}		●	●
q_{12}	●	●	



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	•		
q_3	•	•	
q_6	•	•	
q_7	•	•	•
q_8	•		•
q_{11}		•	
q_{12}	•	•	

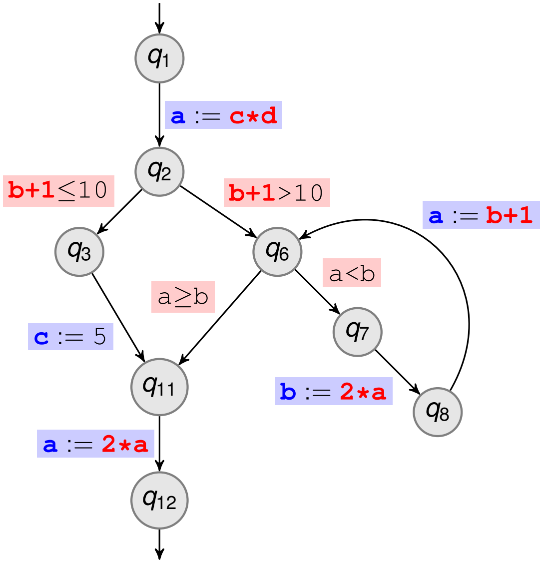
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	•		
q_3	•	•	
q_6	•	•	
q_7	•	•	•
q_8	•		•
q_{11}		•	
q_{12}	•	•	

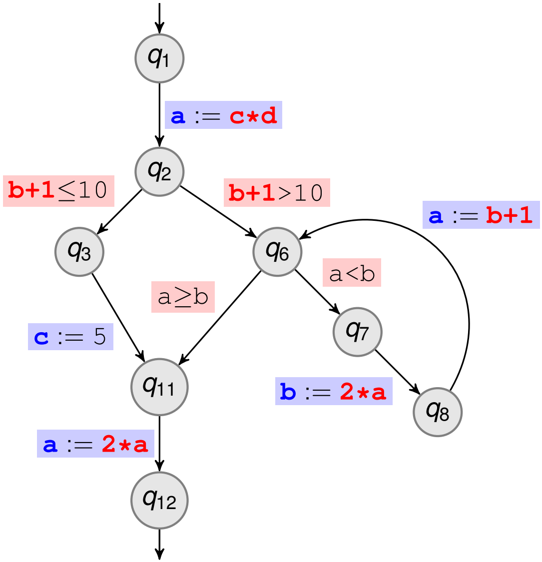
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	•		
q_3	•	•	
q_6	•	•	
q_7	•	•	•
q_8	•		•
q_{11}		•	
q_{12}		•	

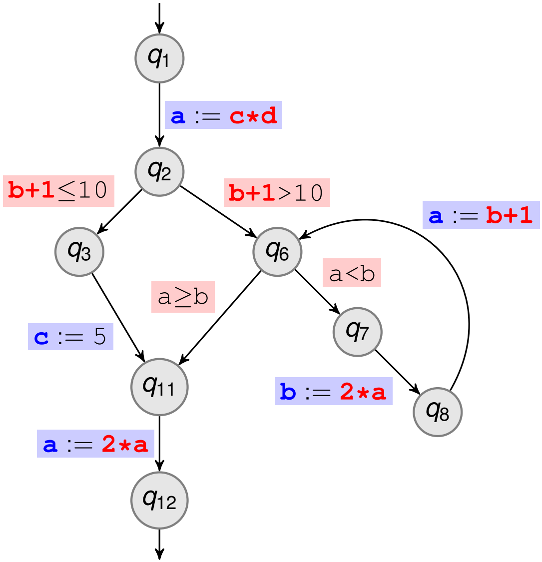
Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

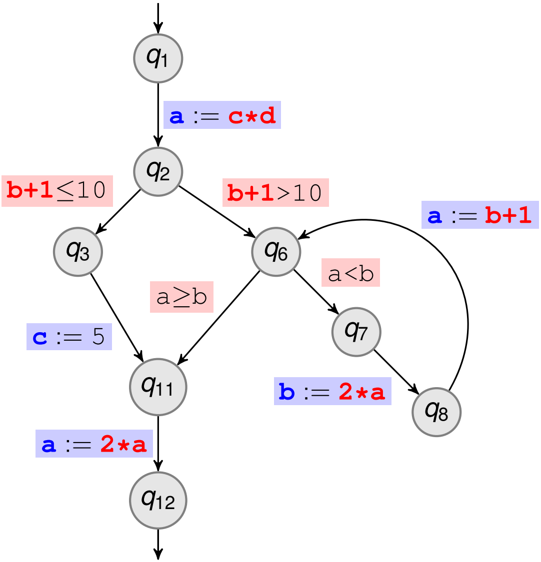
	c*d	b+1	2*a
q_1			
q_2	●		
q_3	●	●	
q_6	●	●	
q_7	●	●	●
q_8	●		●
q_{11}		●	
q_{12}		●	

Available Expressions Analysis: Other Example



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	•		
q_3	•	•	
q_6	•	•	
q_7	•	•	
q_8	•		•
q_{11}		•	
q_{12}		•	



0 : Initialization
1 : Local information
2 : Propagation (→)

	c*d	b+1	2*a
q_1			
q_2	•		
q_3	•	•	
q_6	•	•	
q_7	•	•	
q_8	•		•
q_{11}		•	
q_{12}		•	

Available Expressions Analysis: Formulation

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

System of equations: variables A_q, with $A_q \subseteq SubExp(\rightarrow)$

$$A_q = \bigcap_{q' \xrightarrow{\mathrm{op}} q} Gen_{\mathrm{op}} \cup (A_{q'} \setminus Kill_{\mathrm{op}}) \qquad A(q_{in}) = \emptyset$$

$$Gen_{\mathrm{op}} = \begin{cases} SubExp(g) & \text{if } \mathrm{op} = g \\ \{f \in SubExp(e) \mid x \notin SubExp(e)\} & \text{if } \mathrm{op} = x := e \end{cases}$$

$$Kill_{\mathrm{op}} = \begin{cases} \emptyset & \text{if } \mathrm{op} = g \\ \{e \in SubExp(\rightarrow) \mid x \in Var(e)\} & \text{if } \mathrm{op} = x := e \end{cases}$$

$$f_{\mathrm{op}}(X) = Gen_{\mathrm{op}} \cup (X \setminus Kill_{\mathrm{op}})$$

Available Expressions Analysis: Formulation

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

System of equations: variables A_q, with $A_q \subseteq SubExp(\rightarrow)$

$$A_q \quad = \quad \bigcap_{q' \xrightarrow{\mathrm{op}} q} f_{\mathrm{op}}\left(A_{q'}\right) \qquad\qquad A(q_{in}) \;=\; \emptyset$$

$$Gen_{\mathrm{op}} \;=\; \begin{cases} SubExp(g) & \text{if } \mathrm{op} = g \\ \{f \in SubExp(e) \mid x \notin SubExp(e)\} & \text{if } \mathrm{op} = x := e \end{cases}$$

$$Kill_{\mathrm{op}} \;=\; \begin{cases} \emptyset & \text{if } \mathrm{op} = g \\ \{e \in SubExp(\rightarrow) \mid x \in Var(e)\} & \text{if } \mathrm{op} = x := e \end{cases}$$

$$f_{\mathrm{op}}(X) \;=\; Gen_{\mathrm{op}} \;\cup\; (X \setminus Kill_{\mathrm{op}})$$

Available Expressions Analysis: Applications

Code Optimization

Avoid recomputation of an expression

If e is available at location q_1 then we may reuse its value to evaluate the operation on the edge from q_1 to q_2.

This is sound since the analysis is conservative

Constant Propagation Analysis: Definition

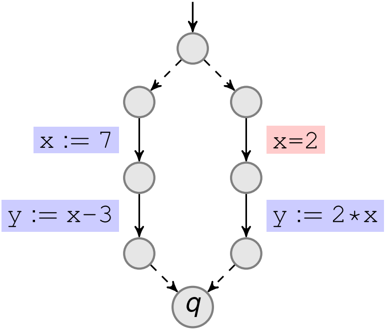
Definition

A variable x is constant at location q if we have $v(x) = v'(x)$ for any two reachable configurations (q, v) and (q, v') in Post^*.

Constant Propagation Analysis: Definition

Definition

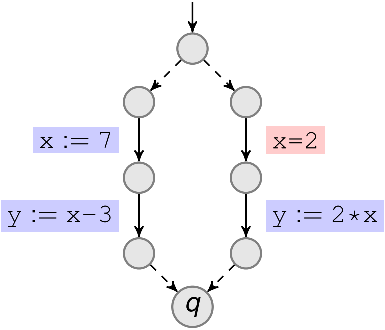
A variable x is constant at location q if we have $v(x) = v'(x)$ for any two reachable configurations (q, v) and (q, v') in Post*.



Constant Propagation Analysis: Definition

Definition

A variable x is constant at location q if we have $v(x) = v'(x)$ for any two reachable configurations (q, v) and (q, v') in Post^*.

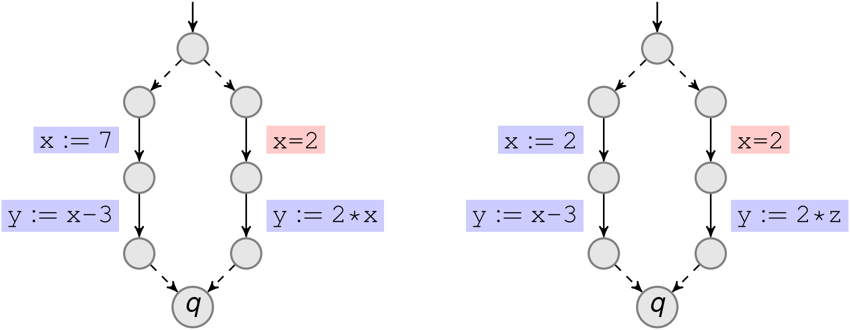


x not constant, y constant

Constant Propagation Analysis: Definition

Definition

A variable x is constant at location q if we have $v(x) = v'(x)$ for any two reachable configurations (q, v) and (q, v') in Post^*.

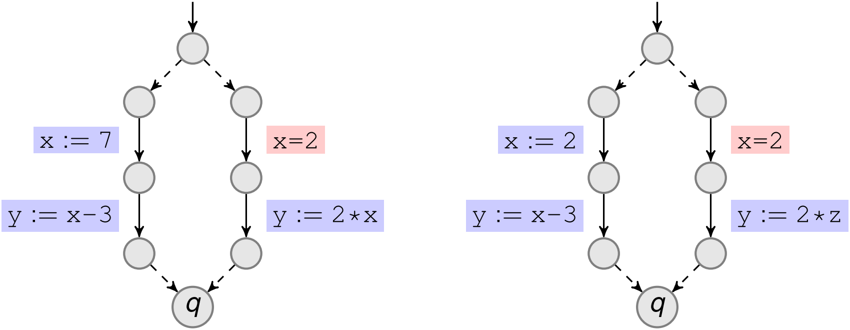


x not constant, y constant

Constant Propagation Analysis: Definition

Definition

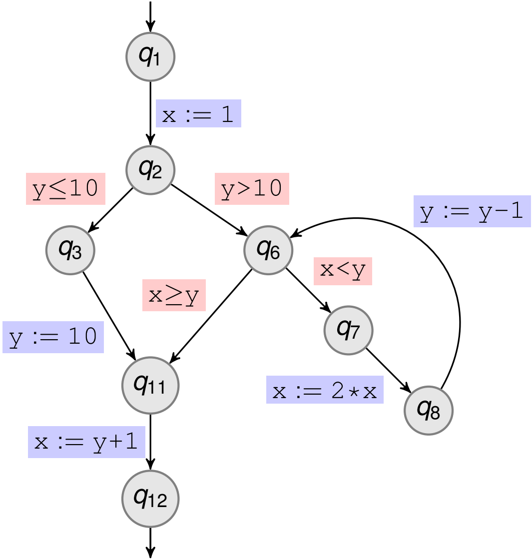
A variable x is constant at location q if we have $v(x) = v'(x)$ for any two reachable configurations (q, v) and (q, v') in Post^*.



