Static analysis

Software Analysis
Topic 5

Carlo A. Furia
USI – Università della Svizzera Italiana

Today’s menu

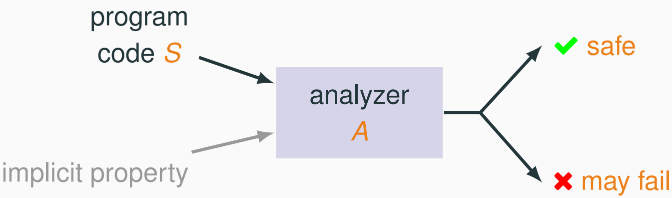
Data-flow analysis

Abstract interpretation

Type systems

Static analysis in practice

Static analysis: the very idea



Static analysis:

- analyzes real program code
- each analyzer targets a fixed set of hard-coded properties (compromise on flexibility)
- the output reports safe/unsafe for each program location individually
- is completely automatic
- is sound but incomplete

Static analysis: checked properties

The properties that are checked by static analysis are often general safety properties – stating the absence of errors of a certain kind:

- integer variables do not overflow
- there are no type errors
- there are no null-pointer dereferencing
- there are no out-of-bound array accesses
- there are no race conditions

Static analysis: this lecture

Static analysis is a vast field that has developed many techniques. Every software analysis technique that is static can be seen as a form of static analysis – although it may not be called that way.

Other names for the whole field are program analysis (which is often implicitly static by default) or software analysis.

Static analysis: this lecture

Static analysis is a vast field that has developed many techniques. Every software analysis technique that is static can be seen as a form of static analysis – although it may not be called that way.

Other names for the whole field are program analysis (which is often implicitly static by default) or software analysis.

In this lecture we have a look at three classic static analysis techniques:

data-flow analysis approximates the behavior of programs on their control-flow graph

abstract interpretation is a general framework to define and check the correctness of static analyses

type systems are a widely used form of static analysis to reason about the values expressions may have at runtime

Using static analysis

Static analysis has numerous applications:

avoiding bugs/verification: checking the absence of erroneous behavior such as overflows, division by zero, and out-of-bound array access

security: checking the enforcement of security properties such as non-interference

compiler optimization: improving the efficiency of programs at compile time based on the static information about their behavior

Using static analysis

Static analysis has numerous applications:

avoiding bugs/verification: checking the absence of erroneous behavior such as overflows, division by zero, and out-of-bound array access

security: checking the enforcement of security properties such as non-interference

compiler optimization: improving the efficiency of programs at compile time based on the static information about their behavior

> *The most important thing I have done as a programmer in recent years is to aggressively pursue static code analysis.*
>
> *John Carmack*

Static vs. dynamic

Static:

- at compile time – before execution
- related to a program's code, or to any other (formal) model of the software
- without executing the software
- on generic inputs

Dynamic:

- at run time – during execution
- related to a program's behavior
- while executing the software
- on specific inputs

Static vs. dynamic

Static:

- at compile time – before execution
- related to a program's code, or to any other (formal) model of the software
- without executing the software
- on generic inputs

Dynamic:

- at run time – during execution
- related to a program's behavior
- while executing the software
- on specific inputs

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.

Static vs. dynamic

Static:

- at compile time – before execution
- related to a program's code, or to any other (formal) model of the software
- without executing the software
- on generic inputs

Dynamic:

- at run time – during execution
- related to a program's behavior
- while executing the software
- on specific inputs

"Software analysis" denotes techniques, methods, and tools useful to establish that some software behaves according to some properties.

Therefore, static analysis infers properties of the dynamic behavior of programs <u>without</u> explicitly <u>running</u> them.

Static analysis: precision and expressiveness

Software analyses that target undecidable properties cannot be both sound and complete.

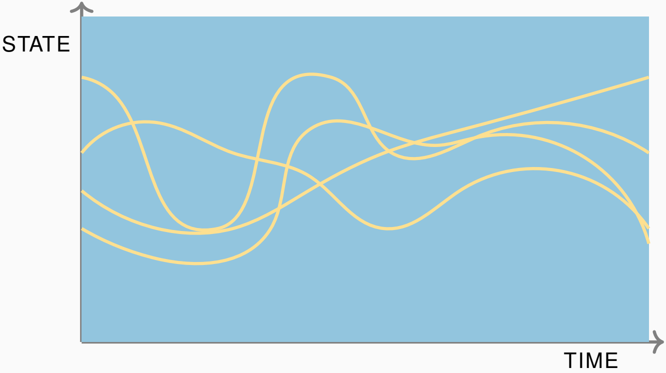
There is also a trade-off between soundness, expressiveness, and automation.

Static analysis:

- achieves soundness but gives up <u>completeness</u> – that is static analysis is imprecise
- targets fixed properties of certain kinds (such as control-flow properties) – thus giving up <u>expressiveness</u> while keeping automation

Program behavior

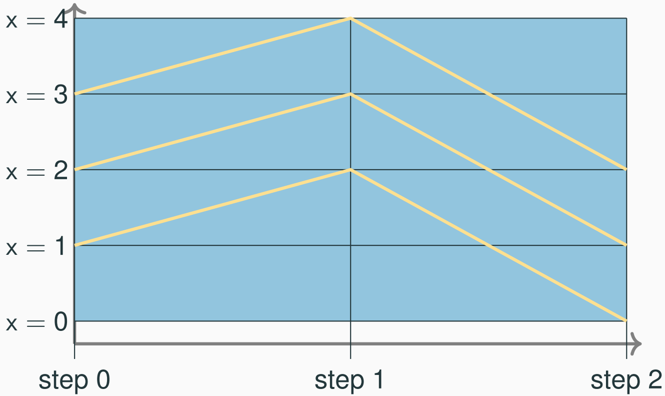
The (generic) behavior of a program consists of all its possible executions as sequences of states:



Each line is a different execution.

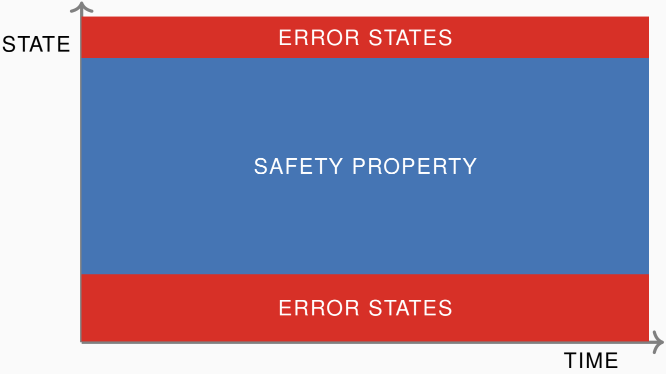
Program behavior: example

```
assume 1 ≤ x ≤ 3
// step 0
x := x + 1
// step 1
x := x - 2
// step 2
```



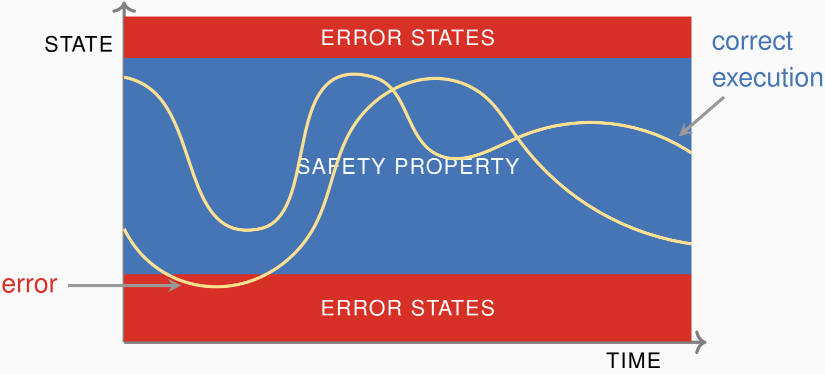
Safety properties and error states

A safety property is a set of program states that characterize correct executions. Its complement is the set of error states.



Safety properties and error states

A safety property is a set of program states that characterize correct executions. Its complement is the set of error states.



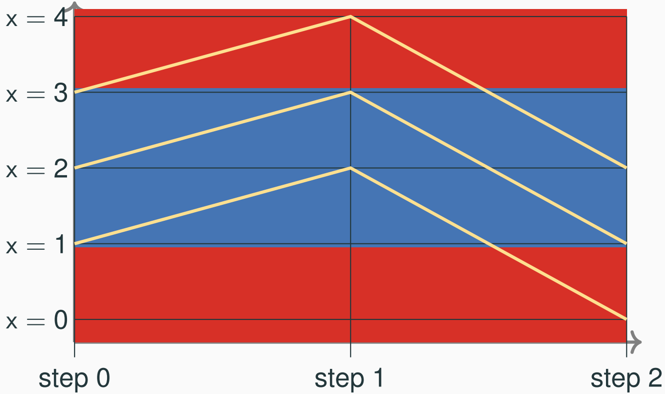
An execution is correct (safe) iff it never enters an error state.

Safety properties and error states: example

```
assume 1 ≤ x ≤ 3
// step 0
x := x + 1
// step 1
x := x - 2
// step 2
```

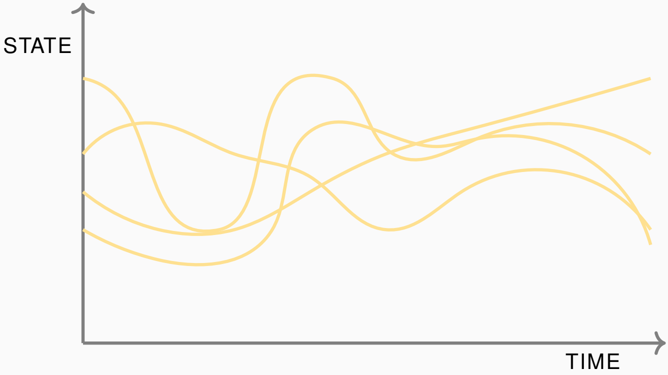
Safety property: $1 \leq x \leq 3$

Error states: $x < 1 \lor x > 3$



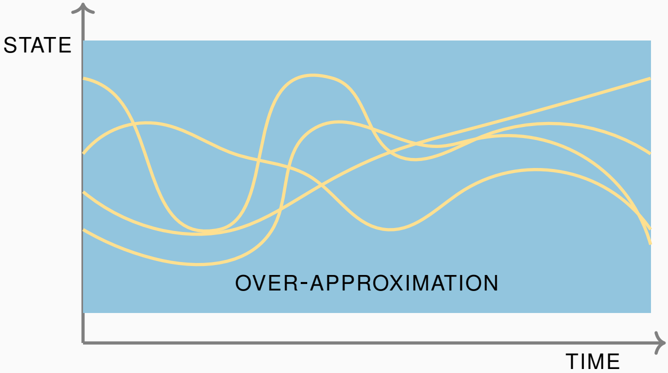
Approximations

An abstraction is an approximation of the behavior – typically in the form of a set of reachable states – which is easier to analyze than the concrete behavior.



Approximations

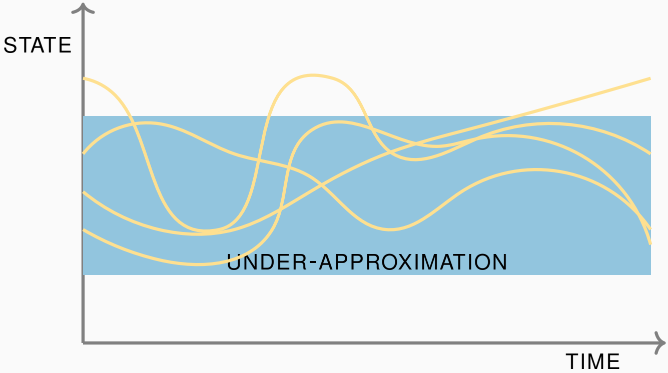
An abstraction is an approximation of the behavior – typically in the form of a set of reachable states – which is easier to analyze than the concrete behavior.