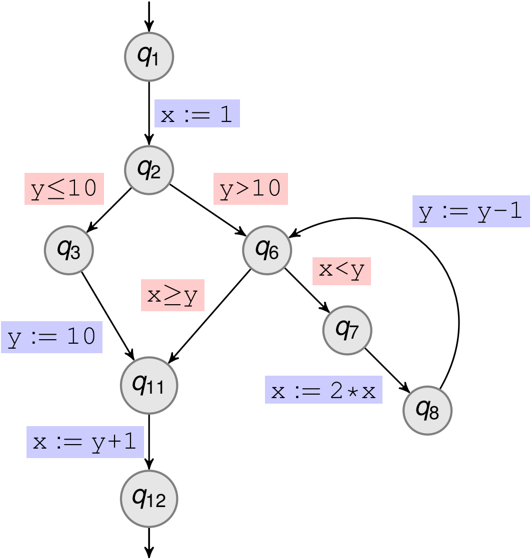
x not constant, y constant

x constant, y not constant

Constant Propagation Analysis: Running Example



Constant Propagation Analysis: Running Example

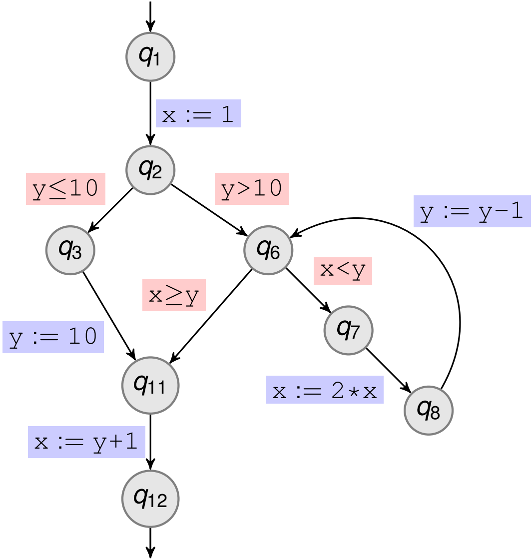


0 : Initialization

1 : Propagation ($\longrightarrow$)

	x	y
q_1	$\top$	$\top$
q_2		
q_3		
q_6		
q_7		
q_8		
q_{11}		
q_{12}		

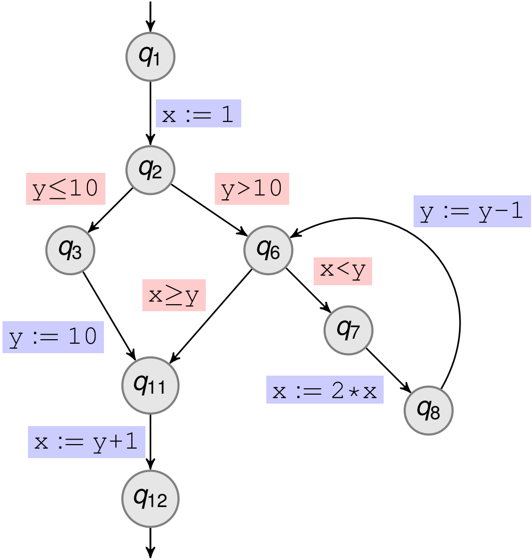
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	⊤	⊤
q_2	1	⊤
q_3		
q_6		
q_7		
q_8		
q_{11}		
q_{12}		

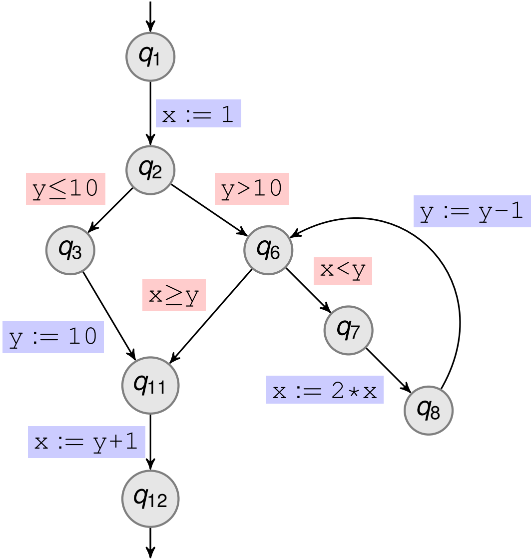
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3		
q_6		
q_7		
q_8		
q_{11}		
q_{12}		

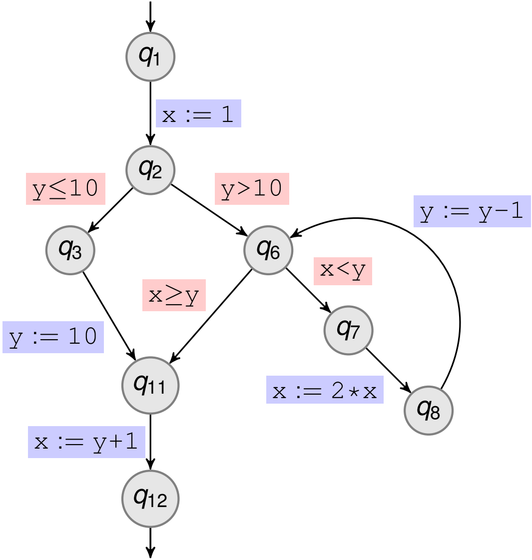
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6		
q_7		
q_8		
q_{11}		
q_{12}		

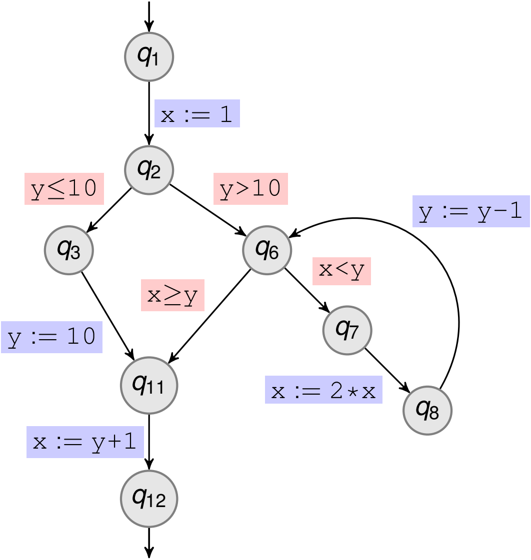
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6		
q_7		
q_8		
q_{11}		
q_{12}		

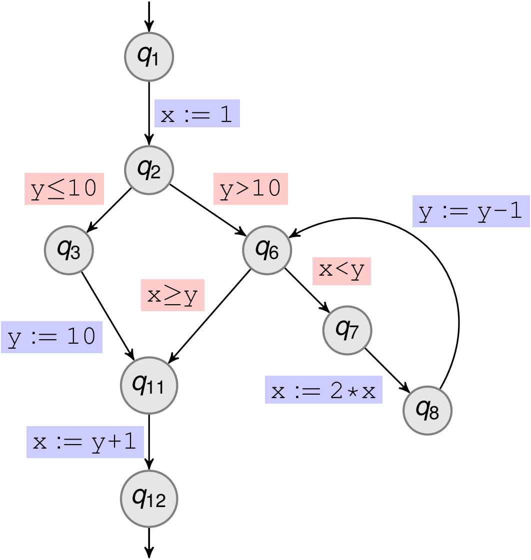
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	⊤	⊤
q_2	1	⊤
q_3	1	⊤
q_6		
q_7		
q_8		
q_{11}	1	10
q_{12}		

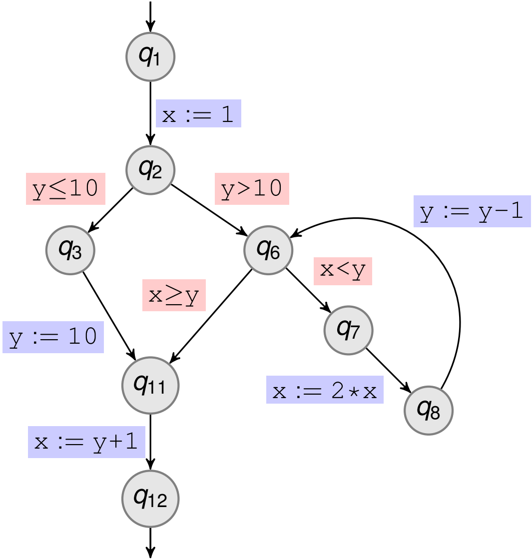
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6		
q_7		
q_8		
q_{11}	1	10
q_{12}		

Constant Propagation Analysis: Running Example

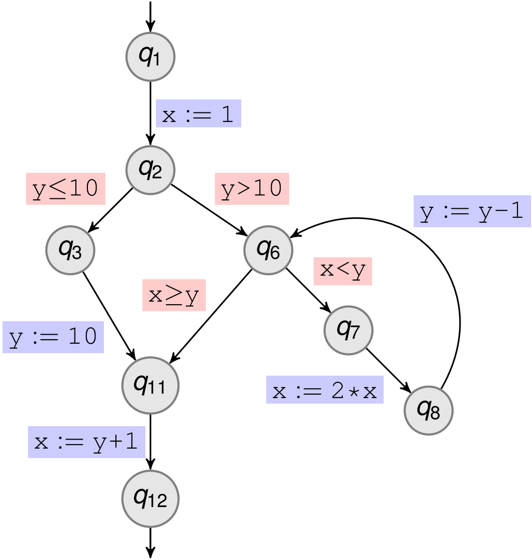


0 : Initialization

1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6		
q_7		
q_8		
q_{11}	1	10
q_{12}	11	10

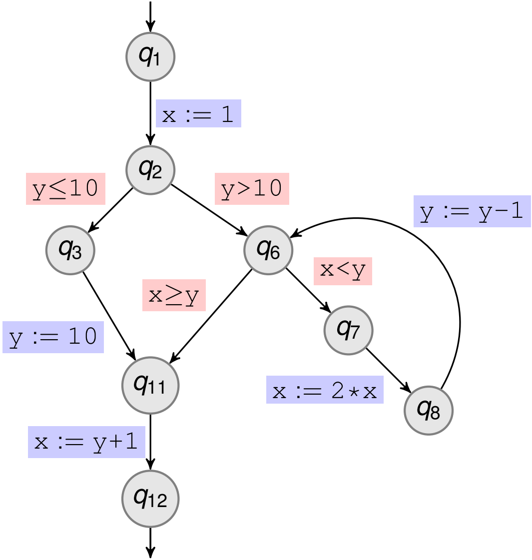
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6		
q_7		
q_8		
q_{11}	1	10
q_{12}	11	10

Constant Propagation Analysis: Running Example

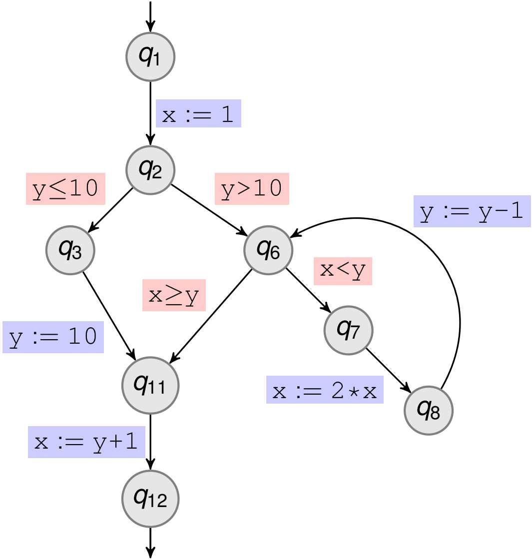


0 : Initialization

1 : Propagation (⟶)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7		
q_8		
q_{11}	1	10
q_{12}	11	10

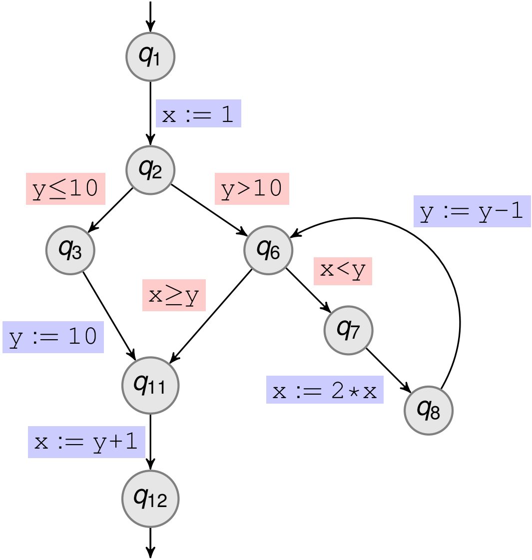
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7		
q_8		
q_{11}	1	10
q_{12}	11	10

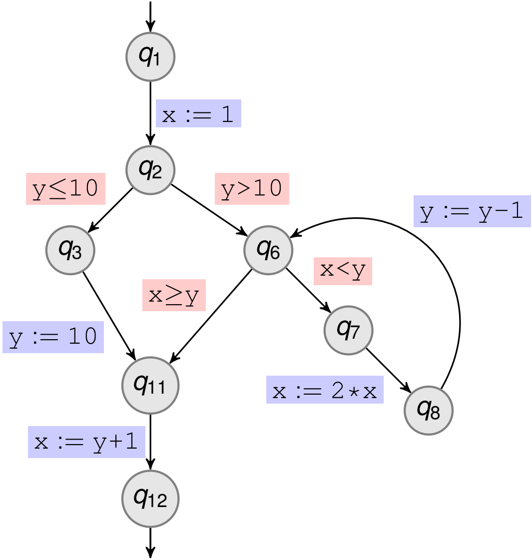
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7		
q_8		
q_{11}	1	10, $\top$
q_{12}	11	10

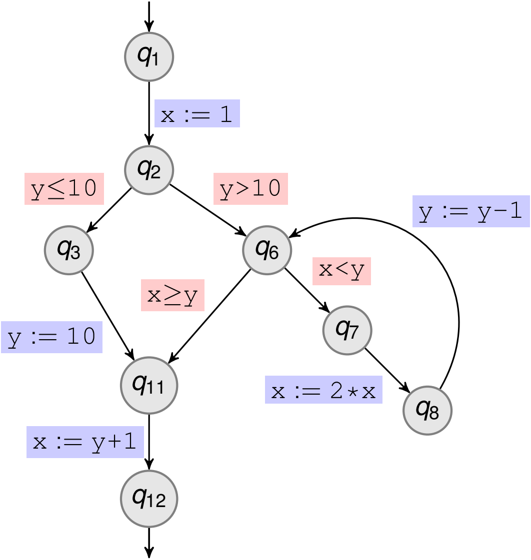
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

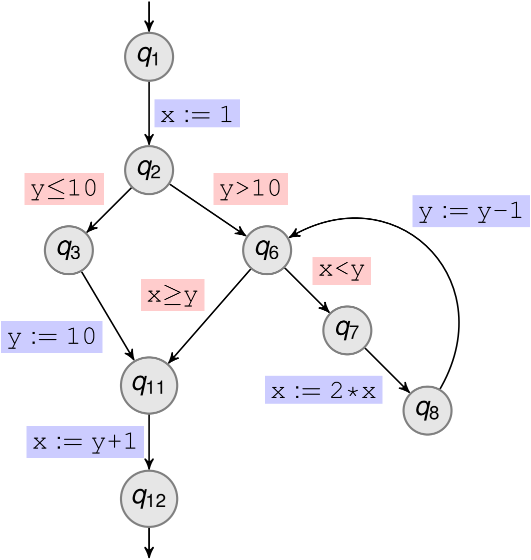
	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7		
q_8		
q_{11}	1	$\top$
q_{12}	11	10

Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

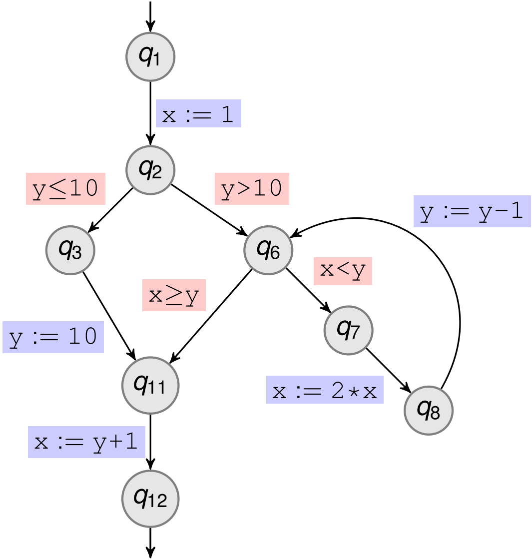
	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7		
q_8		
q_{11}	1	$\top$
q_{12}	$11, 2$	$10, \top$



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7		
q_8		
q_{11}	1	$\top$
q_{12}	$\top$	$\top$

Constant Propagation Analysis: Running Example

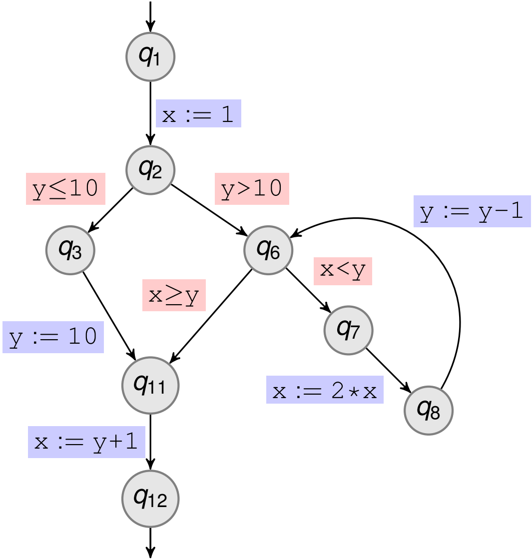


0 : Initialization

1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7	1	$\top$
q_8		
q_{11}	1	$\top$
q_{12}	$\top$	$\top$

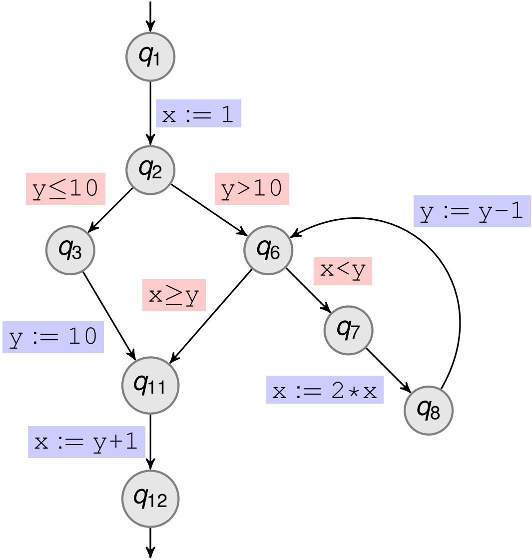
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7	1	$\top$
q_8		
q_{11}	1	$\top$
q_{12}	$\top$	$\top$

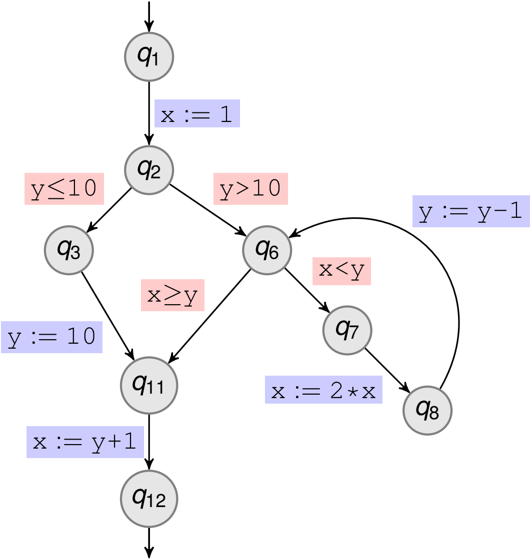
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7	1	$\top$
q_8	2	$\top$
q_{11}	1	$\top$
q_{12}	$\top$	$\top$

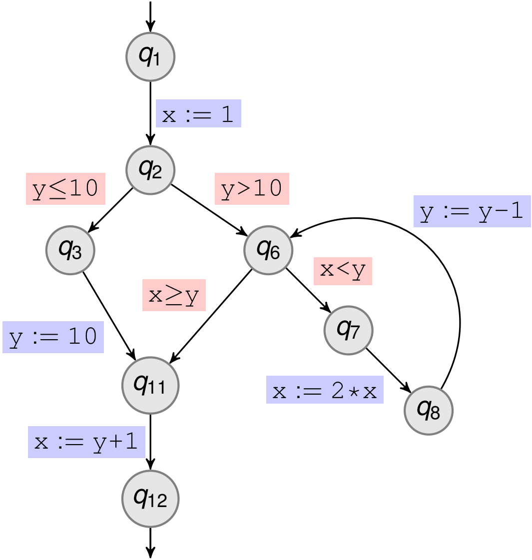
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1	$\top$
q_7	1	$\top$
q_8	2	$\top$
q_{11}	1	$\top$
q_{12}	$\top$	$\top$

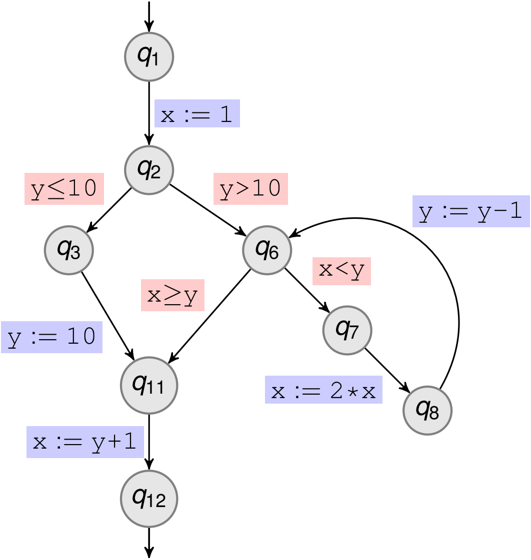
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	1, 2	$\top$
q_7	1	$\top$
q_8	2	$\top$
q_{11}	1	$\top$
q_{12}	$\top$	$\top$

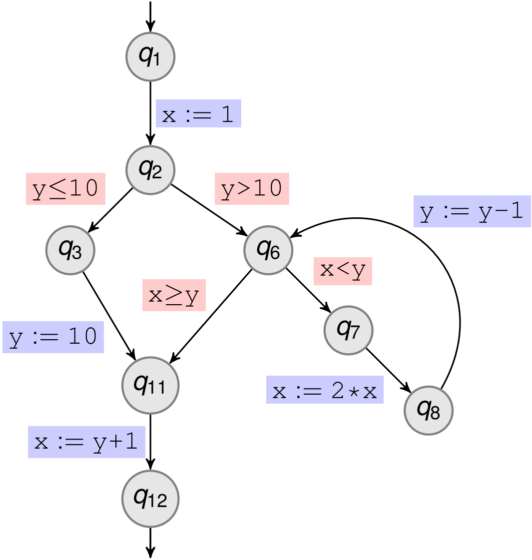
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	$\top$	$\top$
q_7	1	$\top$
q_8	2	$\top$
q_{11}	1	$\top$
q_{12}	$\top$	$\top$

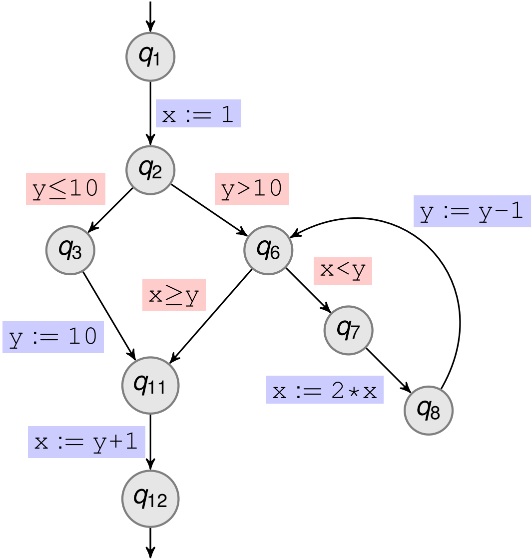
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	$\top$	$\top$
q_7	1	$\top$
q_8	2	$\top$
q_{11}	1, $\top$	$\top$
q_{12}	$\top$	$\top$

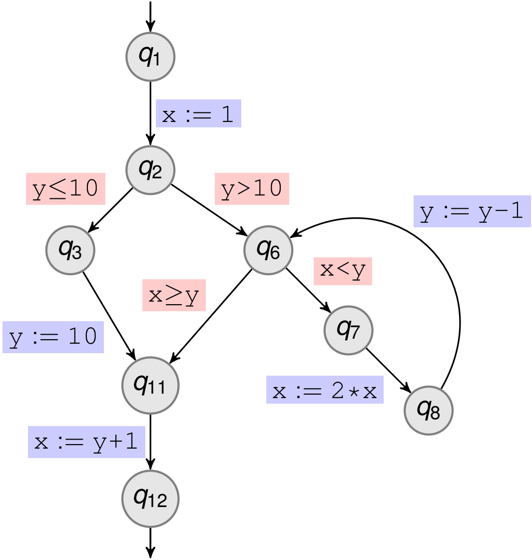
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	$\top$	$\top$
q_7	1	$\top$
q_8	2	$\top$
q_{11}	$\top$	$\top$
q_{12}	$\top$	$\top$

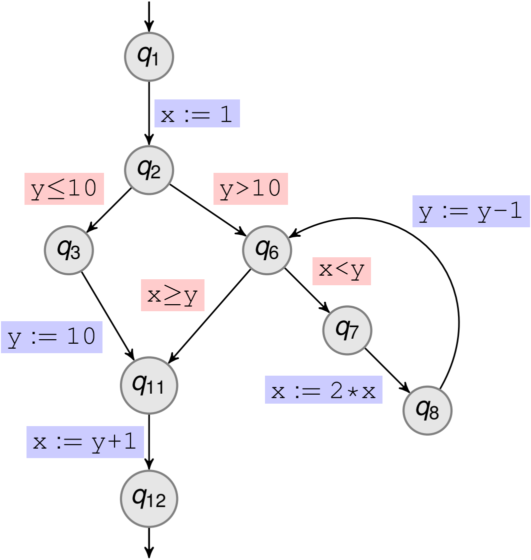
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	$\top$	$\top$
q_7	1, $\top$	$\top$
q_8	2	$\top$
q_{11}	$\top$	$\top$
q_{12}	$\top$	$\top$

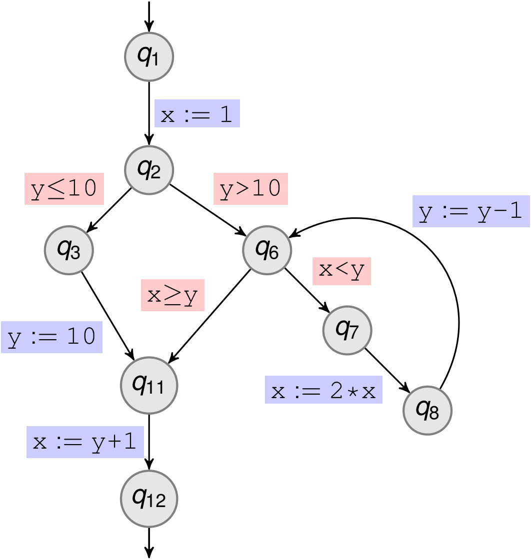
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	$\top$	$\top$
q_7	$\top$	$\top$
q_8	2	$\top$
q_{11}	$\top$	$\top$
q_{12}	$\top$	$\top$

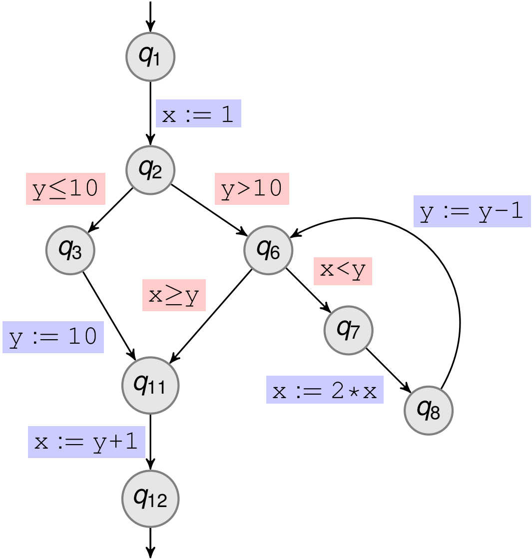
Constant Propagation Analysis: Running Example



0 : Initialization
1 : Propagation (→)

	x	y
q_1	⊤	⊤
q_2	1	⊤
q_3	1	⊤
q_6	⊤	⊤
q_7	⊤	⊤
q_8	2, ⊤	⊤
q_{11}	⊤	⊤
q_{12}	⊤	⊤

Constant Propagation Analysis: Running Example



0 : Initialization

1 : Propagation (→)

	x	y
q_1	$\top$	$\top$
q_2	1	$\top$
q_3	1	$\top$
q_6	$\top$	$\top$
q_7	$\top$	$\top$
q_8	$\top$	$\top$
q_{11}	$\top$	$\top$
q_{12}	$\top$	$\top$