An over-approximation is a superset of all possible executions: it includes all concrete executions but may also include executions that are not feasible.

Approximations

An abstraction is an approximation of the behavior – typically in the form of a set of reachable states – which is easier to analyze than the concrete behavior.



An under-approximation is a subset of all possible executions: it includes no executions that are unfeasible, but may not include all concrete executions.

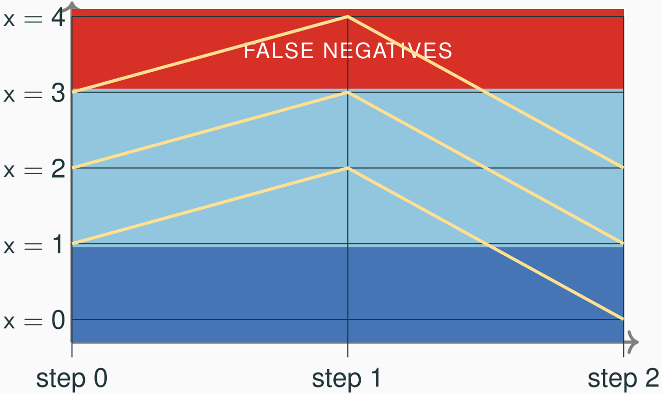
Under-approximation: example

```
assume 1 ≤ x ≤ 3
// step 0
x := x + 1
// step 1
x := x - 2
// step 2
```

Safety property: $x \leq 3$

Error states: $x > 3$

Under-approximation: $1 \leq x \leq 3$



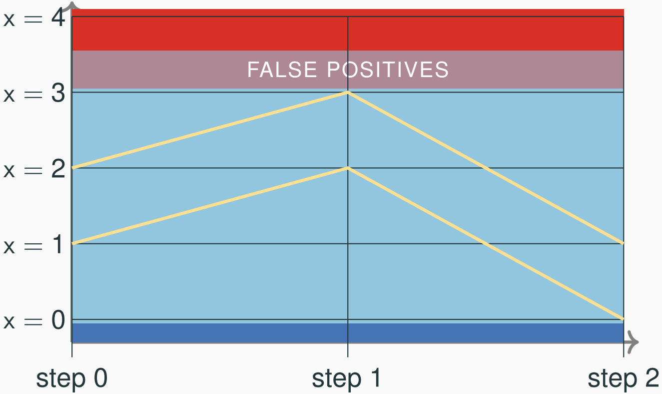
Over-approximation: example

```
assume 1 ≤ x ≤ 2
// step 0
x := x + 1
// step 1
x := x - 2
// step 2
```

Safety property: $x \leq 3$

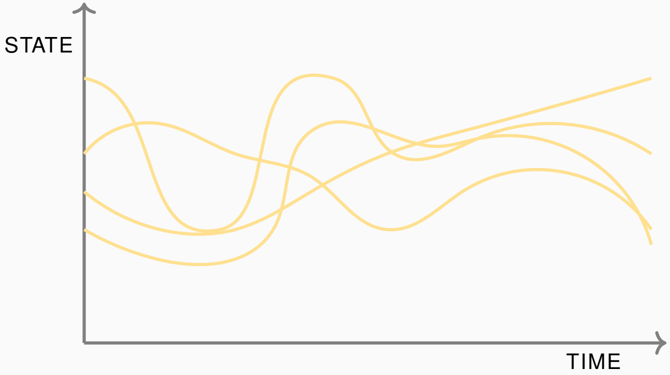
Error states: $x > 3$

Over-approximation: $0 \leq x \leq 3.5$



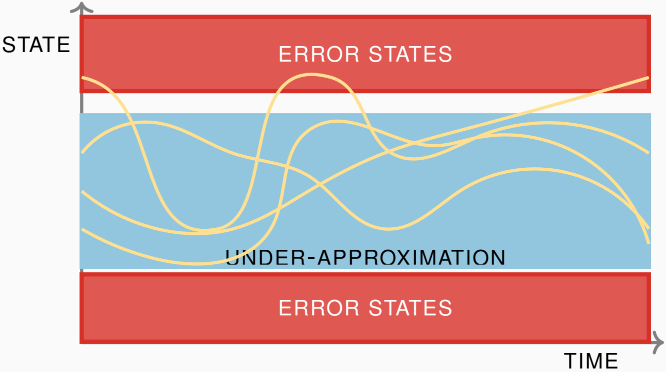
Soundness and precision

Static analysis is based on over-approximations to be sound – possibly sacrificing precision (completeness).



Soundness and precision

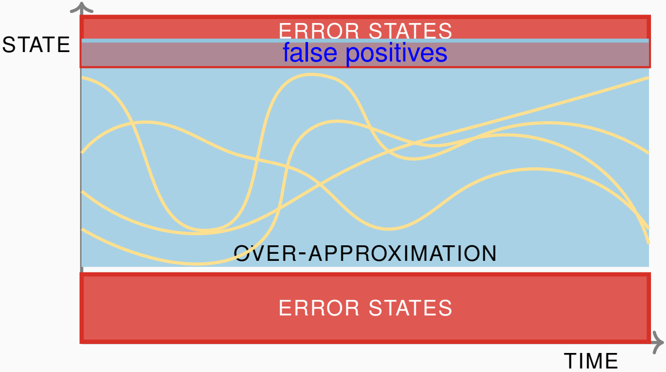
Static analysis is based on over-approximations to be sound – possibly sacrificing precision (completeness).



An analysis based on under-approximations is unsound: it may miss errors (generate false negatives).

Soundness and precision

Static analysis is based on over-approximations to be sound – possibly sacrificing precision (completeness).



An analysis based on over-approximations is imprecise: it may report spurious errors (generate false positives).

Precision vs. efficiency

When designing a static analysis, precision is often traded-off against efficiency:

- perfect precision is often impossible due to undecidability
- even for decidable problems, high precision may still be too computationally expensive
- low precision leads to many false positives, which users have to identified as such manually

Precision vs. efficiency

When designing a static analysis, precision is often traded-off against efficiency:

- perfect precision is often impossible due to undecidability
- even for decidable problems, high precision may still be too computationally expensive
- low precision leads to many false positives, which users have to identified as such manually

Designing a static analysis requires to balance precision and efficiency in a way that is practical.

Data-flow analysis

What is a data-flow analysis

A data-flow analysis is a kind of static analysis that:

- works on the control-flow graph of the input program
- derives information about the data flow: what values are read (used) and written (defined) at specific program points

control-flow graph S $\longrightarrow$ data-flow analyzer A $\longrightarrow$

at ℓ_1: $L_1 = \{x, y\}$
at ℓ_2: $L_2 = \{z\}$
$\vdots$
at ℓ_n: $L_n = \{\ \}$

What is a data-flow analysis

A data-flow analysis is a kind of static analysis that:

- works on the control-flow graph of the input program
- derives information about the data flow: what values are read (used) and written (defined) at specific program points

control-flow graph S $\longrightarrow$ data-flow analyzer A $\longrightarrow$

at ℓ_1: $L_1 = \{x, y\}$
at ℓ_2: $L_2 = \{z\}$
$\vdots$
at ℓ_n: $L_n = \{\ \}$

The property under analysis is derivable from the analysis's output.
Example: live variables analysis.

output: for each program point which variables are live – will be read before being overwritten

property: is variable v live at ℓ_k? Check if $v \in L_k$

Data-flow analysis

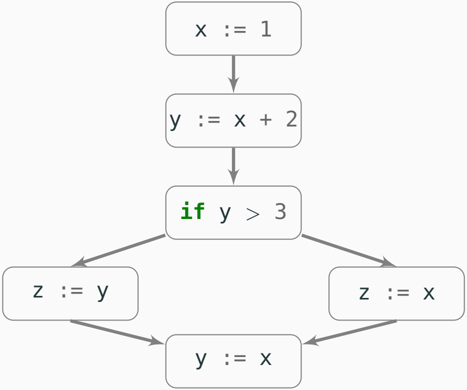
Control-flow graphs

Control-flow graphs

The control-flow graph (CFG) of a program is a directed graph representing possible execution paths:

- each statement corresponds to a node in the graph
- edges connect nodes of consecutive statements

```
x := 1
y := x + 2
if y > 3
  z := y
else
  z := x
y := x
```



Control-flow graphs of Helium programs

We define the control-flow graphs of Helium programs – ignoring declarations since they do not affect the program state which is what static analysis targets.

Control-flow graphs of Helium programs

We define the control-flow graphs of Helium programs – ignoring declarations since they do not affect the program state which is what static analysis targets.

Each atomic statement corresponds to a single CFG node.

skip $\boxed{\texttt{skip}}$

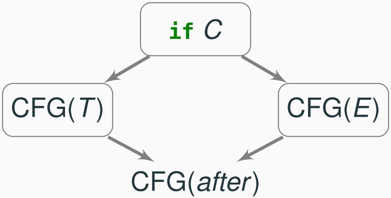
$v_1, \ldots, v_n := E_1, \ldots, E_n$ $\boxed{v_1, \ldots, v_n := E_1, \ldots, E_n}$

Control-flow graphs of Helium programs

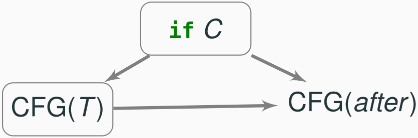
We define the control-flow graphs of Helium programs – ignoring declarations since they do not affect the program state which is what static analysis targets.

Each conditional statement introduces a branch.

`if` *C T* **`else`** *E* ; *after*



`if` *C T* ; *after*

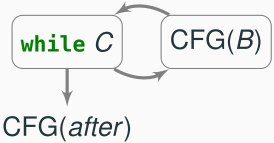


Control-flow graphs of Helium programs

We define the control-flow graphs of Helium programs – ignoring declarations since they do not affect the program state which is what static analysis targets.

Each loop statement introduces a branch and a loop.

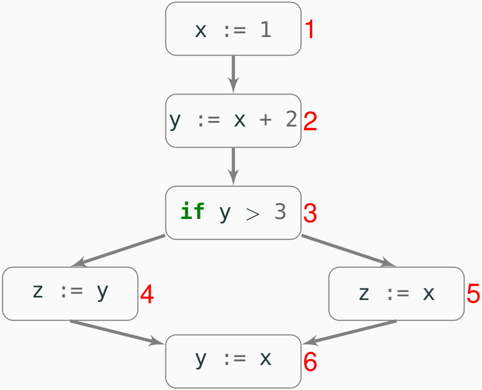
while *C B* ; *after*



Labels

We label statements (and expressions of conditionals and loops) to be able to refer to specific program points.

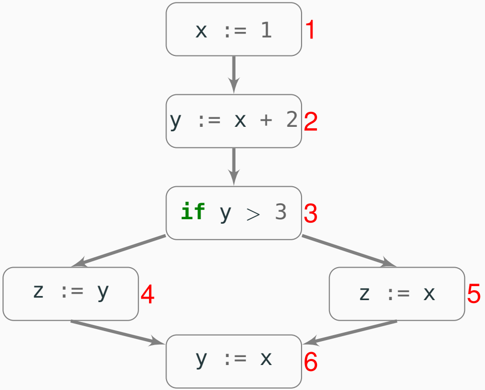
```
{ x := 1 }1
{ y := x + 2 }2
if ( y > 3 )3
  { z := y }4
else
  { z := x }5
{ y := x }6
```



Labels

We label statements (and expressions of conditionals and loops) to be able to refer to specific program points.

```
{ x := 1 }1
{ y := x + 2 }2
if ( y > 3 )3
  { z := y }4
else
  { z := x }5
{ y := x }6
```



elementary block: a labeled node in the CFG (also: program point)

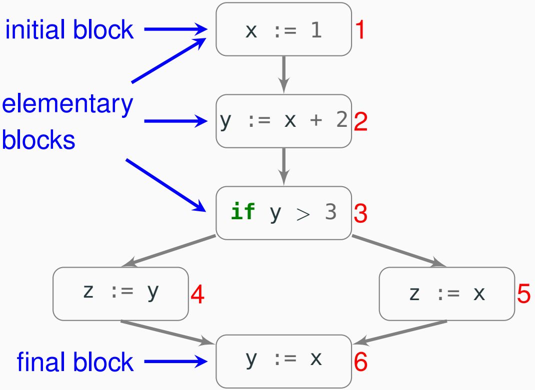
initial block: block where execution begins

final block: block after which execution terminates

Labels

We label statements (and expressions of conditionals and loops) to be able to refer to specific program points.

`{ x := 1 }`^{1}
`{ y := x + 2 }`^{2}
`if` `( y > 3 )`^{3}
  `{ z := y }`^{4}
`else`
  `{ z := x }`^{5}
`{ y := x }`^{6}



elementary block: a labeled node in the CFG (also: program point)

initial block: block where execution begins

final block: block after which execution terminates

Data-flow analysis

Live variables analysis

Live variables

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

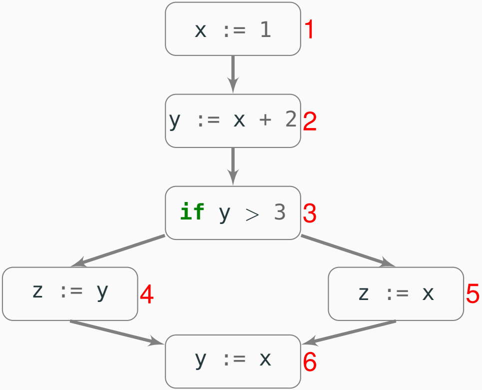
read v

write v

Live variables

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

read v write v



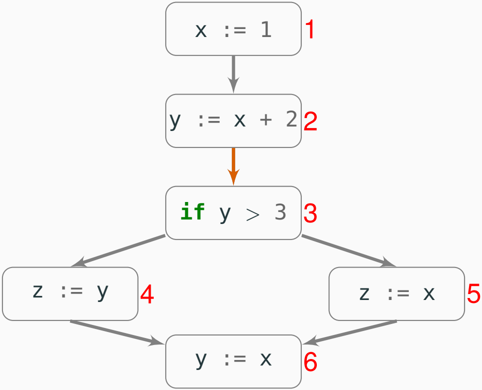
Examples:

- y at 2:
- z at 4:

Live variables

A variable *v* is live at the exit from a block if there is some path (on the CFG) from the block to a use of *v* that does not redefine *v*.

read *v* — write *v*



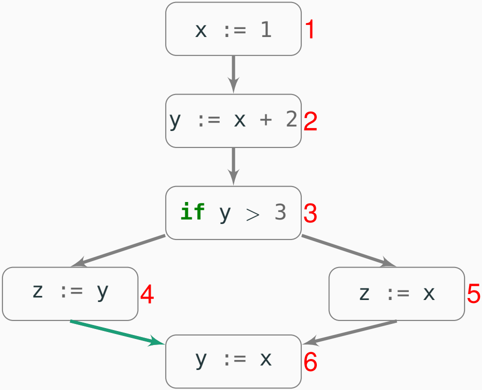
Examples:

- y at 2: live
- z at 4:

Live variables

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

read v — write v



Examples:

- y at 2: live
- z at 4: not live

Live variables analysis

A variable v is live at the <u>exit</u> from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

> Live variables analysis: for each program point, determine which variables may be live at the point.

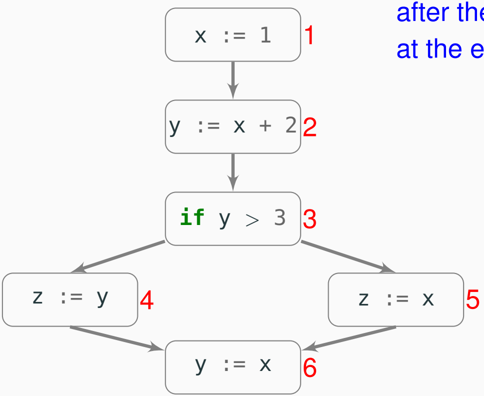
after the point/
at the exit from the block

Live variables analysis

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

> Live variables analysis: for each program point, determine which variables may be live at the point.

after the point/
at the exit from the block



Live variables analysis's output:

$$LV(1) = \{\ldots\}$$
$$LV(2) = \{y, \ldots\}$$
$$\vdots$$
$$LV(6) = \{\}$$

Live variables analysis

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

> Live variables analysis: for each program point, determine which variables may be live at the point.

over-approximation

after the point/
at the exit from the block

A may analysis is an over-approximation:
$LV(k)$ is a superset of the live variables at k.

- if $\mathtt{x} \in LV(k)$ x may or may not be live at k (for example because it is live along certain paths but not live along others)
- if $\mathtt{y} \notin LV(k)$ y is definitely not live at k

The analysis has to be sound, and then as precise as possible given the information available in the CFG.

Live variables analysis: applications

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

If a variable `v` is not live after it is defined in an assignment, the assignment is useless and can be removed without changing program behavior.

Dead assignment elimination: any block k such that:

1. k is an assignment to variable `v`
2. `v` is not live at k – that is, $k \notin LV(k)$

can be eliminated without affecting program behavior.

Live variables analysis: applications

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

If a variable v is not live after it is defined in an assignment, the assignment is useless and can be removed without changing program behavior.

Dead assignment elimination: any block k such that:

1. k is an assignment to variable v
2. v is not live at k – that is, $k \notin LV(k)$

can be eliminated without affecting program behavior.

```
{ x := 4 }1
{ x := 7 }2
if z > y
  y := x
```

Live variables analysis: applications

A variable v is live at the exit from a block if there is some path (on the CFG) from the block to a use of v that does not redefine v.

If a variable v is not live after it is defined in an assignment, the assignment is useless and can be removed without changing program behavior.

Dead assignment elimination: any block k such that:

1. k is an assignment to variable v
2. v is not live at k – that is, $k \notin LV(k)$

can be eliminated without affecting program behavior.

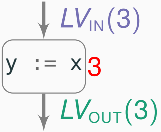
```
{ x := 4 }1
{ x := 7 }2
if z > y
  y := x
```

x not live here: the assignment is useless

x may be live here: the assignment is possibly useful

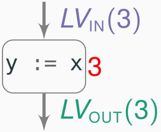
Live variables analysis: idea

Record the possibly live variables at the entry and exit of every elementary block.



Live variables analysis: idea

Record the possibly live variables at the entry and exit of every elementary block.



For each block, relate LV_{IN} to LV_{OUT}.

Live variables analysis: idea

Record the possibly live variables at the entry and exit of every elementary block.

$LV_{IN}(3)$

`y := x` 3

$LV_{OUT}(3)$

For each block, relate LV_{IN} to LV_{OUT}.

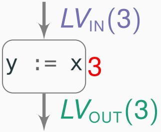
$LV_{IN}(3) = \{x, z\}$

`y := x` 3

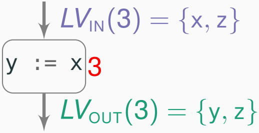
$LV_{OUT}(3) = \{y, z\}$

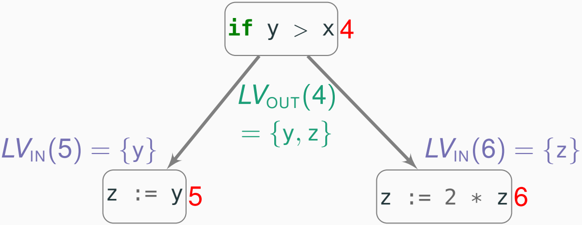
Live variables analysis: idea

Record the possibly live variables at the entry and exit of every elementary block.



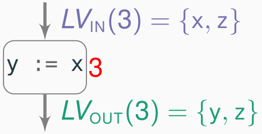
For each block, relate LV_{IN} to LV_{OUT}.

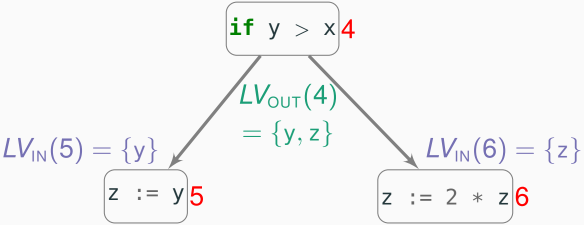




Live variables analysis: idea

For each block, relate LV_{IN} to LV_{OUT}.





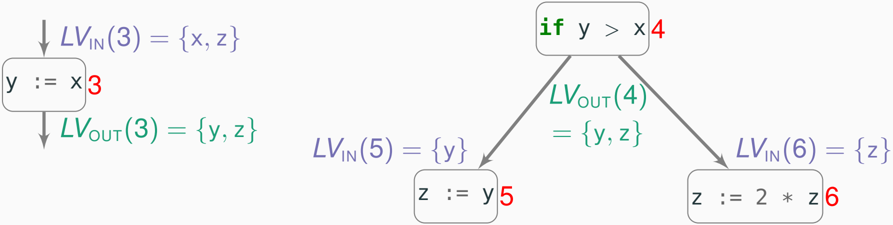
Work backward from the exit block to the entry block.

$$LV_{\text{IN}}(k) = (LV_{\text{OUT}}(k) \setminus \text{“assigned at } k\text{”}) \cup \text{“used at } k\text{”}$$

$$LV_{\text{OUT}}(k) = \bigcup_{h \text{ direct successor of } k} LV_{\text{IN}}(h)$$

Live variables analysis: idea

For each block, relate LV_{IN} to LV_{OUT}.