Constant Propagation Analysis: Formulation

Extend $\mathbb{R}$ with a new element $\top$ to account for non-constant values

Extend $+$, $-$ and $\times$ such that $\top$ is absorbent

$$\begin{array}{rcccl} \top + r & = & r + \top & = & \top \\ \top - r & = & r - \top & = & \top \\ \top \times r & = & r \times \top & = & \top \end{array} \qquad \text{for } r \in \mathbb{R} \cup \{\top\}$$

Extend $[\![e]\!]_v$ to valuations from $\mathbb{X}$ to $\mathbb{R} \cup \{\top\}$

Domain of data flow "information"

$$\mathbb{D} \quad = \quad \mathbb{X} \ \rightarrow \ (\mathbb{R} \cup \{\top\})$$

Constant Propagation Analysis: Formulation

$$\mathbb{D} \quad = \quad \mathbb{X} \quad \to \quad (\mathbb{R} \cup \{\top\})$$

System of equations: variables C_q for $q \in Q$, with $C_q \in \mathbb{D}$

$$C_q \quad = \quad \bigotimes_{q' \xrightarrow{\mathrm{op}} q} f_{\mathrm{op}}\left(C_{q'}\right) \qquad\qquad C(q_{in}) \; = \; \lambda x \,.\, \top$$

$$v \otimes v' \quad = \quad \lambda y \,.\, \begin{cases} v(y) & \text{if } v(y) = v'(y) \\ \top & \text{otherwise} \end{cases}$$

Functions f_{op}

$$f_{x:=e}(v) \quad = \quad \lambda y \,.\, \begin{cases} v(y) & \text{if } y \neq x \\ [\![e]\!]_v & \text{if } y = x \end{cases} \qquad\qquad f_g(v) \quad = \quad v$$

Constant Propagation Analysis: Formulation

$$\mathbb{D} \quad = \quad \mathbb{X} \quad \rightarrow \quad (\mathbb{R} \;\cup\; \{\top\})$$

System of equations: variables C_q for $q \in Q$, with $C_q \in \mathbb{D}$

$$C_q \quad = \quad \bigotimes_{q' \xrightarrow{\mathrm{op}} q} \quad f_{\mathrm{op}}\left(C_{q'}\right) \qquad\qquad C(q_{in}) \;=\; \lambda\, x \,.\, \top$$

$$v \;\otimes\; v' \quad = \quad \lambda\, y \,.\, \begin{cases} v(y) & \text{if } v(y) = v'(y) \\ \top & \text{otherwise} \end{cases}$$

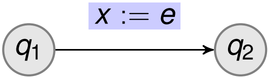
Functions f_{op}

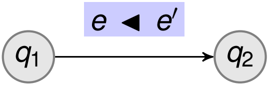
$$f_{x:=e}(v) \;=\; \lambda\, y \,.\, \begin{cases} v(y) & \text{if } y \neq x \\ [\![e]\!]_v & \text{if } y = x \end{cases} \qquad\qquad f_g(v) \;=\; v$$

Constant Propagation Analysis: Applications

Code Optimization

Constant folding





For each variable y occurring in e, if y is constant at location q_1 then we may replace y with its constant value in e.

This is sound since the analysis is conservative

Common Form of Data Flow Equations

- Domain $\mathbb{D}$ of data flow "information"
  - sets of variables, sets of expressions, valuations, ...

- Variables D_q for $q \in Q$, with value in $\mathbb{D}$
  - D_q holds data-flow information for location q

$$D_q \quad = \quad \bigwedge f\left(D_{q'}\right)$$

- "Confluence" operator $\bigwedge$ on $\mathbb{D}$ to merge data flow information
  - $\cup$, $\cap$, $\otimes$, ...

- Functions $f : \mathbb{D} \to \mathbb{D}$ to model the effect of operations

Common Form of Data Flow Equations

- Domain $\mathbb{D}$ of data flow "information"
  - sets of variables, sets of expressions, valuations, ...
- Variables D_q for $q \in Q$, with value in $\mathbb{D}$
  - D_q holds data-flow information for location q

$$D_q \quad = \quad \bigwedge f\left(D_{q'}\right)$$

- "Confluence" operator $\bigwedge$ on $\mathbb{D}$ to merge data flow information
  - $\cup$, $\cap$, $\otimes$, ...
- Functions $f : \mathbb{D} \to \mathbb{D}$ to model the effect of operations

Outline — Data Flow Analysis

Partial Order

A partial order on a set L is any binary relation $\sqsubseteq \subseteq L \times L$ satisfying for all $x, y, z \in L$:

$$x \sqsubseteq x \qquad \text{(reflexivity)}$$

$$x \sqsubseteq y \ \wedge \ y \sqsubseteq x \quad \Longrightarrow \quad x = y \qquad \text{(antisymmetry)}$$

$$x \sqsubseteq y \ \wedge \ y \sqsubseteq z \quad \Longrightarrow \quad x \sqsubseteq z \qquad \text{(transitivity)}$$

A partially ordered set is any pair $(L, \sqsubseteq)$ where L is a set and $\sqsubseteq$ is a partial order on L.

There can be x and y in L such that $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$.

Partial Order

A partial order on a set L is any binary relation $\sqsubseteq \subseteq L \times L$ satisfying for all $x, y, z \in L$:

$$x \sqsubseteq x \qquad \text{(reflexivity)}$$

$$x \sqsubseteq y \ \wedge \ y \sqsubseteq x \implies x = y \qquad \text{(antisymmetry)}$$

$$x \sqsubseteq y \ \wedge \ y \sqsubseteq z \implies x \sqsubseteq z \qquad \text{(transitivity)}$$

A partially ordered set is any pair $(L, \sqsubseteq)$ where L is a set and $\sqsubseteq$ is a partial order on L.

There can be x and y in L such that $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$.

Lower and Upper Bounds

Consider a partially ordered set $(L, \sqsubseteq)$ and a subset $X \subseteq L$.

Greatest Lower Bound

A lower bound of X is any $b \in X$ such that $b \sqsubseteq x$ for all $x \in X$.

A greatest lower bound of X is any $glb \in X$ such that:

1. glb is a lower bound of X,
2. $glb \sqsupseteq b$ for any lower bound b of X.

If X has a greatest lower bound, then it is *unique* and written $\bigsqcap X$.

Lower and Upper Bounds

Consider a partially ordered set $(L, \sqsubseteq)$ and a subset $X \subseteq L$.

Greatest Lower Bound

A lower bound of X is any $b \in X$ such that $b \sqsubseteq x$ for all $x \in X$.

A greatest lower bound of X is any $glb \in X$ such that: [. . .]

If X has a greatest lower bound, then it is *unique* and written $\bigsqcap X$.

Least Upper Bound

An upper bound of X is any $b \in X$ such that $b \sqsupseteq x$ for all $x \in X$.

A least upper bound of X is any $lub \in X$ such that:

1. lub is an upper bound of X,
2. $lub \sqsubseteq b$ for any upper bound b of X.

If X has a least upper bound, then it is *unique* and written $\bigsqcup X$.

Lower and Upper Bounds: Examples

$(\mathbb{R}, \leq)$

$$\bigsqcup \left\{0, \sqrt{2}, 4\right\} \;=\; 4 \qquad\qquad \bigsqcap \left\{ \frac{1}{2^n} \;\middle|\; n \in \mathbb{N} \right\} \;=\; 0$$

But $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ has no upper bound and no lower bound.

$(\mathcal{P}(\{-1, 0, 1\}), \subseteq)$

$$\bigsqcup \{\{0\}, \{1\}\} \;=\; \{0, 1\}$$

$$\bigsqcup \{\{-1\}, \{0, 1\}\} \;=\; \{-1, 0, 1\}$$

$$\bigsqcap \{\{-1, 0\}, \{0, 1\}\} \;=\; \{0\}$$

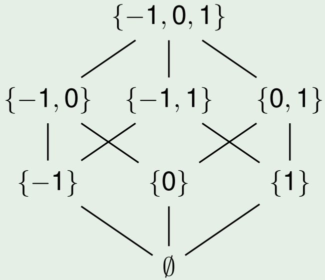
Lower and Upper Bounds: Examples

$(\mathbb{R}, \leq)$

$$\bigsqcup \left\{0, \sqrt{2}, 4\right\} \;=\; 4 \qquad\qquad \bigsqcap \left\{ \frac{1}{2^n} \;\middle|\; n \in \mathbb{N} \right\} \;=\; 0$$

But $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ has no upper bound and no lower bound.

$(\mathcal{P}(\{-1, 0, 1\}), \subseteq)$



$$\bigsqcup \{\{0\}, \{1\}\} \;=\; \{0, 1\}$$

$$\bigsqcup \{\{-1\}, \{0, 1\}\} \;=\; \{-1, 0, 1\}$$

$$\bigsqcap \{\{-1, 0\}, \{0, 1\}\} \;=\; \{0\}$$

Complete Lattice

Definition

A lattice is any partially ordered set $(L, \sqsubseteq)$ where every finite subset $X \subseteq L$ has a greatest lower bound and a least upper bound.

Definition

A complete lattice is any partially ordered set $(L, \sqsubseteq)$ where every subset $X \subseteq L$ has a greatest lower bound and a least upper bound.

The least element $\bot$ and greatest element $\top$ are defined by:

$$\bot = \bigsqcap L = \bigsqcup \emptyset \qquad \top = \bigsqcup L = \bigsqcap \emptyset$$

Example

$(\mathbb{R}, \leq)$ is a lattice, but it is not a complete lattice.

Complete Lattice

Definition

A lattice is any partially ordered set $(L, \sqsubseteq)$ where every finite subset $X \subseteq L$ has a greatest lower bound and a least upper bound.

Definition

A complete lattice is any partially ordered set $(L, \sqsubseteq)$ where every subset $X \subseteq L$ has a greatest lower bound and a least upper bound.

The least element $\bot$ and greatest element $\top$ are defined by:

$$\bot = \bigsqcap L = \bigsqcup \emptyset \qquad \top = \bigsqcup L = \bigsqcap \emptyset$$

Example

$(\mathbb{R}, \leq)$ is a lattice, but it is not a complete lattice.

Complete Lattice

Definition

A lattice is any partially ordered set $(L, \sqsubseteq)$ where every finite subset $X \subseteq L$ has a greatest lower bound and a least upper bound.

Definition

A complete lattice is any partially ordered set $(L, \sqsubseteq)$ where every subset $X \subseteq L$ has a greatest lower bound and a least upper bound.

The least element $\bot$ and greatest element $\top$ are defined by:

$$\bot = \bigsqcap L = \bigsqcup \emptyset \qquad \top = \bigsqcup L = \bigsqcap \emptyset$$

Example

$(\mathbb{R}, \leq)$ is a lattice, but it is not a complete lattice.

Fixpoints

Let $f : L \to L$ be a function on a partially ordered set $(L, \sqsubseteq)$.

Definition

A fixpoint of f is any $x \in L$ such that $f(x) = x$.

Definition

A least fixpoint of f is any $lfp \in X$ such that:

1. lfp is a fixpoint of f,
2. $lfp \sqsubseteq x$ for any fixpoint x of f.

If f has a least fixpoint, then it is *unique* and written lfp(f).

Definition

A greatest fixpoint of f is any $gfp \in X$ such that:

1. gfp is a fixpoint of f,
2. $gfp \sqsupseteq x$ for any fixpoint x of f.

Fixpoints

Let $f : L \to L$ be a function on a partially ordered set $(L, \sqsubseteq)$.

Definition

A fixpoint of f is any $x \in L$ such that $f(x) = x$.

Definition

A least fixpoint of f is any $lfp \in X$ such that:

1. lfp is a fixpoint of f,
2. $lfp \sqsubseteq x$ for any fixpoint x of f.

If f has a least fixpoint, then it is *unique* and written $\mathrm{lfp}(f)$.

Definition

A greatest fixpoint of f is any $gfp \in X$ such that:

1. gfp is a fixpoint of f,
2. $gfp \sqsupseteq x$ for any fixpoint x of f.

Fixpoints

Let $f : L \to L$ be a function on a partially ordered set $(L, \sqsubseteq)$.

Definition

A fixpoint of f is any $x \in L$ such that $f(x) = x$.

Definition

A least fixpoint of f is any $lfp \in X$ such that: [. . .]

If f has a least fixpoint, then it is *unique* and written $\text{lfp}(f)$.

Definition

A greatest fixpoint of f is any $gfp \in X$ such that:

1. gfp is a fixpoint of f,
2. $gfp \sqsupseteq x$ for any fixpoint x of f.

If f has a greatest fixpoint, then it is *unique* and written $\text{gfp}(f)$.

Knaster-Tarski Fixpoint Theorem

A function $f : L \to L$ on a partially ordered set $(L, \sqsubseteq)$ is monotonic if for all $x, y \in L$:

$$x \sqsubseteq y \quad \Longrightarrow \quad f(x) \sqsubseteq f(y)$$

Theorem

Every monotonic function f on a complete lattice $(L, \sqsubseteq)$ has a least fixpoint $\mathrm{lfp}(f)$ *and a greatest fixpoint* $\mathrm{gfp}(f)$. *Moreover:*

$$\mathrm{lfp}(f) \quad = \quad \bigsqcap \{x \in L \mid f(x) \sqsubseteq x\}$$

$$\mathrm{gfp}(f) \quad = \quad \bigsqcup \{x \in L \mid f(x) \sqsupseteq x\}$$

Order Duality

If $(L, \sqsubseteq)$ is a partially ordered set then so is $(L, \sqsupseteq)$.

If $(L, \sqsubseteq)$ is a complete lattice then so is $(L, \sqsupseteq)$.

$$\bigsqcap_{(L,\sqsupseteq)} = \bigsqcup_{(L,\sqsubseteq)} \qquad \bot_{(L,\sqsupseteq)} = \top_{(L,\sqsubseteq)}$$

$$\bigsqcup_{(L,\sqsupseteq)} = \bigsqcap_{(L,\sqsubseteq)} \qquad \top_{(L,\sqsupseteq)} = \bot_{(L,\sqsubseteq)}$$

For any monotonic function $f : L \to L$ on a complete lattice $(L, \sqsubseteq)$,

$$\text{lfp}_{(L,\sqsubseteq)}(f) = gfp_{(L,\sqsupseteq)}(f)$$

$$\text{gfp}_{(L,\sqsubseteq)}(f) = lfp_{(L,\sqsupseteq)}(f)$$

We shall focus on least fixpoints.

Order Duality

If $(L, \sqsubseteq)$ is a partially ordered set then so is $(L, \sqsupseteq)$.

If $(L, \sqsubseteq)$ is a complete lattice then so is $(L, \sqsupseteq)$.

$$\bigsqcap_{(L,\sqsupseteq)} = \bigsqcup_{(L,\sqsubseteq)} \qquad \bot_{(L,\sqsupseteq)} = \top_{(L,\sqsubseteq)}$$

$$\bigsqcup_{(L,\sqsupseteq)} = \bigsqcap_{(L,\sqsubseteq)} \qquad \top_{(L,\sqsupseteq)} = \bot_{(L,\sqsubseteq)}$$

For any monotonic function $f : L \to L$ on a complete lattice $(L, \sqsubseteq)$,

$$\mathrm{lfp}_{(L,\sqsubseteq)}(f) = gfp_{(L,\sqsupseteq)}(f)$$

$$\mathrm{gfp}_{(L,\sqsubseteq)}(f) = lfp_{(L,\sqsupseteq)}(f)$$

We shall focus on least fixpoints.

Ascending Chain Condition

An ascending chain in a partially ordered set $(L, \sqsubseteq)$ is any infinite sequence $x_0, x_1, \ldots$ of elements of L satisfying $x_i \sqsubseteq x_{i+1}$ for all $i \in \mathbb{N}$.

A partially ordered set $(L, \sqsubseteq)$ satisfies the ascending chain condition if every ascending chain $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ of elements of L is eventually stationary.

Examples

$(\mathbb{R}, \leq)$ does not satisfy the ascending chain condition.

$(\mathbb{N}, \geq)$ satisfies the ascending chain condition.

Ascending Chain Condition

An ascending chain in a partially ordered set $(L, \sqsubseteq)$ is any infinite sequence $x_0, x_1, \ldots$ of elements of L satisfying $x_i \sqsubseteq x_{i+1}$ for all $i \in \mathbb{N}$.

A partially ordered set $(L, \sqsubseteq)$ satisfies the ascending chain condition if every ascending chain $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ of elements of L is eventually stationary.

Examples

$(\mathbb{R}, \leq)$ does not satisfy the ascending chain condition.

$(\mathbb{N}, \geq)$ satisfies the ascending chain condition.

Kleene Iteration

Consider a partially ordered set $(L, \sqsubseteq)$ and $f : L \to L$ monotonic.

The Kleene iteration $(f^i(\bot))_{i \in \mathbb{N}}$ is an ascending chain:

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

For every $k \in \mathbb{N}$, if $f^k(\bot) = f^{k+1}(\bot)$ then $f^k(\bot)$ is the least fixpoint of f.

```
LFP(f : L → L)
x ← ⊥
repeat
    t ← x
    x ← f(x)
until t = x
return x
```

Correction and termination

1. For every monotonic f, if LFP(f) terminates then it returns lfp(f).
2. If L satisfies the ascending chain condition then LFP(f) always terminates (on monotonic f).

Kleene Iteration

Consider a partially ordered set $(L, \sqsubseteq)$ and $f : L \rightarrow L$ monotonic.

The Kleene iteration $\left(f^i(\bot)\right)_{i \in \mathbb{N}}$ is an ascending chain:

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

For every $k \in \mathbb{N}$, if $f^k(\bot) = f^{k+1}(\bot)$ then $f^k(\bot)$ is the least fixpoint of f.

```
LFP(f : L → L)
x ← ⊥
repeat
    t ← x
    x ← f(x)
until t = x
return x
```

Correction and termination

1. For every monotonic f, if LFP(f) terminates then it returns lfp(f).
2. If L satisfies the ascending chain condition then LFP(f) always terminates (on monotonic f).

Constructing Complete Lattices: Power Set

For any set S, the pair $(\mathcal{P}(S), \sqsubseteq)$ is a complete lattice, where $\sqsubseteq = \subseteq$.

$\sqcap$, $\bigsqcup$, $\bot$ and $\top$ satisfy:

$$\bigsqcap = \bigcap \qquad \bot = \emptyset$$
$$\bigsqcup = \bigcup \qquad \top = S$$

If S is finite then $(\mathcal{P}(S), \sqsubseteq)$ satisfies the ascending chain condition.

Constructing Complete Lattices: Functions

For any set S and complete lattice $(L, \sqsubseteq)$, the pair $(S \to L, \sqsubseteq)$ is a complete lattice, where $\sqsubseteq$ is defined by:

$$f \sqsubseteq g \quad \text{if} \quad f(x) \sqsubseteq g(x) \quad \text{for all } x \in S$$

$\bigsqcap$, $\bigsqcup$, $\bot$ and $\top$ satisfy:

$$\bigsqcap X = \lambda x . \bigsqcap \{f(x) \mid f \in X\} \qquad \bot = \lambda x . \bot$$

$$\bigsqcup X = \lambda x . \bigsqcup \{f(x) \mid f \in X\} \qquad \top = \lambda x . \top$$

If S is finite and $(L, \sqsubseteq)$ satisfies the ascending chain condition then $(S \to L, \sqsubseteq)$ satisfies the ascending chain condition.

Outline — Data Flow Analysis

Common Form of Data Flow Equations (Recall)

- Domain $\mathbb{D}$ of data flow "information"
  - sets of variables, sets of expressions, valuations, ...
- Variables D_q for $q \in Q$, with value in $\mathbb{D}$
  - D_q holds data-flow information for location q

$$D_q = \bigwedge f(D_{q'})$$

- "Confluence" operator $\bigwedge$ on $\mathbb{D}$ to merge data flow information
  - $\cup$, $\cap$, $\otimes$, ...
- Functions $f : \mathbb{D} \to \mathbb{D}$ to model the effect of operations

Monotone Frameworks

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts
- Set $\mathcal{F}$ of monotonic transfer functions $f : L \to L$

Partial order $\sqsubseteq$ compares the precision of data flow facts:

- $\phi \sqsubseteq \psi$ means that ϕ is more precise than ψ.
- $\bigsqcup X$ is the most precise fact consistent with all facts $\phi \in X$.

Conservative Approximation

$\phi \sqsubseteq \psi$ means that ψ soundly approximates ϕ.

If $\phi \sqsubseteq \psi$ then it is sound, but less precise, to replace ϕ by ψ.

Monotone Frameworks

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts
- Set $\mathcal{F}$ of monotonic transfer functions $f : L \to L$

Partial order $\sqsubseteq$ compares the precision of data flow facts:

- $\phi \sqsubseteq \psi$ means that ϕ is more precise than ψ.
- $\bigsqcup X$ is the most precise fact consistent with all facts $\phi \in X$.

Conservative Approximation

$\phi \sqsubseteq \psi$ means that ψ soundly approximates ϕ.

If $\phi \sqsubseteq \psi$ then it is sound, but less precise, to replace ϕ by ψ.

Monotone Frameworks

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts
- Set $\mathcal{F}$ of monotonic transfer functions $f : L \to L$

Partial order $\sqsubseteq$ compares the precision of data flow facts:

- $\phi \sqsubseteq \psi$ means that ϕ is more precise than ψ.
- $\bigsqcup X$ is the most precise fact consistent with all facts $\phi \in X$.

Conservative Approximation

$\phi \sqsubseteq \psi$ means that ψ soundly approximates ϕ.

If $\phi \sqsubseteq \psi$ then it is sound, but less precise, to replace ϕ by ψ.

Data Flow Facts: Example for Live Variables Analysis

Semantic Definition of Liveness

A variable x is live at location q if there exists an execution path starting from q where x is used before it is modified.

Consider a control flow automaton with variables $\mathrm{X} = \{\mathrm{x}, \mathrm{y}, \mathrm{z}\}$.

Complete lattice $(L, \sqsubseteq)$ of data flow facts: $(\mathcal{P}(\mathrm{X}), \subseteq)$

The fact $\{\mathrm{x}, \mathrm{z}\}$ means: *the variables that are live are among* $\{x, z\}$*.*

i.e. *the variable* y *is not live.*

The fact $\{\mathrm{x}\}$ is more precise than $\{\mathrm{x}, \mathrm{z}\}$, but incomparable with $\{\mathrm{y}\}$.

The fact $\{\mathrm{x}, \mathrm{z}\}$ soundly approximates the fact $\{\mathrm{x}\}$.

Data Flow Instances

Data Flow Instance

- Monotone framework $\langle (L, \sqsubseteq), \mathcal{F} \rangle$
- Control flow automaton $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$
- Transfer mapping $f : \mathrm{Op} \rightarrow \mathcal{F}$
- Initial data flow value $\imath \in L$

Notation for transfer mapping: f_{op} instead of $f(\mathrm{op})$

Two possible directions for data flow analysis: forward and backward

Transfer functions f_{op} must be defined in accordance with the direction of the analysis.

Data Flow Instances

Data Flow Instance

- Monotone framework $\langle (L, \sqsubseteq), \mathcal{F} \rangle$
- Control flow automaton $\langle Q, q_{in}, q_{out}, \mathtt{X}, \rightarrow \rangle$
- Transfer mapping $f : \mathtt{Op} \rightarrow \mathcal{F}$
- Initial data flow value $\imath \in L$

Notation for transfer mapping: $f_{\mathtt{op}}$ instead of $f(\mathtt{op})$

Two possible directions for data flow analysis: forward and backward

Transfer functions $f_{\mathtt{op}}$ must be defined in accordance with the direction of the analysis.

Data Flow Equations

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

System of equations: variables A_q for $q \in Q$, with $A_q \in L$

Forward Analysis

$$A_q = I_q \sqcup \bigsqcup_{q' \xrightarrow{\mathrm{op}} q} f_{\mathrm{op}}(A_{q'}) \qquad I_q = \begin{cases} \imath & \text{if } q = q_{in} \\ \bot & \text{otherwise} \end{cases}$$

Backward Analysis

$$A_q = I_q \sqcup \bigsqcup_{q \xrightarrow{\mathrm{op}} q'} f_{\mathrm{op}}(A_{q'}) \qquad I_q = \begin{cases} \imath & \text{if } q = q_{out} \\ \bot & \text{otherwise} \end{cases}$$

Data Flow Equations

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

System of equations: variables A_q for $q \in Q$, with $A_q \in L$

Forward Analysis

$$A_q = I_q \sqcup \bigsqcup_{q' \xrightarrow{\text{op}} q} f_{\text{op}}(A_{q'}) \qquad I_q = \begin{cases} \imath & \text{if } q = q_{in} \\ \bot & \text{otherwise} \end{cases}$$

Backward Analysis

$$A_q = I_q \sqcup \bigsqcup_{q \xrightarrow{\text{op}} q'} f_{\text{op}}(A_{q'}) \qquad I_q = \begin{cases} \imath & \text{if } q = q_{out} \\ \bot & \text{otherwise} \end{cases}$$

Minimal Fixpoint (MFP) Solution

The system of data flow equations may have several solutions...

We are interested in the "least solution" to the data flow equations.

Complete lattice $(L, \sqsubseteq)$ extended to $(Q \to L, \sqsubseteq)$

The forward minimal fixpoint solution $\overrightarrow{\text{MFP}}$ of the data flow instance is the least fixpoint of the monotonic function $\overrightarrow{\Delta}$ on $(Q \to L)$:

$$\overrightarrow{\Delta}(a) \;=\; \lambda q \,.\, \begin{cases} \imath \;\sqcup\; \bigsqcup_{q' \xrightarrow{\text{op}} q} f_{\text{op}}(a(q')) & \text{if } q = q_{in} \\ \bigsqcup_{q' \xrightarrow{\text{op}} q} f_{\text{op}}(a(q')) & \text{otherwise} \end{cases}$$

Minimal Fixpoint (MFP) Solution

The system of data flow equations may have several solutions…

We are interested in the "least solution" to the data flow equations.

Complete lattice $(L, \sqsubseteq)$ extended to $(Q \rightarrow L, \sqsubseteq)$

The forward minimal fixpoint solution $\overrightarrow{\mathrm{MFP}}$ of the data flow instance is the least fixpoint of the monotonic function $\overrightarrow{\Delta}$ on $(Q \rightarrow L)$:

$$\overrightarrow{\Delta}(a) \quad = \quad \lambda q \, . \begin{cases} \imath \ \sqcup \displaystyle\bigsqcup_{q' \xrightarrow{\mathrm{op}} q} f_{\mathrm{op}}(a(q')) & \text{if } q = q_{in} \\ \displaystyle\bigsqcup_{q' \xrightarrow{\mathrm{op}} q} f_{\mathrm{op}}(a(q')) & \text{otherwise} \end{cases}$$

Minimal Fixpoint (MFP) Solution

The system of data flow equations may have several solutions...

We are interested in the "least solution" to the data flow equations.