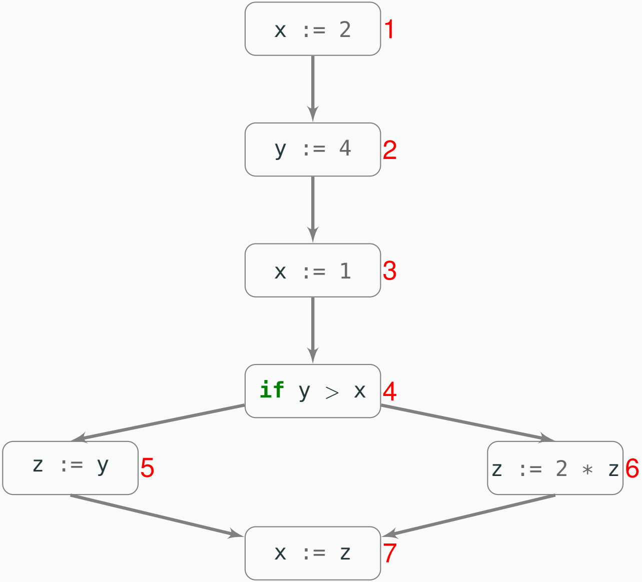
Work backward from the exit block to the entry block.

$$LV_{\text{IN}}(k) = (LV_{\text{OUT}}(k) \setminus \text{“assigned at } k\text{”}) \cup \text{“used at } k\text{”}$$

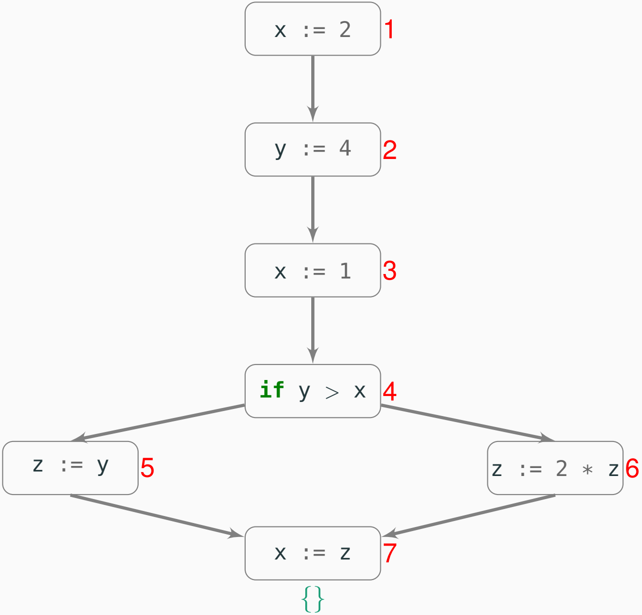
$$LV_{\text{OUT}}(k) = \bigcup_{h \text{ direct successor of } k} LV_{\text{IN}}(h)$$

The analysis's final output is: $LV(1) = LV_{\text{OUT}}(1), \ldots, LV(n) = LV_{\text{OUT}}(n)$

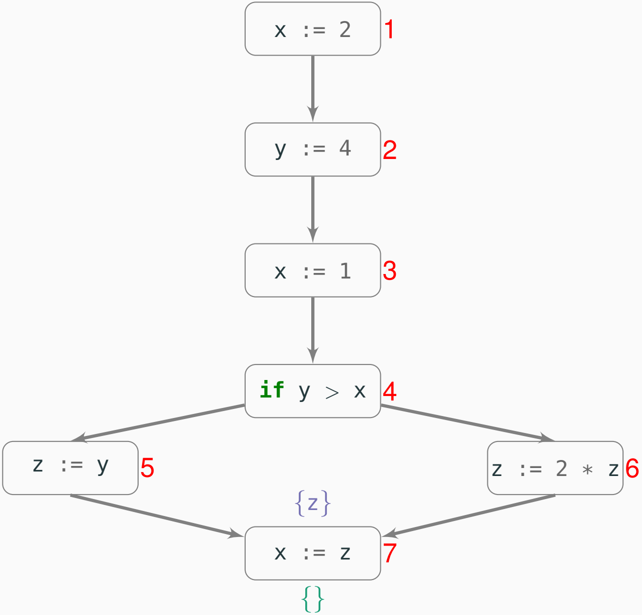
Live variables analysis: example



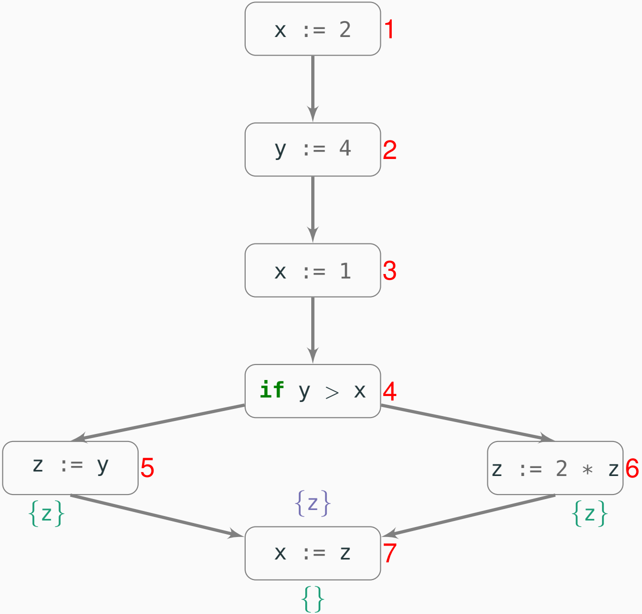
Live variables analysis: example



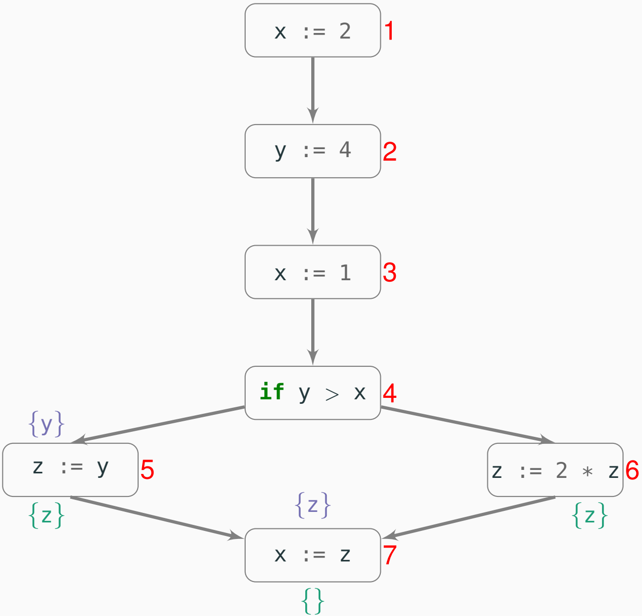
Live variables analysis: example



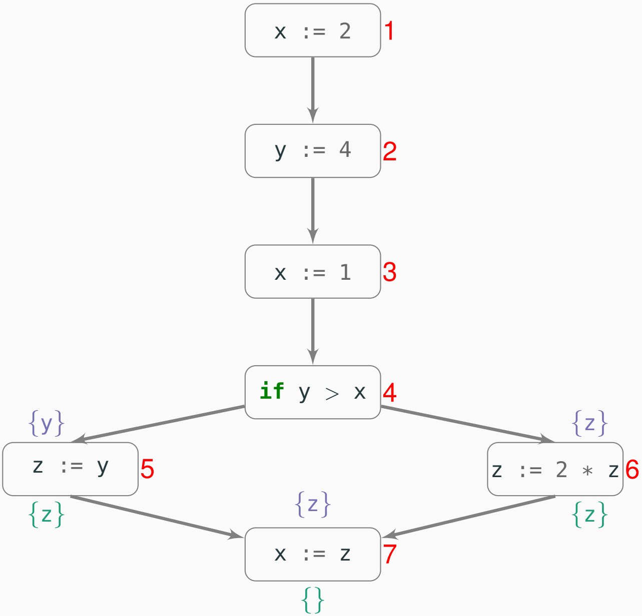
Live variables analysis: example



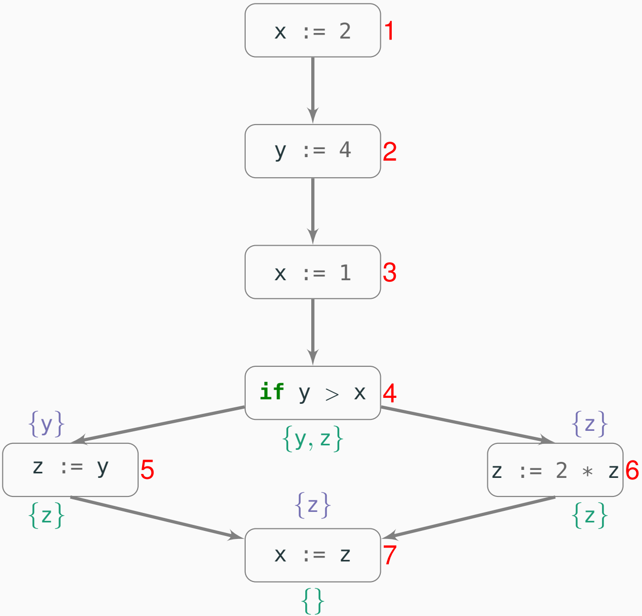
Live variables analysis: example



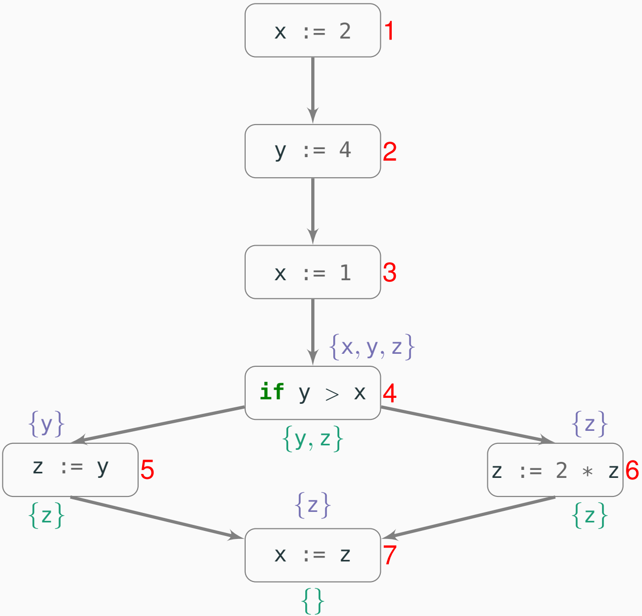
Live variables analysis: example



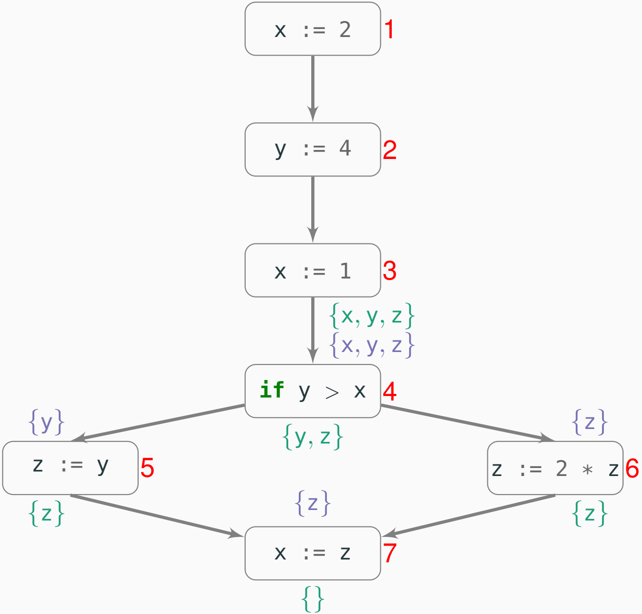
Live variables analysis: example



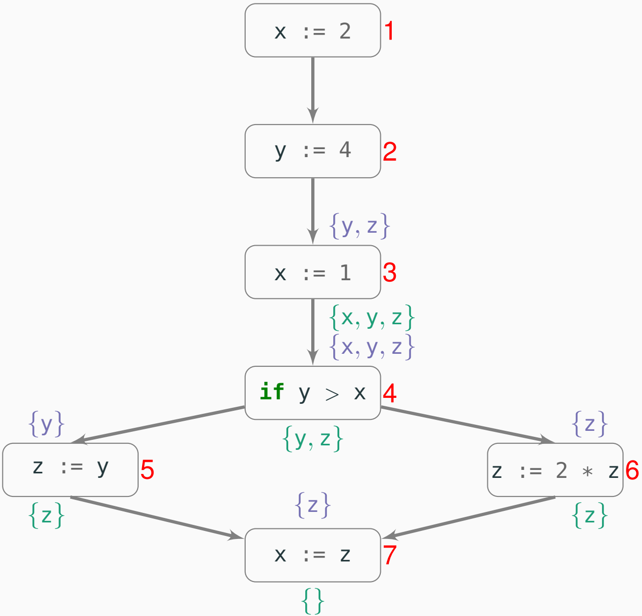
Live variables analysis: example



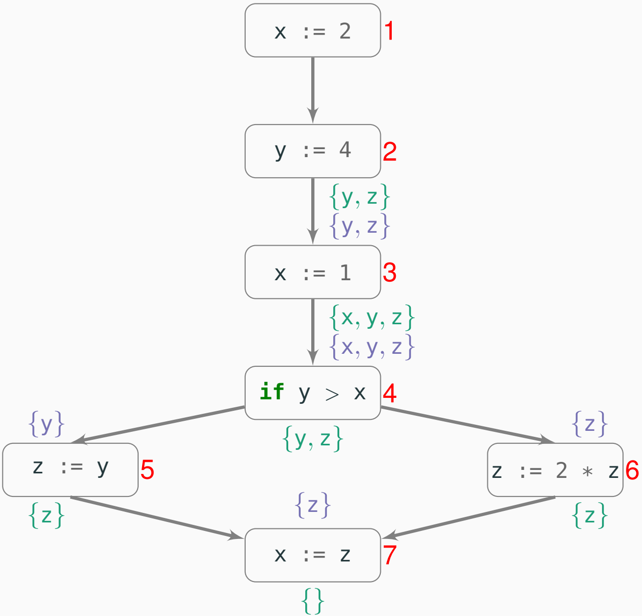
Live variables analysis: example



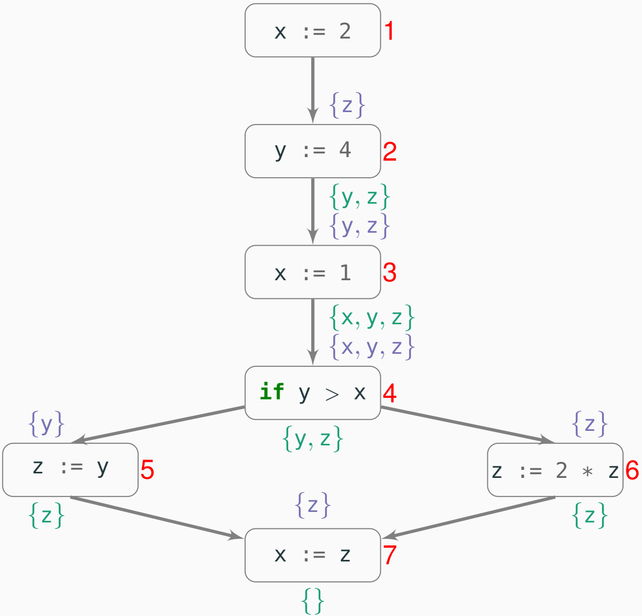
Live variables analysis: example



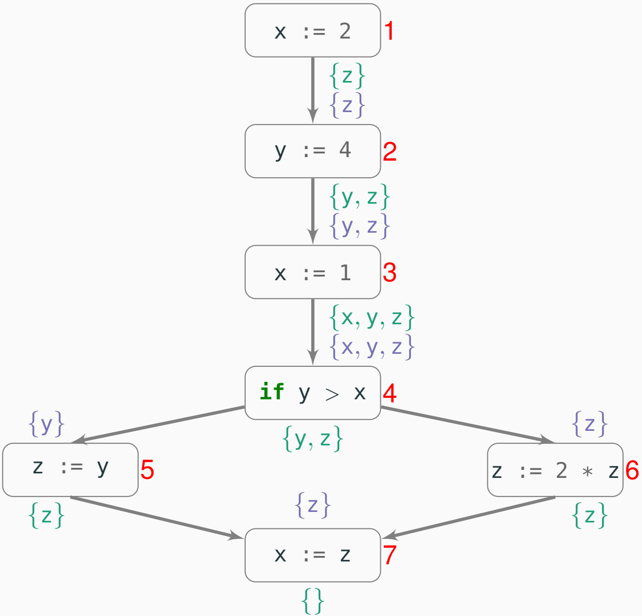
Live variables analysis: example



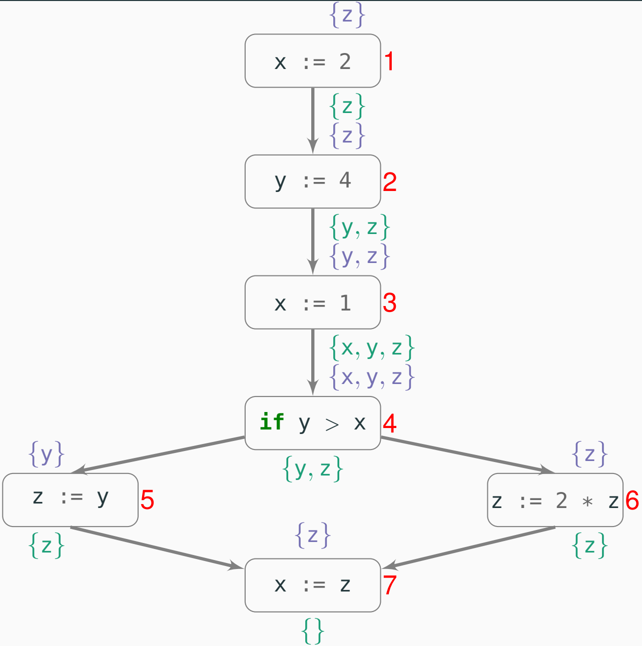
Live variables analysis: example



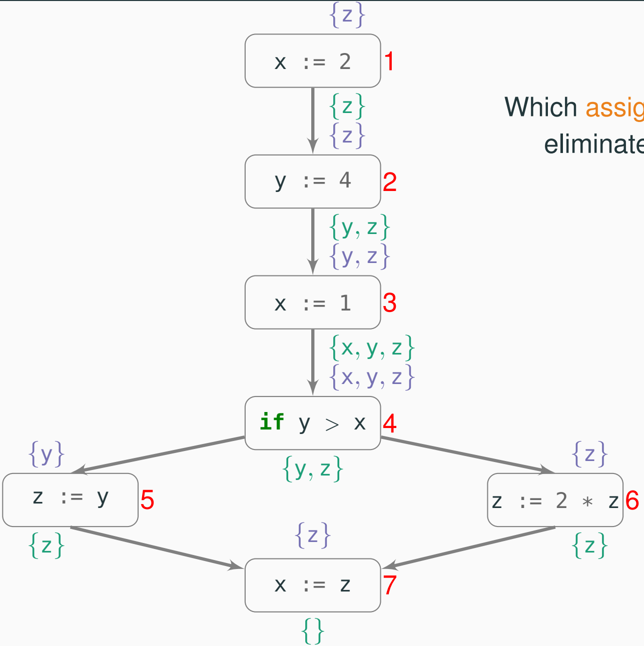
Live variables analysis: example



Live variables analysis: example

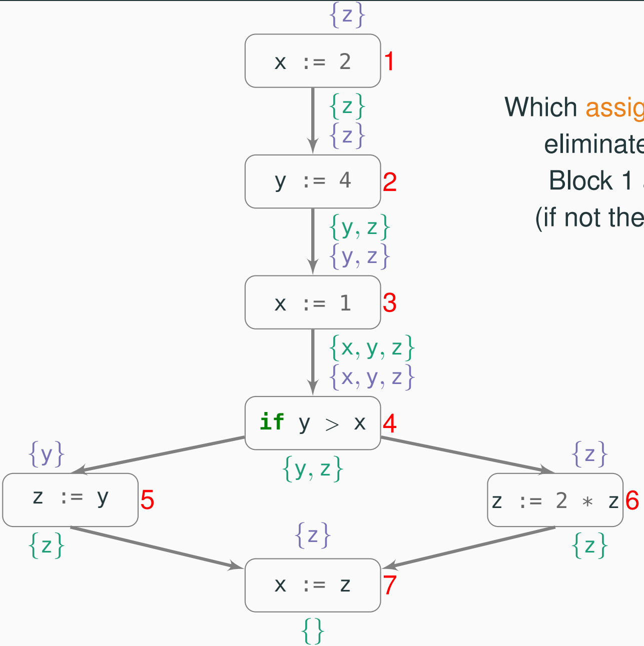


Live variables analysis: example



Which assignments can be eliminated as dead?

Live variables analysis: example



Which assignments can be eliminated as dead?

Block 1 and block 7

(if not the final output)

Formalizing data-flow analyses

We formalize the idea of live variables analysis as an equation system:

- $LV_{\text{IN}}(k)$ and $LV_{\text{OUT}}(k)$ are variables
- the equations formalize the relations:

$$\begin{aligned} LV_{\text{IN}}(k) &= \quad (LV_{\text{OUT}}(k) \setminus \text{“assigned at } k\text{”}) \cup \text{“used at } k\text{”} \\ LV_{\text{OUT}}(k) &= \bigcup_{h \text{ direct successor of } k} LV_{\text{IN}}(h) \end{aligned}$$

for every possible block type (assignment or branch condition)

The analysis result is the solution of the equation system, which can be computed using standard algorithms.

Data-flow equations

for every node h that follows k in the CFG (i.e., h is a direct successor of k)

For every block k:

$$LV_{\text{OUT}}(k) = \bigcup_{(k \to h) \in \text{CFG}} LV_{\text{IN}}(h)$$

$$LV_{\text{IN}}(k) = \left(LV_{\text{OUT}}(k) \setminus \text{kill}_{LV}(k)\right) \cup \text{gen}_{LV}(k)$$

variables assigned at k

variables used at k

Data-flow equations

for every node h that follows k in the CFG
(i.e., h is a direct successor of k)

For every block k:

$$LV_{\text{OUT}}(k) = \bigcup_{(k \to h) \in \text{CFG}} LV_{\text{IN}}(h)$$

$$LV_{\text{IN}}(k) = \big(LV_{\text{OUT}}(k) \setminus \text{kill}_{LV}(k)\big) \cup \text{gen}_{LV}(k)$$

variables assigned at k

variables used at k

If f is a final node, it has no successors, and hence $LV_{\text{OUT}}(f) = \{\}$.

Data-flow equations

for every node h that follows k in the CFG (i.e., h is a direct successor of k)

For every block k:

$$LV_{\text{OUT}}(k) = \bigcup_{(k \to h) \in \text{CFG}} LV_{\text{IN}}(h)$$

$$LV_{\text{IN}}(k) = \big(LV_{\text{OUT}}(k) \setminus \text{kill}_{LV}(k)\big) \cup \text{gen}_{LV}(k)$$

variables assigned at k

variables used at k

If f is a final node, it has no successors, and hence $LV_{\text{OUT}}(f) = \{\}$.