Complete lattice $(L, \sqsubseteq)$ extended to $(Q \to L, \sqsubseteq)$

The backward minimal fixpoint solution $\overleftarrow{\mathrm{MFP}}$ of the data flow instance is the least fixpoint of the monotonic function $\overleftarrow{\Delta}$ on $(Q \to L)$:

$$\overleftarrow{\Delta}(a) \;=\; \lambda q \,.\, \begin{cases} \imath \;\sqcup\; \bigsqcup_{q \xrightarrow{\mathrm{op}} q'} f_{\mathrm{op}}(a(q')) & \text{if } q = q_{out} \\ \bigsqcup_{q \xrightarrow{\mathrm{op}} q'} f_{\mathrm{op}}(a(q')) & \text{otherwise} \end{cases}$$

Constraint-Based Formulation

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

Constraint system: variables A_q for $q \in Q$, with $A_q \in L$

Forward Analysis

$$\overrightarrow{(CS)} \quad \begin{cases} A_{q_{in}} & \sqsupseteq \quad \imath \\ A_{q'} & \sqsupseteq \quad f_{\mathrm{op}}(A_q) \quad \text{for each } q \xrightarrow{\mathrm{op}} q' \end{cases}$$

By Knaster-Tarski Fixpoint Theorem,

$$\overrightarrow{\mathrm{MFP}} \quad = \quad \bigsqcap \left\{ a \in Q \rightarrow L \;\middle|\; a \models \overrightarrow{(CS)} \right\}$$

Any solution to $\overrightarrow{(CS)}$ is a sound approximation of $\overrightarrow{\mathrm{MFP}}$.

Constraint-Based Formulation

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

Constraint system: variables A_q for $q \in Q$, with $A_q \in L$

Backward Analysis

$$\overleftarrow{(CS)} \quad \begin{cases} A_{q_{out}} & \sqsupseteq \quad \imath \\ A_{q'} & \sqsupseteq \quad f_{\mathrm{op}}(A_q) \quad \text{for each } q' \xrightarrow{\mathrm{op}} q \end{cases}$$

By Knaster-Tarski Fixpoint Theorem,

$$\overleftarrow{\mathrm{MFP}} \quad = \quad \bigsqcap \left\{ a \in Q \rightarrow L \;\middle|\; a \models \overleftarrow{(CS)} \right\}$$

Any solution to $\overleftarrow{(CS)}$ is a sound approximation of $\overleftarrow{\mathrm{MFP}}$.

Live Variables Analysis (Revisited)

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow\rangle$

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts: $(\mathcal{P}(\mathrm{X}), \subseteq)$
- Set $\mathcal{F}$ of monotonic transfer functions:

$$\mathcal{F} \quad = \quad \{\lambda\,\phi\,.\,gen \cup (\phi \setminus kill) \mid gen, kill \in L\}$$

Data Flow Instance

- Initial data flow value: $\emptyset$
- Transfer mapping: $f_{\mathrm{op}}(\phi) \; = \; Gen_{\mathrm{op}} \; \cup \; (\phi \setminus Kill_{\mathrm{op}})$

Backward analysis

Available Expressions Analysis (Revisited)

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts: $(\mathcal{P}(\mathit{SubExp}(\rightarrow)), \supseteq)$
- Set $\mathcal{F}$ of monotonic transfer functions:

$$\mathcal{F} = \{\lambda\,\phi\,.\,\mathit{gen} \cup (\phi \setminus \mathit{kill}) \mid \mathit{gen}, \mathit{kill} \in L\}$$

Data Flow Instance

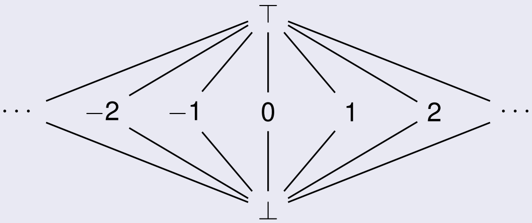
- Initial data flow value: $\emptyset$
- Transfer mapping: $f_{\mathrm{op}}(\phi) = \mathit{Gen}_{\mathrm{op}} \cup (\phi \setminus \mathit{Kill}_{\mathrm{op}})$

Forward analysis

Constant Propagation Analysis (Revisited)

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

Constant Propagation Lattice for a Single Variable



$(\mathbb{R} \cup \{\bot, \top\}, \sqsubseteq)$

ϕ	*Meaning*
$\top$	$\mathbb{R}$
$r \in \mathbb{R}$	$\{r\}$
$\bot$	$\emptyset$

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts: $(\mathrm{X} \rightarrow (\mathbb{R} \cup \{\bot, \top\}), \sqsubseteq)$
- Set $\mathcal{F}$ defined as the set of all monotonic transfer functions on L.

Constant Propagation Analysis (Revisited)

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathbb{X}, \rightarrow \rangle$

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts: $(\mathbb{X} \rightarrow (\mathbb{R} \cup \{\bot, \top\}), \sqsubseteq)$
- Set $\mathcal{F}$ defined as the set of all monotonic transfer functions on L.

Data Flow Instance

- Initial data flow value: $\top$
- Transfer mapping:

$$f_{x:=e}(\phi) \quad = \quad \lambda\, y\,.\begin{cases} \phi(y) & \text{if } y \neq x \\ [\![e]\!]_\phi & \text{if } y = x \end{cases} \qquad\qquad f_g(\phi) \quad = \quad \phi$$

Forward analysis

Extension of $[\![e]\!]$ to valuations in $\mathbb{X} \rightarrow (\mathbb{R} \cup \{\bot, \top\})$

For $r \in \mathbb{R} \cup \{\top\}$

$$\begin{aligned} \top + r &= r + \top = \top \\ \top - r &= r - \top = \top \\ \top \times r &= r \times \top = \top \end{aligned}$$

For $r \in \mathbb{R} \cup \{\bot, \top\}$

$$\begin{aligned} \bot + r &= r + \bot = \bot \\ \bot - r &= r - \bot = \bot \\ \bot \times r &= r \times \bot = \bot \end{aligned}$$

Expressions: $[\![e]\!]_v$

$$\begin{aligned} [\![c]\!]_v &= c & [c \in \mathbb{Q}] \\ [\![x]\!]_v &= v(x) & [x \in \mathbb{X}] \\ [\![e_1 + e_2]\!]_v &= [\![e_1]\!]_v + [\![e_2]\!]_v \\ [\![e_1 - e_2]\!]_v &= [\![e_1]\!]_v - [\![e_2]\!]_v \\ [\![e_1 * e_2]\!]_v &= [\![e_1]\!]_v \times [\![e_2]\!]_v \end{aligned}$$

Constant Propagation Analysis (Revisited)

Extension of $[\![e]\!]$ to valuations in $\mathbb{X} \rightarrow (\mathbb{R} \cup \{\bot, \top\})$

For $r \in \mathbb{R} \cup \{\top\}$

$$\begin{aligned} \top + r &= r + \top = \top \\ \top - r &= r - \top = \top \\ \top \times r &= r \times \top = \top \end{aligned}$$

For $r \in \mathbb{R} \cup \{\bot, \top\}$

$$\begin{aligned} \bot + r &= r + \bot = \bot \\ \bot - r &= r - \bot = \bot \\ \bot \times r &= r \times \bot = \bot \end{aligned}$$

Expressions: $[\![e]\!]_v$

$$\begin{aligned} [\![c]\!]_v &= c \quad [c \in \mathbb{Q}] \\ [\![x]\!]_v &= v(x) \quad [x \in \mathbb{X}] \\ [\![e_1 + e_2]\!]_v &= [\![e_1]\!]_v + [\![e_2]\!]_v \\ [\![e_1 - e_2]\!]_v &= [\![e_1]\!]_v - [\![e_2]\!]_v \\ [\![e_1 * e_2]\!]_v &= [\![e_1]\!]_v \times [\![e_2]\!]_v \end{aligned}$$

(Forward) MFP Computation by Kleene Iteration

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

```
a ← λq.⊥
repeat
    b ← a
    a ← Δ⃗(a)
until b = a
return a
```

```
foreach q ∈ Q
    a[q] ← ⊥
a[q_in] ← ı
repeat
    foreach q ∈ Q
        b[q] ← a[q]
    foreach q ∈ Q
        a[q] ← ⊔_{q' -op-> q} f_op(b[q'])
until (∀ q ∈ Q · b[q] = a[q])
return a
```

Correction and termination

1. Returns $\overrightarrow{\mathrm{MFP}}$ when it terminates
2. Always terminates when $(L, \sqsubseteq)$ satisfies the ascending chain condition

(Forward) MFP Computation by Kleene Iteration

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

```
a ← λq.⊥
repeat
    b ← a
    a ← Δ→(a)
until b = a
return a
```

```
foreach q ∈ Q
    a[q] ← ⊥
a[q_in] ← ı
repeat
    foreach q ∈ Q
        b[q] ← a[q]
    foreach q ∈ Q
        a[q] ← ⊔_{q' --op--> q} f_op(b[q'])
until (∀ q ∈ Q · b[q] = a[q])
return a
```

Correction and termination

1. Returns $\overrightarrow{\mathrm{MFP}}$ when it terminates
2. Always terminates when $(L, \sqsubseteq)$ satisfies the ascending chain condition

(Forward) MFP Computation by Kleene Iteration

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

```
a ← λq.⊥
repeat
    b ← a
    a ← Δ⃗(a)
until b = a
return a
```

```
foreach q ∈ Q
    a[q] ← ⊥
a[q_in] ← ı
repeat
    foreach q ∈ Q
        b[q] ← a[q]
    foreach q ∈ Q
        a[q] ← ⊔_{q' -op-> q} f_op(b[q'])
until (∀ q ∈ Q · b[q] = a[q])
return a
```

Correction and termination

1. Returns $\overrightarrow{\mathrm{MFP}}$ when it terminates
2. Always terminates when $(L, \sqsubseteq)$ satisfies the ascending chain condition

We can improve!

(Forward) MFP Computation by Round-Robin Iteration

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

$$
\begin{array}{l}
\textbf{foreach } q \in Q \\
\quad \mathtt{a}[q] \leftarrow \bot \\
\mathtt{a}[q_{in}] \leftarrow \imath \\
\textbf{do} \\
\quad \mathtt{change} \leftarrow \mathtt{false} \\
\quad \textbf{foreach } q \xrightarrow{\mathrm{op}} q' \\
\quad\quad \mathtt{new} \leftarrow f_{\mathrm{op}}(\mathtt{a}[q]) \\
\quad\quad \textbf{if } \mathtt{new} \not\sqsubseteq \mathtt{a}[q'] \\
\quad\quad\quad \mathtt{a}[q'] \leftarrow \mathtt{a}[q'] \sqcup \mathtt{new} \\
\quad\quad\quad \mathtt{change} \leftarrow \mathtt{true} \\
\textbf{while } \mathtt{change} \\
\textbf{return } \mathtt{a}
\end{array}
$$

The foreach loop iterates over transitions in $\rightarrow$.

Propagation of facts

- benefits from previous propagations
- records whether there was a change

Correct and always faster than Kleene iteration

(Forward) MFP Computation by Round-Robin Iteration

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

```
foreach q ∈ Q
    a[q] ← ⊥
a[q_in] ← ı
do
    change ← false
    foreach q --op--> q'
        new ← f_op(a[q])
        if new ⋢ a[q']
            a[q'] ← a[q'] ⊔ new
            change ← true
while change
return a
```

The foreach loop iterates over transitions in $\rightarrow$.

Propagation of facts

- benefits from previous propagations
- records whether there was a change

Correct and always faster than Kleene iteration

(Forward) MFP Computation by Worklist Iteration

```
wl ← nil
foreach q' --op--> q
   wl ← cons((q, op, q'), wl)
foreach q ∈ Q
    a[q] ← ⊥
a[q_in] ← ı
while wl ≠ nil
    (q, op, q') ← head(wl)
    wl ← tail(wl)
    new ← f_op(a[q])
        if new ⋢ a[q']
            a[q'] ← a[q] ⊔ new
            foreach q' --op'--> q''
                wl ← cons((q', op', q''), wl)
return a
```

Vs Round-Robin

- ☺ Less computations
- ☹ Overhead

Worklist structures

- LIFO
- FIFO
- Set
- ...

(Forward) MFP Computation by Worklist Iteration

```
wl ← nil
foreach q' --op--> q
   wl ← cons((q, op, q'), wl)
foreach q ∈ Q
    a[q] ← ⊥
a[q_in] ← ı
while wl ≠ nil
    (q, op, q') ← head(wl)
    wl ← tail(wl)
    new ← f_op(a[q])
        if new ⋢ a[q']
            a[q'] ← a[q] ⊔ new
            foreach q' --op'--> q''
                wl ← cons((q', op', q''), wl)
return a
```

Vs Round-Robin

☺ Less computations

☹ Overhead

Worklist structures

- LIFO
- FIFO
- Set
- ...

Optimization of MFP Computation with SCCs

1. Decompose control flow automaton into strongly connected components
2. Transitions between SCCs induce a partial order between SCCs
3. Compute the MFP solution component after component, following the partial order between SCCs

This optimization often pays off in practice

Further optimizations are possible. . .

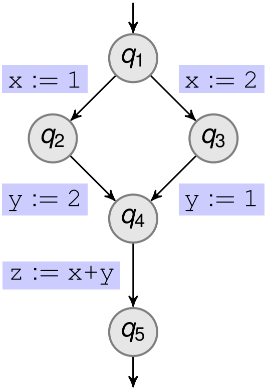
Optimization of MFP Computation with SCCs

1. Decompose control flow automaton into strongly connected components
2. Transitions between SCCs induce a partial order between SCCs
3. Compute the MFP solution component after component, following the partial order between SCCs

This optimization often pays off in practice

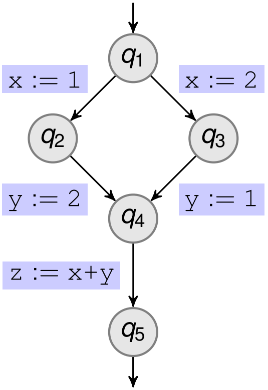
Further optimizations are possible. . .