We define kill_{LV} and gen_{LV} for every block type:

$$\text{kill}_{LV}(\texttt{skip}) = \{\} \qquad \text{gen}_{LV}(\texttt{skip}) = \{\}$$

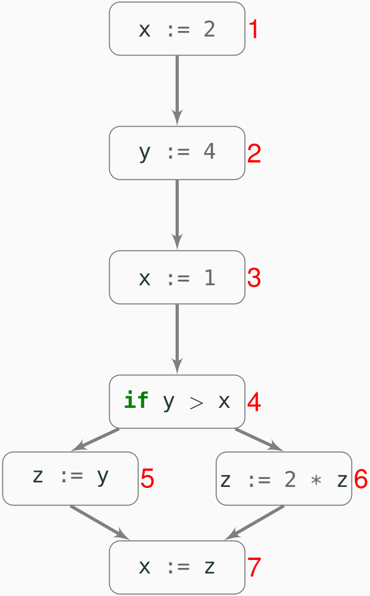
$$\text{kill}_{LV}(\texttt{v := } E) = \{\texttt{v}\} \qquad \text{gen}_{LV}(\texttt{v := } E) = \{\texttt{x} \mid \texttt{x} \text{ is a (free) variable in } E\}$$

$$\text{kill}_{LV}(\texttt{if}/\texttt{while } C) = \{\} \qquad \text{gen}_{LV}(\texttt{if}/\texttt{while } C) = \{\texttt{x} \mid \texttt{x} \text{ is a (free) variable in } C\}$$

Equation system for live variables analysis: example

$$LV_{\text{OUT}}(1) = LV_{\text{IN}}(2)$$
$$LV_{\text{OUT}}(2) = LV_{\text{IN}}(3)$$
$$LV_{\text{OUT}}(3) = LV_{\text{IN}}(4)$$
$$LV_{\text{OUT}}(4) = LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6)$$
$$LV_{\text{OUT}}(5) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(6) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(7) = \{\}$$

$$LV_{\text{IN}}(1) = LV_{\text{OUT}}(1) \setminus \{x\}$$
$$LV_{\text{IN}}(2) = LV_{\text{OUT}}(2) \setminus \{y\}$$
$$LV_{\text{IN}}(3) = LV_{\text{OUT}}(3) \setminus \{x\}$$
$$LV_{\text{IN}}(4) = LV_{\text{OUT}}(4) \cup \{x, y\}$$
$$LV_{\text{IN}}(5) = (LV_{\text{OUT}}(5) \setminus \{z\}) \cup \{y\}$$
$$LV_{\text{IN}}(6) = (LV_{\text{OUT}}(6) \setminus \{z\}) \cup \{z\}$$
$$LV_{\text{IN}}(7) = (LV_{\text{OUT}}(7) \setminus \{x\}) \cup \{z\}$$



Data-flow analysis

Equation solving

Vector equations

$$LV_{\text{OUT}}(1) = LV_{\text{IN}}(2)$$
$$LV_{\text{OUT}}(2) = LV_{\text{IN}}(3)$$
$$LV_{\text{OUT}}(3) = LV_{\text{IN}}(4)$$
$$LV_{\text{OUT}}(4) = LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6)$$
$$LV_{\text{OUT}}(5) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(6) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(7) = \{\}$$

$$LV_{\text{IN}}(1) = LV_{\text{OUT}}(1) \setminus \{x\}$$
$$LV_{\text{IN}}(2) = LV_{\text{OUT}}(2) \setminus \{y\}$$
$$LV_{\text{IN}}(3) = LV_{\text{OUT}}(3) \setminus \{x\}$$
$$LV_{\text{IN}}(4) = LV_{\text{OUT}}(4) \cup \{x, y\}$$
$$LV_{\text{IN}}(5) = (LV_{\text{OUT}}(5) \setminus \{z\}) \cup \{y\}$$
$$LV_{\text{IN}}(6) = (LV_{\text{OUT}}(6) \setminus \{z\}) \cup \{z\}$$
$$LV_{\text{IN}}(7) = (LV_{\text{OUT}}(7) \setminus \{x\}) \cup \{z\}$$

Vector equations

$$LV_{\text{OUT}}(1) = LV_{\text{IN}}(2)$$
$$LV_{\text{OUT}}(2) = LV_{\text{IN}}(3)$$
$$LV_{\text{OUT}}(3) = LV_{\text{IN}}(4)$$
$$LV_{\text{OUT}}(4) = LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6)$$
$$LV_{\text{OUT}}(5) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(6) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(7) = \{\}$$

$$LV_{\text{IN}}(1) = LV_{\text{OUT}}(1) \setminus \{\mathrm{x}\}$$
$$LV_{\text{IN}}(2) = LV_{\text{OUT}}(2) \setminus \{\mathrm{y}\}$$
$$LV_{\text{IN}}(3) = LV_{\text{OUT}}(3) \setminus \{\mathrm{x}\}$$
$$LV_{\text{IN}}(4) = LV_{\text{OUT}}(4) \cup \{\mathrm{x}, \mathrm{y}\}$$
$$LV_{\text{IN}}(5) = (LV_{\text{OUT}}(5) \setminus \{\mathrm{z}\}) \cup \{\mathrm{y}\}$$
$$LV_{\text{IN}}(6) = (LV_{\text{OUT}}(6) \setminus \{\mathrm{z}\}) \cup \{\mathrm{z}\}$$
$$LV_{\text{IN}}(7) = (LV_{\text{OUT}}(7) \setminus \{\mathrm{x}\}) \cup \{\mathrm{z}\}$$

The equations over variables $LV_{\text{OUT}}(1), LV_{\text{OUT}}(2), \ldots, LV_{\text{IN}}(7)$ are formally equivalent to equations over set variables $X_1, \ldots, X_{14}$.

Vector equations

$$
\begin{aligned}
LV_{\text{OUT}}(1) &= LV_{\text{IN}}(2) \\
LV_{\text{OUT}}(2) &= LV_{\text{IN}}(3) \\
LV_{\text{OUT}}(3) &= LV_{\text{IN}}(4) \\
LV_{\text{OUT}}(4) &= LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6) \\
LV_{\text{OUT}}(5) &= LV_{\text{IN}}(7) \\
LV_{\text{OUT}}(6) &= LV_{\text{IN}}(7) \\
LV_{\text{OUT}}(7) &= \{\}
\end{aligned}
\qquad
\begin{aligned}
X_1 &= LV_{\text{IN}}(2) \\
X_2 &= LV_{\text{IN}}(3) \\
X_3 &= LV_{\text{IN}}(4) \\
X_4 &= LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6) \\
X_5 &= LV_{\text{IN}}(7) \\
X_6 &= LV_{\text{IN}}(7) \\
X_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
LV_{\text{IN}}(1) &= LV_{\text{OUT}}(1) \setminus \{x\} \\
LV_{\text{IN}}(2) &= LV_{\text{OUT}}(2) \setminus \{y\} \\
LV_{\text{IN}}(3) &= LV_{\text{OUT}}(3) \setminus \{x\} \\
LV_{\text{IN}}(4) &= LV_{\text{OUT}}(4) \cup \{x, y\} \\
LV_{\text{IN}}(5) &= (LV_{\text{OUT}}(5) \setminus \{z\}) \cup \{y\} \\
LV_{\text{IN}}(6) &= (LV_{\text{OUT}}(6) \setminus \{z\}) \cup \{z\} \\
LV_{\text{IN}}(7) &= (LV_{\text{OUT}}(7) \setminus \{x\}) \cup \{z\}
\end{aligned}
\qquad
\begin{aligned}
X_8 &= LV_{\text{OUT}}(1) \setminus \{x\} \\
X_9 &= LV_{\text{OUT}}(2) \setminus \{y\} \\
X_{10} &= LV_{\text{OUT}}(3) \setminus \{x\} \\
X_{11} &= LV_{\text{OUT}}(4) \cup \{x, y\} \\
X_{12} &= (LV_{\text{OUT}}(5) \setminus \{z\}) \cup \{y\} \\
X_{13} &= (LV_{\text{OUT}}(6) \setminus \{z\}) \cup \{z\} \\
X_{14} &= (LV_{\text{OUT}}(7) \setminus \{x\}) \cup \{z\}
\end{aligned}
$$

Vector equations

$$
\begin{aligned}
LV_{\text{OUT}}(1) &= LV_{\text{IN}}(2) & X_1 &= X_9 \\
LV_{\text{OUT}}(2) &= LV_{\text{IN}}(3) & X_2 &= X_{10} \\
LV_{\text{OUT}}(3) &= LV_{\text{IN}}(4) & X_3 &= X_{11} \\
LV_{\text{OUT}}(4) &= LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6) & X_4 &= X_{12} \cup X_{13} \\
LV_{\text{OUT}}(5) &= LV_{\text{IN}}(7) & X_5 &= X_{14} \\
LV_{\text{OUT}}(6) &= LV_{\text{IN}}(7) & X_6 &= X_{14} \\
LV_{\text{OUT}}(7) &= \{\} & X_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
LV_{\text{IN}}(1) &= LV_{\text{OUT}}(1) \setminus \{x\} & X_8 &= X_1 \setminus \{x\} \\
LV_{\text{IN}}(2) &= LV_{\text{OUT}}(2) \setminus \{y\} & X_9 &= X_2 \setminus \{y\} \\
LV_{\text{IN}}(3) &= LV_{\text{OUT}}(3) \setminus \{x\} & X_{10} &= X_3 \setminus \{x\} \\
LV_{\text{IN}}(4) &= LV_{\text{OUT}}(4) \cup \{x, y\} & X_{11} &= X_4 \cup \{x, y\} \\
LV_{\text{IN}}(5) &= (LV_{\text{OUT}}(5) \setminus \{z\}) \cup \{y\} & X_{12} &= (X_5 \setminus \{z\}) \cup \{y\} \\
LV_{\text{IN}}(6) &= (LV_{\text{OUT}}(6) \setminus \{z\}) \cup \{z\} & X_{13} &= (X_6 \setminus \{z\}) \cup \{z\} \\
LV_{\text{IN}}(7) &= (LV_{\text{OUT}}(7) \setminus \{x\}) \cup \{z\} & X_{14} &= (X_7 \setminus \{x\}) \cup \{z\}
\end{aligned}
$$

Vector equations

$$\begin{aligned}
LV_{\text{OUT}}(1) &= LV_{\text{IN}}(2) & X_1 &= F_1(X_1, \ldots, X_{14}) \\
LV_{\text{OUT}}(2) &= LV_{\text{IN}}(3) & X_2 &= F_2(X_1, \ldots, X_{14}) \\
LV_{\text{OUT}}(3) &= LV_{\text{IN}}(4) & X_3 &= F_3(X_1, \ldots, X_{14}) \\
LV_{\text{OUT}}(4) &= LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6) & X_4 &= F_4(X_1, \ldots, X_{14}) \\
LV_{\text{OUT}}(5) &= LV_{\text{IN}}(7) & X_5 &= F_5(X_1, \ldots, X_{14}) \\
LV_{\text{OUT}}(6) &= LV_{\text{IN}}(7) & X_6 &= F_6(X_1, \ldots, X_{14}) \\
LV_{\text{OUT}}(7) &= \{\} & X_7 &= F_7(X_1, \ldots, X_{14}) \\
LV_{\text{IN}}(1) &= LV_{\text{OUT}}(1) \setminus \{\mathrm{x}\} & X_8 &= F_8(X_1, \ldots, X_{14}) \\
LV_{\text{IN}}(2) &= LV_{\text{OUT}}(2) \setminus \{\mathrm{y}\} & X_9 &= F_9(X_1, \ldots, X_{14}) \\
LV_{\text{IN}}(3) &= LV_{\text{OUT}}(3) \setminus \{\mathrm{x}\} & X_{10} &= F_{10}(X_1, \ldots, X_{14}) \\
LV_{\text{IN}}(4) &= LV_{\text{OUT}}(4) \cup \{\mathrm{x}, \mathrm{y}\} & X_{11} &= F_{11}(X_1, \ldots, X_{14}) \\
LV_{\text{IN}}(5) &= (LV_{\text{OUT}}(5) \setminus \{\mathrm{z}\}) \cup \{\mathrm{y}\} & X_{12} &= F_{12}(X_1, \ldots, X_{14}) \\
LV_{\text{IN}}(6) &= (LV_{\text{OUT}}(6) \setminus \{\mathrm{z}\}) \cup \{\mathrm{z}\} & X_{13} &= F_{13}(X_1, \ldots, X_{14}) \\
LV_{\text{IN}}(7) &= (LV_{\text{OUT}}(7) \setminus \{\mathrm{x}\}) \cup \{\mathrm{z}\} & X_{14} &= F_{14}(X_1, \ldots, X_{14})
\end{aligned}$$