Loss of Precision with the MFP Solution



At q_5, we have $z = 3$

Loss of Precision with the MFP Solution



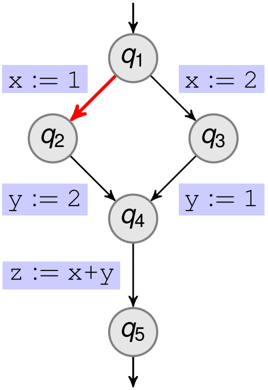
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	$\bot$	$\bot$	$\bot$
q_3	$\bot$	$\bot$	$\bot$
q_4	$\bot$	$\bot$	$\bot$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution

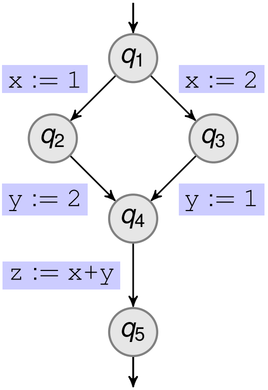


	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	$\bot$	$\bot$	$\bot$
q_4	$\bot$	$\bot$	$\bot$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information



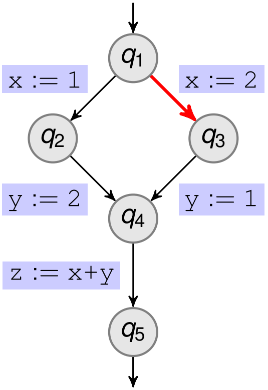
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	$\bot$	$\bot$	$\bot$
q_4	$\bot$	$\bot$	$\bot$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



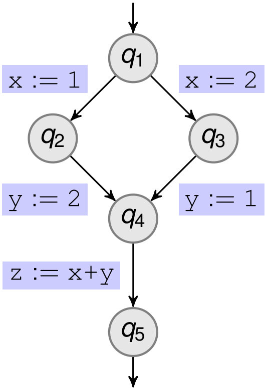
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	$\bot$	$\bot$	$\bot$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



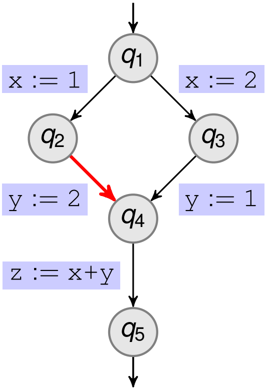
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	$\bot$	$\bot$	$\bot$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $\texttt{z} = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



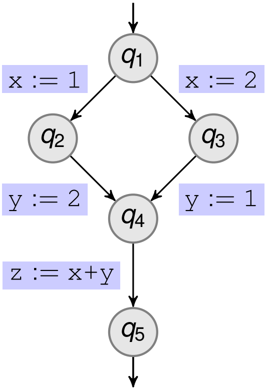
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	1	2	$\top$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



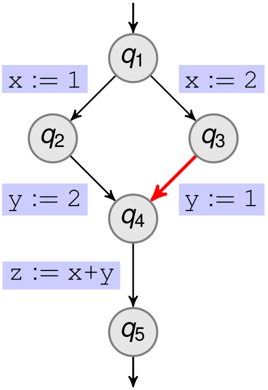
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	1	2	$\top$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $\mathtt{z} = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



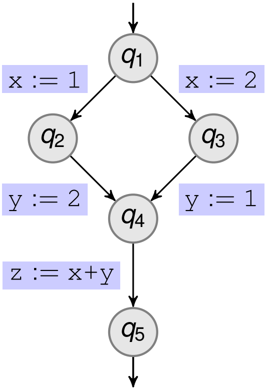
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	$1 \sqcup 2$	$2 \sqcup 1$	$\top$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $\texttt{z} = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



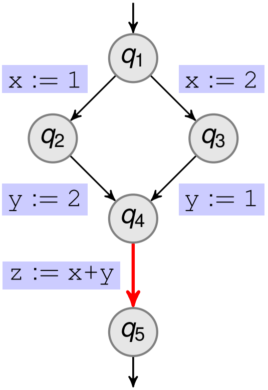
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	$\top$	$\top$	$\top$
q_5	$\bot$	$\bot$	$\bot$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



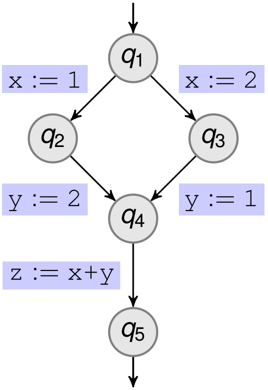
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	$\top$	$\top$	$\top$
q_5	$\top$	$\top$	$\top$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



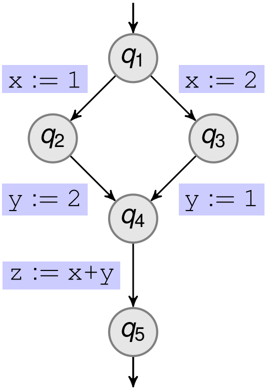
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	$\top$	$\top$	$\top$
q_5	$\top$	$\top$	$\top$

At q_5, we have $z = 3$

Loss of Precision

Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Loss of Precision with the MFP Solution



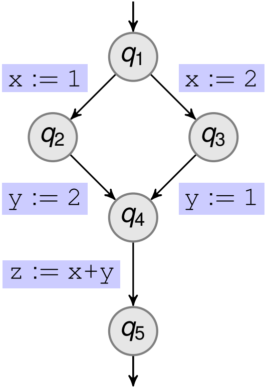
	x	y	z
q_1	$\top$	$\top$	$\top$
q_2	1	$\top$	$\top$
q_3	2	$\top$	$\top$
q_4	$\top$	$\top$	$\top$
q_5	$\top$	$\top$	$\top$

At q_5, we have $\mathtt{z} = 3$

Loss of Precision

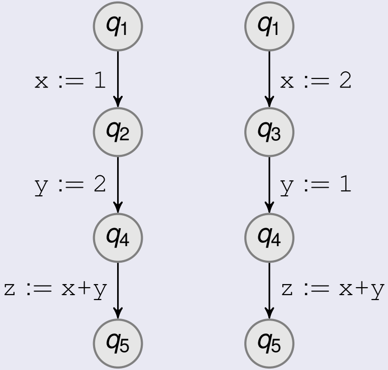
Cause: application of $\bigsqcup$ at q_4 to merge data flow information

Alternative Approach for Better Precision

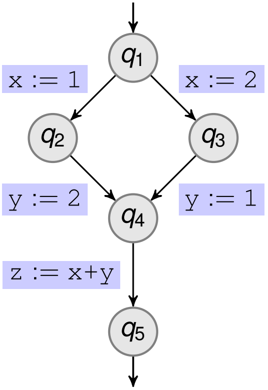


At q_5, we have $z = 3$

Control Paths from q_1 to q_5

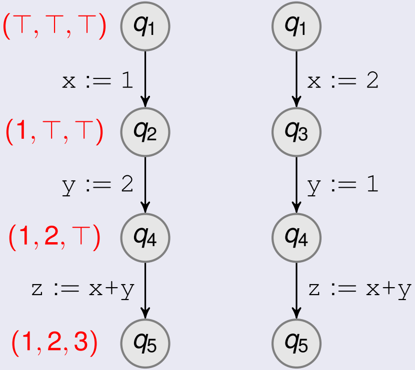


Alternative Approach for Better Precision

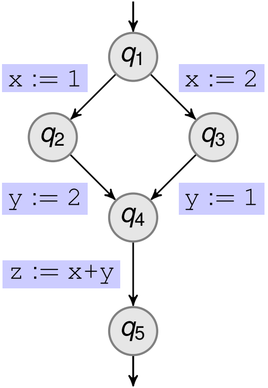


At q_5, we have $z = 3$

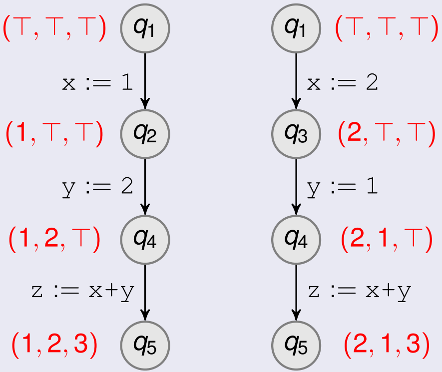
Control Paths from q_1 to q_5



Alternative Approach for Better Precision

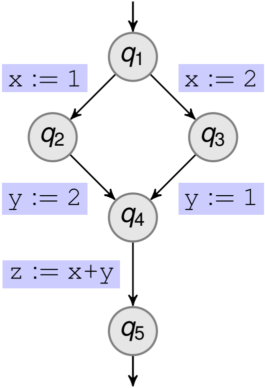


Control Paths from q_1 to q_5



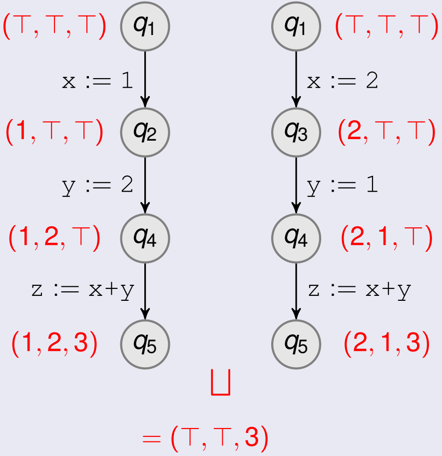
At q_5, we have $z = 3$

Alternative Approach for Better Precision



At q_5, we have $z = 3$

Control Paths from q_1 to q_5



Meet Over All Paths (MOP) Solution

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

Forward Meet Over All Paths Solution

$$\overrightarrow{\mathrm{MOP}} = \lambda q \,.\, \bigsqcup \left\{ f_{\mathrm{op}_k} \circ \cdots \circ f_{\mathrm{op}_0}(\imath) \;\middle|\; q_{in} \xrightarrow{\mathrm{op}_0} q_1 \cdots q_k \xrightarrow{\mathrm{op}_k} q \right\}$$

Backward Meet Over All Paths Solution

$$\overleftarrow{\mathrm{MOP}} = \lambda q \,.\, \bigsqcup \left\{ f_{\mathrm{op}_0} \circ \cdots \circ f_{\mathrm{op}_k}(\imath) \;\middle|\; q \xrightarrow{\mathrm{op}_0} q_1 \cdots q_k \xrightarrow{\mathrm{op}_k} q_{out} \right\}$$

More precise than MFP

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

Not Computable in General

$\overrightarrow{\mathrm{MOP}}(q) \stackrel{?}{=} 1$ is undecidable for constant propagation

Meet Over All Paths (MOP) Solution

Consider a data flow instance $\langle (L, \sqsubseteq), \mathcal{F}, Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow, f, \imath \rangle$.

Forward Meet Over All Paths Solution

$$\overrightarrow{\mathrm{MOP}} = \lambda q \,.\, \bigsqcup \left\{ f_{\mathrm{op}_k} \circ \cdots \circ f_{\mathrm{op}_0}(\imath) \;\middle|\; q_{in} \xrightarrow{\mathrm{op}_0} q_1 \cdots q_k \xrightarrow{\mathrm{op}_k} q \right\}$$

Backward Meet Over All Paths Solution

$$\overleftarrow{\mathrm{MOP}} = \lambda q \,.\, \bigsqcup \left\{ f_{\mathrm{op}_0} \circ \cdots \circ f_{\mathrm{op}_k}(\imath) \;\middle|\; q \xrightarrow{\mathrm{op}_0} q_1 \cdots q_k \xrightarrow{\mathrm{op}_k} q_{out} \right\}$$

More precise than MFP

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

Not Computable in General

$\overrightarrow{\mathrm{MOP}}(q) \stackrel{?}{=} 1$ is undecidable for constant propagation

MOP = MFP in Distributive Frameworks

A monotone framework $\langle (L, \sqsubseteq), \mathcal{F} \rangle$ is distributive if every $f \in \mathcal{F}$ is completely additive:

$$f\left(\bigsqcup X\right) = \bigsqcup \{f(\phi) \mid \phi \in X\} \qquad \text{(for all } X \subseteq L\text{)}$$

Theorem

For any data flow instance over a distributive monotone framework,

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

Intuition

In a distributive framework, applying $\bigsqcup$ "early" does not lose precision:

$$f_{\mathrm{op}_5}\left(f_{\mathrm{op}_2}(\phi) \sqcup f_{\mathrm{op}_3}(\psi)\right) = f_{\mathrm{op}_5} \circ f_{\mathrm{op}_2}(\phi) \sqcup f_{\mathrm{op}_5} \circ f_{\mathrm{op}_3}(\psi)$$

MOP = MFP in Distributive Frameworks

A monotone framework $\langle (L, \sqsubseteq), \mathcal{F} \rangle$ is distributive if every $f \in \mathcal{F}$ is completely additive:

$$f\left(\bigsqcup X\right) = \bigsqcup \{f(\phi) \mid \phi \in X\} \qquad \text{(for all } X \subseteq L\text{)}$$

Theorem

For any data flow instance over a distributive monotone framework,

$$\begin{aligned} \overrightarrow{\mathrm{MOP}} &= \overrightarrow{\mathrm{MFP}} \\ \overleftarrow{\mathrm{MOP}} &= \overleftarrow{\mathrm{MFP}} \end{aligned}$$

Intuition

In a distributive framework, applying $\bigsqcup$ "early" does not lose precision:

$$f_{\mathrm{op}_5}\left(f_{\mathrm{op}_2}(\phi) \sqcup f_{\mathrm{op}_3}(\psi)\right) = f_{\mathrm{op}_5} \circ f_{\mathrm{op}_2}(\phi) \sqcup f_{\mathrm{op}_5} \circ f_{\mathrm{op}_3}(\psi)$$

Examples of Distributive Monotone Frameworks

Gen / Kill Monotone Frameworks

- Complete lattice $(L, \sqsubseteq)$ of data flow facts:

$$L \; = \; \mathcal{P}(S) \text{ for some set } S \qquad\qquad \sqsubseteq \text{ is } \subseteq \text{ or } \supseteq$$

- Set $\mathcal{F}$ of monotonic transfer functions:

$$\mathcal{F} \;\; = \;\; \{\lambda\,\phi\,.\,\mathit{gen} \cup (\phi \setminus \mathit{kill}) \mid \mathit{gen}, \mathit{kill} \in L\}$$

All gen / kill monotone frameworks are distributive

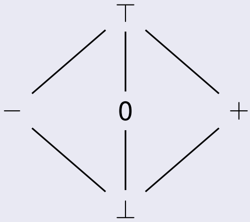
Examples

- Live Variables
- Available Expressions
- Uninitialized Variables
- ...

Sign Analysis: Monotone Framework

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

(Simplified) Sign Lattice for a Single Variable: $(\mathit{Sign}, \sqsubseteq)$



ϕ	*Meaning*
$\top$	$\mathbb{R}$
$-$	$\{r \in \mathbb{R} \mid r < 0\}$
$+$	$\{r \in \mathbb{R} \mid r > 0\}$
0	$\{0\}$
$\bot$	$\emptyset$

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts: $(\mathrm{X} \rightarrow \mathit{Sign}, \sqsubseteq)$
- Set $\mathcal{F}$ defined as the set of all monotonic transfer functions on L.

Sign Analysis: Data Flow Instance

Control Flow Automaton: $\langle Q, q_{in}, q_{out}, \mathrm{X}, \rightarrow \rangle$

Monotone Framework

- Complete lattice $(L, \sqsubseteq)$ of data flow facts: $(\mathrm{X} \rightarrow \mathit{Sign}, \sqsubseteq)$
- Set $\mathcal{F}$ defined as the set of all monotonic transfer functions on L.

Data Flow Instance

- Initial data flow value: $\top$
- Transfer mapping:

$$f_{x:=e}(\phi) \quad = \quad \lambda y \,.\, \begin{cases} \phi(y) & \text{if } y \neq x \\ [\![e]\!]_\phi & \text{if } y = x \end{cases} \qquad\qquad f_g(\phi) \quad = \quad \phi$$

Forward analysis

Sign Analysis: Transfer Mapping

Need to define $[\![e]\!]$ for valuations v in $\mathbb{X} \to \{-, 0, +, \bot, \top\}$

Expressions: $[\![e]\!]_v$

$$
\begin{aligned}
[\![c]\!]_v &= sign(c) \quad [c \in \mathbb{Q}] \\
[\![x]\!]_v &= v(x) \quad [x \in \mathbb{X}] \\
[\![e_1 + e_2]\!]_v &= [\![e_1]\!]_v \oplus [\![e_2]\!]_v \\
[\![e_1 - e_2]\!]_v &= [\![e_1]\!]_v \ominus [\![e_2]\!]_v \\
[\![e_1 * e_2]\!]_v &= [\![e_1]\!]_v \otimes [\![e_2]\!]_v
\end{aligned}
$$

$$
sign(c) = \begin{cases} - & \text{if } c < 0 \\ 0 & \text{if } c = 0 \\ + & \text{if } c > 0 \end{cases}
$$

"Abstract" Addition

$\oplus$	$\bot$	$-$	0	$+$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$-$	$\top$	$\top$
0	$\bot$	$-$	0	$+$	$\top$
$+$	$\bot$	$\top$	$+$	$+$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

Tables also required for:

- "abstract" subtraction
- "abstract" multiplication

Sign Analysis: Transfer Mapping

Need to define $[\![e]\!]$ for valuations v in $\mathbb{X} \to \{-, 0, +, \bot, \top\}$

Expressions: $[\![e]\!]_v$

$$[\![c]\!]_v = sign(c) \quad [c \in \mathbb{Q}]$$
$$[\![x]\!]_v = v(x) \quad [x \in \mathbb{X}]$$
$$[\![e_1 + e_2]\!]_v = [\![e_1]\!]_v \oplus [\![e_2]\!]_v$$
$$[\![e_1 - e_2]\!]_v = [\![e_1]\!]_v \ominus [\![e_2]\!]_v$$
$$[\![e_1 * e_2]\!]_v = [\![e_1]\!]_v \otimes [\![e_2]\!]_v$$

"Abstract" Addition

$\oplus$	$\bot$	$-$	0	$+$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$-$	$\top$	$\top$
0	$\bot$	$-$	0	$+$	$\top$
$+$	$\bot$	$\top$	$+$	$+$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

$$sign(c) = \begin{cases} - & \text{if } c < 0 \\ 0 & \text{if } c = 0 \\ + & \text{if } c > 0 \end{cases}$$

Tables also required for:

- "abstract" subtraction
- "abstract" multiplication

Sign Analysis: Transfer Mapping

Need to define $[\![e]\!]$ f[illegible]valuations v in $\mathbb{X} \rightarrow \{-, 0, +, \bot, \top\}$

Are these tables correct?

Expressions: $[\![e]\!]_v$

$$
\begin{array}{rcll}
[\![c]\!]_v & = & sign(c) & [c \in \mathbb{Q}] \\
[\![x]\!]_v & = & v(x) & [x \in \mathbb{X}] \\
[\![e_1 + e_2]\!]_v & = & [\![e_1]\!]_v \oplus [\![e_2]\!]_v & \\
[\![e_1 - e_2]\!]_v & = & [\![e_1]\!]_v \ominus [\![e_2]\!]_v & \\
[\![e_1 * e_2]\!]_v & = & [\![e_1]\!]_v \otimes [\![e_2]\!]_v &
\end{array}
$$

[illegible]stract” Addition

[illegible]	[illegible]	[illegible]	0	$+$	$\top$
$\bot$	[illegible]	[illegible]	[illegible]	[illegible]	$\bot$
$-$	$\bot$	[illegible]	[illegible]	[illegible]	[illegible]
0	$\bot$	$-$	[illegible]	[illegible]	[illegible]
$+$	$\bot$	$\top$	$+$	$+$	[illegible]
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

$$
sign(c) = \begin{cases} - & \text{if } c < 0 \\ 0 & \text{if } c = 0 \\ + & \text{if } c > 0 \end{cases}
$$

Tables also required for:

- “abstract” subtraction
- “abstract” multiplication

Sign Analysis: Transfer Mapping

Need to define $[\![e]\!]$ f… valuations v in $\mathbb{X} \rightarrow \{-, 0, +, \perp, \top\}$

Expressions: $[\![e]\!]_v$

$[\![c]\!]_v = sign(c) \quad [c \in \mathbb{Q}]$

$[\![x]\!]_v$ …

…stract" Addition

	0	+	⊤
			⊥

Are these tables correct?

Does this data flow instance really perform sign analysis?

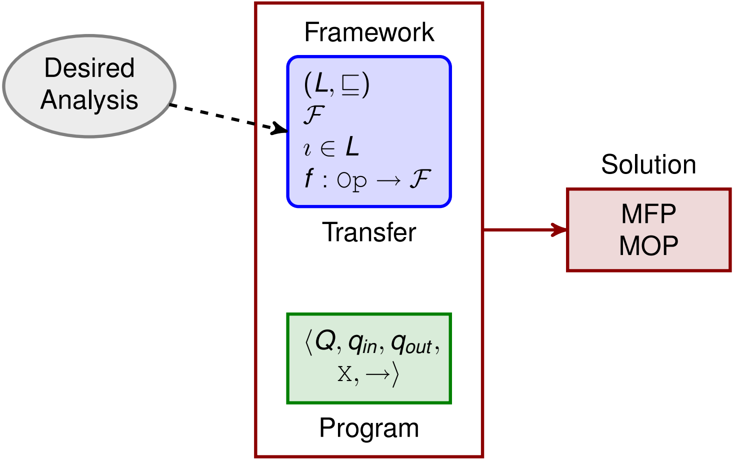
Is the analysis correct?

Is it precise?

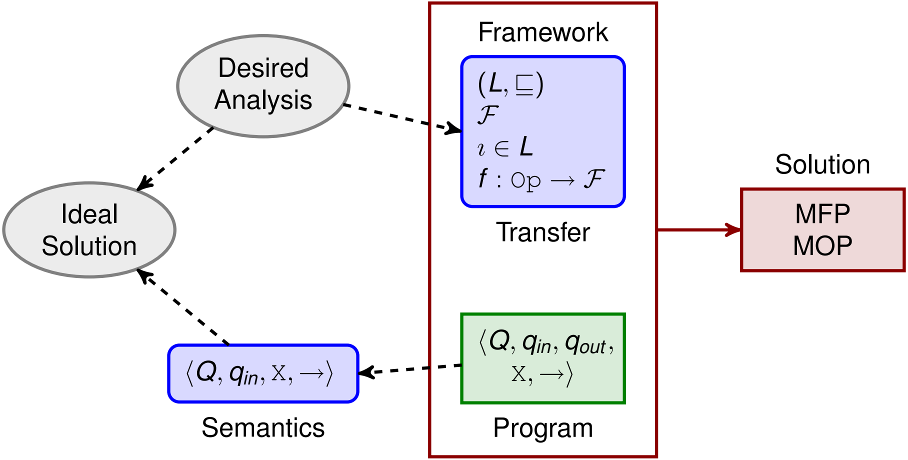
…tion

…ultiplication

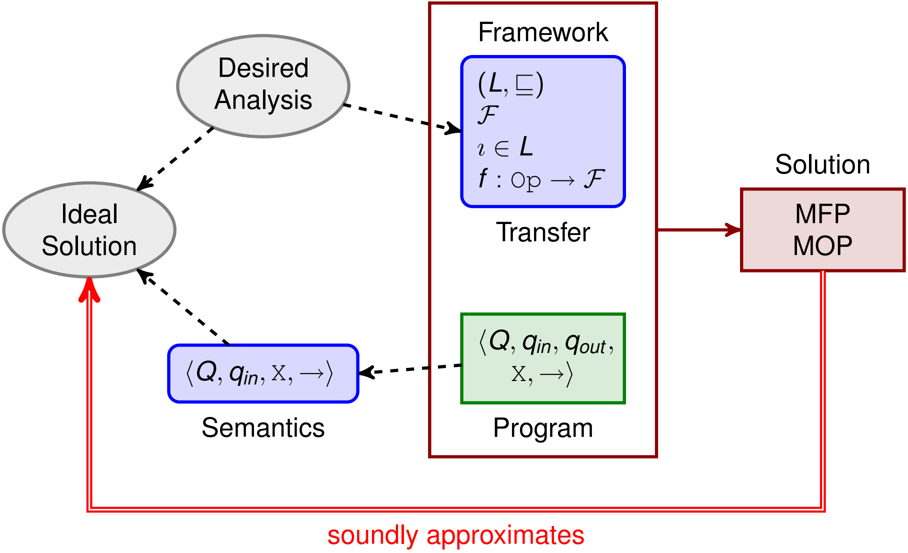
What About Correctness of Data Flow Analyses?



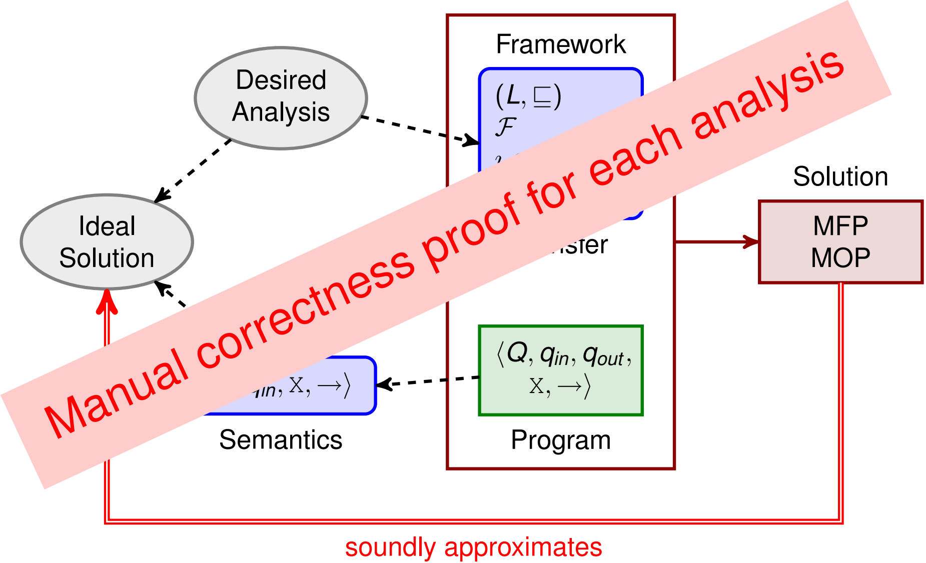
What About Correctness of Data Flow Analyses?



What About Correctness of Data Flow Analyses?



What About Correctness of Data Flow Analyses?



How to Systematically Ensure Correctness?

Data flow facts have an intended meaning.

The transfer mapping is designed according to this intended meaning.

We need a formal link to relate data flow facts and transfer functions with the formal semantics.

Solution: Abstract Interpretation

« This paper is devoted to the systematic and correct design of program analysis frameworks with respect to a formal semantics. »

P. Cousot & R. Cousot. Systematic Design of Program Analysis Frameworks. *Sixth Annual Symposium on Principles of Programming Languages*, 1979.

How to Systematically Ensure Correctness?

Data flow facts have an intended meaning.

The transfer mapping is designed according to this intended meaning.

We need a formal link to relate data flow facts and transfer functions with the formal semantics.

Solution: Abstract Interpretation

« This paper is devoted to the systematic and correct design of program analysis frameworks with respect to a formal semantics. »

P. Cousot & R. Cousot. Systematic Design of Program Analysis Frameworks. *Sixth Annual Symposium on Principles of Programming Languages*, 1979.