Vector equations

$$LV_{\text{OUT}}(1) = LV_{\text{IN}}(2)$$
$$LV_{\text{OUT}}(2) = LV_{\text{IN}}(3)$$
$$LV_{\text{OUT}}(3) = LV_{\text{IN}}(4)$$
$$LV_{\text{OUT}}(4) = LV_{\text{IN}}(5) \cup LV_{\text{IN}}(6)$$
$$LV_{\text{OUT}}(5) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(6) = LV_{\text{IN}}(7)$$
$$LV_{\text{OUT}}(7) = \{\}$$

$$LV_{\text{IN}}(1) = LV_{\text{OUT}}(1) \setminus \{\text{x}\}$$
$$LV_{\text{IN}}(2) = LV_{\text{OUT}}(2) \setminus \{\text{y}\}$$
$$LV_{\text{IN}}(3) = LV_{\text{OUT}}(3) \setminus \{\text{x}\}$$
$$LV_{\text{IN}}(4) = LV_{\text{OUT}}(4) \cup \{\text{x}, \text{y}\}$$
$$LV_{\text{IN}}(5) = (LV_{\text{OUT}}(5) \setminus \{\text{z}\}) \cup \{\text{y}\}$$
$$LV_{\text{IN}}(6) = (LV_{\text{OUT}}(6) \setminus \{\text{z}\}) \cup \{\text{z}\}$$
$$LV_{\text{IN}}(7) = (LV_{\text{OUT}}(7) \setminus \{\text{x}\}) \cup \{\text{z}\}$$

$$\vec{X} = F(\vec{X})$$

- $\vec{X} = X_1, \ldots, X_{14}$ is a vector of variables
- F is a vector function whose components are $F_1, \ldots, F_{14}$

Least solutions

$$
\begin{aligned}
X_1 &= X_9 \\
X_2 &= X_{10} \\
X_3 &= X_{11} \\
X_4 &= X_{12} \cup X_{13} \\
X_5 &= X_{14} \\
X_6 &= X_{14} \\
X_7 &= \{\} \\
X_8 &= X_1 \setminus \{x\} \\
X_9 &= X_2 \setminus \{y\} \\
X_{10} &= X_3 \setminus \{x\} \\
X_{11} &= X_4 \cup \{x, y\} \\
X_{12} &= (X_5 \setminus \{z\}) \cup \{y\} \\
X_{13} &= (X_6 \setminus \{z\}) \cup \{z\} \\
X_{14} &= (X_7 \setminus \{x\}) \cup \{z\}
\end{aligned}
$$

We already know a solution to this set of equations:

$$
\begin{aligned}
X_7 &= \{\} \\
X_8, X_1, X_9, X_{13}, X_5, X_6, X_{14} &= \{z\} \\
X_{12} &= \{y\} \\
X_2, X_{10}, X_4 &= \{y, z\} \\
X_3, X_{11} &= \{x, y, z\}
\end{aligned}
$$

Least solutions

$$
\begin{aligned}
X_1 &= X_9 \\
X_2 &= X_{10} \\
X_3 &= X_{11} \\
X_4 &= X_{12} \cup X_{13} \\
X_5 &= X_{14} \\
X_6 &= X_{14} \\
X_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
X_8 &= X_1 \setminus \{x\} \\
X_9 &= X_2 \setminus \{y\} \\
X_{10} &= X_3 \setminus \{x\} \\
X_{11} &= X_4 \cup \{x, y\} \\
X_{12} &= (X_5 \setminus \{z\}) \cup \{y\} \\
X_{13} &= (X_6 \setminus \{z\}) \cup \{z\} \\
X_{14} &= (X_7 \setminus \{x\}) \cup \{z\}
\end{aligned}
$$

We already know a solution to this set of equations:

$$
\begin{aligned}
X_7 &= \{\} \\
X_8, X_1, X_9, X_{13}, X_5, X_6, X_{14} &= \{z\} \\
X_{12} &= \{y\} \\
X_2, X_{10}, X_4 &= \{y, z\} \\
X_3, X_{11} &= \{x, y, z\}
\end{aligned}
$$

In this particular case, the equation system admits only this solution. However, in more general cases, there may be more than one solution. Intuitively, we want the least solution – that is the solution with the smallest sets – because it corresponds to a more precise (less conservative) analysis.

Partially ordered sets

By introducing an ordering of sets we can order solutions to data-flow equations from small to large – that is from more to less precise.

A partial ordering is a relation $\sqsubseteq$ that is:

reflexive: $\forall d \bullet (d \sqsubseteq d)$

transitive: $\forall c, d, e \bullet (c \sqsubseteq d \wedge d \sqsubseteq e \Longrightarrow c \sqsubseteq e)$

anti-symmetric: $\forall c, d \bullet (c \sqsubseteq d \wedge d \sqsubseteq c \Longrightarrow c = d)$

A partially ordered set (poset) $\langle D, \sqsubseteq \rangle$ is a set D whose elements are partially ordered according to $\sqsubseteq$.

Some familiar examples of posets:

$\langle \mathbb{R}, \leq \rangle$	the real numbers with the usual order
$\langle \mathbb{N}, \leq \rangle$	the natural numbers (nonnegative integers) with the usual order
$\langle \wp(S), \subseteq \rangle$	the power set $\wp(S)$ of S with the subset order

Fixed points

$X_1 =$
$X_2 =$
$X_3 =$
$X_4 =$
$X_5 =$
$X_6 =$
$X_7 =$
$X_8 =$
$X_9 =$
$X_{10} =$
$X_{11} =$
$X_{12} =$
$X_{13} =$
$X_{14} =$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

	$\vec{X}^0$
$X_1 =$	$\{\}$
$X_2 =$	$\{\}$
$X_3 =$	$\{\}$
$X_4 =$	$\{\}$
$X_5 =$	$\{\}$
$X_6 =$	$\{\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{\}$
$X_{10} =$	$\{\}$
$X_{11} =$	$\{\}$
$X_{12} =$	$\{\}$
$X_{13} =$	$\{\}$
$X_{14} =$	$\{\}$

Each variable $X_1, \dots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

	$F(\vec{X}^0)$
$X_1 =$	$\{\}$
$X_2 =$	$\{\}$
$X_3 =$	$\{\}$
$X_4 =$	$\{\} \cup \{\}$
$X_5 =$	$\{\}$
$X_6 =$	$\{\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\} \setminus \{x\}$
$X_9 =$	$\{\} \setminus \{y\}$
$X_{10} =$	$\{\} \setminus \{x\}$
$X_{11} =$	$\{\} \cup \{x, y\}$
$X_{12} =$	$(\{\} \setminus \{z\}) \cup \{y\}$
$X_{13} =$	$(\{\} \setminus \{z\}) \cup \{z\}$
$X_{14} =$	$(\{\} \setminus \{x\}) \cup \{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$\vec{X}^1$	
$X_1 =$	$\{\}$
$X_2 =$	$\{\}$
$X_3 =$	$\{\}$
$X_4 =$	$\{\}$
$X_5 =$	$\{\}$
$X_6 =$	$\{\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{\}$
$X_{10} =$	$\{\}$
$X_{11} =$	$\{x, y\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$F(\vec{X}^1)$$

$$\begin{aligned}
X_1 &= \{\} \\
X_2 &= \{\} \\
X_3 &= \{x, y\} \\
X_4 &= \{y\} \cup \{z\} \\
X_5 &= \{z\} \\
X_6 &= \{z\} \\
X_7 &= \{\} \\
X_8 &= \{\} \setminus \{x\} \\
X_9 &= \{\} \setminus \{y\} \\
X_{10} &= \{\} \setminus \{x\} \\
X_{11} &= \{\} \cup \{x, y\} \\
X_{12} &= (\{\} \setminus \{z\}) \cup \{y\} \\
X_{13} &= (\{\} \setminus \{z\}) \cup \{z\} \\
X_{14} &= (\{\} \setminus \{x\}) \cup \{z\}
\end{aligned}$$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

	$\vec{X}^2$
$X_1 =$	$\{\}$
$X_2 =$	$\{\}$
$X_3 =$	$\{x, y\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{\}$
$X_{10} =$	$\{\}$
$X_{11} =$	$\{x, y\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$F(\vec{X}^2)$	
$X_1 =$	$\{\}$
$X_2 =$	$\{\}$
$X_3 =$	$\{x, y\}$
$X_4 =$	$\{y\} \cup \{z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\} \setminus \{x\}$
$X_9 =$	$\{\} \setminus \{y\}$
$X_{10} =$	$\{x, y\} \setminus \{x\}$
$X_{11} =$	$\{y, z\} \cup \{x, y\}$
$X_{12} =$	$(\{z\} \setminus \{z\}) \cup \{y\}$
$X_{13} =$	$(\{z\} \setminus \{z\}) \cup \{z\}$
$X_{14} =$	$(\{\} \setminus \{x\}) \cup \{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$\vec{X}^3$	
$X_1 =$	$\{\}$
$X_2 =$	$\{\}$
$X_3 =$	$\{x, y\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{\}$
$X_{10} =$	$\{y\}$
$X_{11} =$	$\{x, y, z\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$F(\vec{X}^3)$$

$$\begin{aligned}
X_1 &= \{\} \\
X_2 &= \{y\} \\
X_3 &= \{x, y, z\} \\
X_4 &= \{y\} \cup \{z\} \\
X_5 &= \{z\} \\
X_6 &= \{z\} \\
X_7 &= \{\} \\
X_8 &= \{\} \setminus \{x\} \\
X_9 &= \{\} \setminus \{y\} \\
X_{10} &= \{x, y\} \setminus \{x\} \\
X_{11} &= \{y, z\} \cup \{x, y\} \\
X_{12} &= (\{z\} \setminus \{z\}) \cup \{y\} \\
X_{13} &= (\{z\} \setminus \{z\}) \cup \{z\} \\
X_{14} &= (\{\} \setminus \{x\}) \cup \{z\}
\end{aligned}$$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$\vec{X}^4$	
$X_1 =$	$\{\}$
$X_2 =$	$\{y\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{\}$
$X_{10} =$	$\{y\}$
$X_{11} =$	$\{x, y, z\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$
\begin{array}{rl}
& F(\vec{X}^4) \\
\hline
X_1 = & \{\} \\
X_2 = & \{y\} \\
X_3 = & \{x, y, z\} \\
X_4 = & \{y\} \cup \{z\} \\
X_5 = & \{z\} \\
X_6 = & \{z\} \\
X_7 = & \{\} \\
X_8 = & \{\} \setminus \{x\} \\
X_9 = & \{y\} \setminus \{y\} \\
X_{10} = & \{x, y, z\} \setminus \{x\} \\
X_{11} = & \{y, z\} \cup \{x, y\} \\
X_{12} = & (\{z\} \setminus \{z\}) \cup \{y\} \\
X_{13} = & (\{z\} \setminus \{z\}) \cup \{z\} \\
X_{14} = & (\{\} \setminus \{x\}) \cup \{z\}
\end{array}
$$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$\vec{X}^5$	
$X_1 =$	$\{\}$
$X_2 =$	$\{y\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{\}$
$X_{10} =$	$\{y, z\}$
$X_{11} =$	$\{x, y, z\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \dots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k(X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$F(\vec{X}^5)$$

$$\begin{aligned}
X_1 &= \{\} \\
X_2 &= \{y, z\} \\
X_3 &= \{x, y, z\} \\
X_4 &= \{y\} \cup \{z\} \\
X_5 &= \{z\} \\
X_6 &= \{z\} \\
X_7 &= \{\} \\
X_8 &= \{\} \setminus \{x\} \\
X_9 &= \{y\} \setminus \{y\} \\
X_{10} &= \{x, y, z\} \setminus \{x\} \\
X_{11} &= \{y, z\} \cup \{x, y\} \\
X_{12} &= (\{z\} \setminus \{z\}) \cup \{y\} \\
X_{13} &= (\{z\} \setminus \{z\}) \cup \{z\} \\
X_{14} &= (\{\} \setminus \{x\}) \cup \{z\}
\end{aligned}$$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$\vec{X}^6$$

$$\begin{aligned}
X_1 &= \{\} \\
X_2 &= \{y, z\} \\
X_3 &= \{x, y, z\} \\
X_4 &= \{y, z\} \\
X_5 &= \{z\} \\
X_6 &= \{z\} \\
X_7 &= \{\} \\
X_8 &= \{\} \\
X_9 &= \{\} \\
X_{10} &= \{y, z\} \\
X_{11} &= \{x, y, z\} \\
X_{12} &= \{y\} \\
X_{13} &= \{z\} \\
X_{14} &= \{z\}
\end{aligned}$$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$F(\vec{X}^6)$$

$$
\begin{aligned}
X_1 &= \{\} \\
X_2 &= \{y, z\} \\
X_3 &= \{x, y, z\} \\
X_4 &= \{y\} \cup \{z\} \\
X_5 &= \{z\} \\
X_6 &= \{z\} \\
X_7 &= \{\} \\
X_8 &= \{\} \setminus \{x\} \\
X_9 &= \{y, z\} \setminus \{y\} \\
X_{10} &= \{x, y, z\} \setminus \{x\} \\
X_{11} &= \{y, z\} \cup \{x, y\} \\
X_{12} &= (\{z\} \setminus \{z\}) \cup \{y\} \\
X_{13} &= (\{z\} \setminus \{z\}) \cup \{z\} \\
X_{14} &= (\{\} \setminus \{x\}) \cup \{z\}
\end{aligned}
$$

Each variable $X_1, \dots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$\vec{X}^7$$

$X_1 =$	$\{\}$
$X_2 =$	$\{y, z\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{z\}$
$X_{10} =$	$\{y, z\}$
$X_{11} =$	$\{x, y, z\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k(X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

	$F(\vec{X}^7)$
$X_1 =$	$\{z\}$
$X_2 =$	$\{y, z\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y\} \cup \{z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\} \setminus \{x\}$
$X_9 =$	$\{y, z\} \setminus \{y\}$
$X_{10} =$	$\{x, y, z\} \setminus \{x\}$
$X_{11} =$	$\{y, z\} \cup \{x, y\}$
$X_{12} =$	$(\{z\} \setminus \{z\}) \cup \{y\}$
$X_{13} =$	$(\{z\} \setminus \{z\}) \cup \{z\}$
$X_{14} =$	$(\{\} \setminus \{x\}) \cup \{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$\vec{X}^8$$

$X_1 =$	$\{z\}$
$X_2 =$	$\{y, z\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{\}$
$X_9 =$	$\{z\}$
$X_{10} =$	$\{y, z\}$
$X_{11} =$	$\{x, y, z\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$$F(\vec{X}^8)$$

$$\begin{aligned}
X_1 &= \{z\} \\
X_2 &= \{y, z\} \\
X_3 &= \{x, y, z\} \\
X_4 &= \{y\} \cup \{z\} \\
X_5 &= \{z\} \\
X_6 &= \{z\} \\
X_7 &= \{\} \\
X_8 &= \{z\} \setminus \{x\} \\
X_9 &= \{y, z\} \setminus \{y\} \\
X_{10} &= \{x, y, z\} \setminus \{x\} \\
X_{11} &= \{y, z\} \cup \{x, y\} \\
X_{12} &= (\{z\} \setminus \{z\}) \cup \{y\} \\
X_{13} &= (\{z\} \setminus \{z\}) \cup \{z\} \\
X_{14} &= (\{\} \setminus \{x\}) \cup \{z\}
\end{aligned}$$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

	$\vec{X}^9$
$X_1 =$	$\{z\}$
$X_2 =$	$\{y, z\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{z\}$
$X_9 =$	$\{z\}$
$X_{10} =$	$\{y, z\}$
$X_{11} =$	$\{x, y, z\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

	$F(\vec{X}^9)$
$X_1 =$	$\{z\}$
$X_2 =$	$\{y, z\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y\} \cup \{z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{z\} \setminus \{x\}$
$X_9 =$	$\{y, z\} \setminus \{y\}$
$X_{10} =$	$\{x, y, z\} \setminus \{x\}$
$X_{11} =$	$\{y, z\} \cup \{x, y\}$
$X_{12} =$	$(\{z\} \setminus \{z\}) \cup \{y\}$
$X_{13} =$	$(\{z\} \setminus \{z\}) \cup \{z\}$
$X_{14} =$	$(\{\} \setminus \{x\}) \cup \{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Fixed points

$\vec{X}^{10}$	
$X_1 =$	$\{z\}$
$X_2 =$	$\{y, z\}$
$X_3 =$	$\{x, y, z\}$
$X_4 =$	$\{y, z\}$
$X_5 =$	$\{z\}$
$X_6 =$	$\{z\}$
$X_7 =$	$\{\}$
$X_8 =$	$\{z\}$
$X_9 =$	$\{z\}$
$X_{10} =$	$\{y, z\}$
$X_{11} =$	$\{x, y, z\}$
$X_{12} =$	$\{y\}$
$X_{13} =$	$\{z\}$
$X_{14} =$	$\{z\}$

Each variable $X_1, \ldots, X_{13}$ ranges over the poset $\langle \wp(\{x, y, z\}), \subseteq \rangle$.

Vector variable $\vec{X}$ also ranges over the poset $\langle \wp(\{x, y, z\})^{14}, \sqsubseteq \rangle$, where:

$$\vec{X} \sqsubseteq \vec{Y} \quad \text{iff} \quad \forall k (X_k \subseteq Y_k)$$

We can find a solution as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

A value $\vec{X}^k$ such that $F(\vec{X}^k) = \vec{X}^k$ is called a fixed point of F.

Least fixed points and monotonicity

We can find a solution of the data-flow equations as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

We are interested in the least fixed point of F – that is the smallest according to $\sqsubseteq$.

Least fixed points and monotonicity

We can find a solution of the data-flow equations as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

We are interested in the least fixed point of F – that is the smallest according to $\sqsubseteq$.

- Does the algorithm above always terminate?
- Does it find a least fixed point?

Least fixed points and monotonicity

We can find a solution of the data-flow equations as follows:

1. start with the least element of the poset: $\vec{X}^0 = \{\} \times \cdots \times \{\}$
2. apply F to the current vector $\vec{X}^k$: $\vec{X}^{k+1} = F(\vec{X}^k) = F^{k+1}(\vec{X}^0)$
3. stop when $\vec{X}^{k+1} = \vec{X}^k$

We are interested in the least fixed point of F – that is the smallest according to $\sqsubseteq$.

- Does the algorithm above always terminate?
- Does it find a least fixed point?

F is a monotonic function: $\vec{X} \sqsubseteq \vec{Y}$ implies $F(\vec{X}) \sqsubseteq F(\vec{Y})$

Therefore, by induction on k: $F^k(\vec{X}^0) \sqsubseteq F^{k+1}(\vec{X}^0)$.

Since the poset $\wp(\{\mathrm{x}, \mathrm{y}, \mathrm{z}\})^{14}$ is finite, F must have a fixed point: it cannot keep on generating new values (finite domain), and it cannot "jump up and down" (monotonicity).

Finally, the fixed point computed from the least element $\vec{X}^0$ has to be the least fixed point (again thanks to monotonicity).

Naive fixed point algorithm

The algorithm that iterates F until it finds a fixed point is guaranteed to terminate but may be inefficient, as it propagates only a few updates in each iteration.

```
// naive fixed point algorithm
X⃗ := {} × ··· × {}
while F(X⃗) ≠ X⃗
    X⃗ := F(X⃗)
```

In our running example, the naive fixed point algorithm takes 10 iterations, which correspond to evaluating $140 = 10 \cdot 14$ equations.

Chaotic iteration

More efficient algorithms avoid recomputing all flow equations, while only propagating those that change. For example the chaotic iteration algorithm propagates one random component that changes in each iteration.

```
// chaotic iteration algorithm
X_1, ..., X_n := {}, ..., {}
while F_k(X_k) ≠ X_k for any k
    X_k := F_k(X_k)
```

In our running example, the chaotic algorithm takes 15–19 iterations, each evaluating one equation; the exact number depends on the random order in which elements are computed.

Worklist algorithm

A more efficient algorithm uses a worklist: a stack of edges in the CFG that should be processed.

- For each edge $\langle \texttt{from}, \texttt{to} \rangle$ in the worklist, compute the data-flow equation for $LV_{\text{IN}}(\texttt{to})$. If it is not a fixed point:
  - Update $LV_{\text{IN}}(\texttt{to}) = LV_{\text{OUT}}(\texttt{from})$ to the new value
  - Add all edges that lead to `from` to the top of the worklist, so that predecessors of `from` will be processed next

If an edge $\langle \ell, \ell' \rangle$ is in the worklist, it means that the result at block ℓ' has changed and must be propagated backward to its predecessors by computing the data-flow equation for block ℓ'.

Worklist algorithm

```
// worklist algorithm
```
$LV_{\mathrm{OUT}}(1), \ldots, LV_{\mathrm{OUT}}(n) := \{\}, \ldots, \{\}$
```
// the worklist initially includes all edges in the CFG
```
$\mathtt{W} := \mathit{edges}(\mathit{CFG})$

while $\mathtt{W.length} > 0$

  $\langle \mathtt{from},\ \mathtt{to} \rangle := \mathtt{W.pop()}$ `// remove top edge in worklist`

  if $\overbrace{(LV_{\mathrm{OUT}}(\mathtt{to}) \setminus \mathrm{kill}_{LV}(\mathtt{to})) \cup \mathrm{gen}_{LV}(\mathtt{to})}^{\text{update equation for } LV_{\mathrm{IN}}(\mathtt{to})} \not\subseteq LV_{\mathrm{OUT}}(\mathtt{from})$

    `// update OUT of 'from'`

    $LV_{\mathrm{OUT}}(\mathtt{from}) := LV_{\mathrm{OUT}}(\mathtt{from}) \cup (LV_{\mathrm{OUT}}(\mathtt{to}) \setminus \mathrm{kill}_{LV}(\mathtt{to})) \cup \mathrm{gen}_{LV}(\mathtt{to})$

    $LV_{\mathrm{IN}}(\mathtt{to}) := LV_{\mathrm{OUT}}(\mathtt{from})$ `// IN of successor`

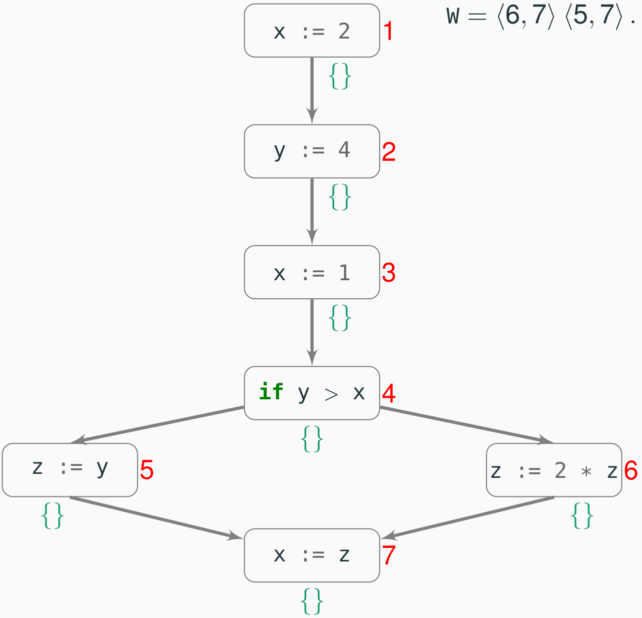
    `// add predecessors of 'from' to worklist`

    for $\langle \mathtt{before},\ \mathtt{from} \rangle \in \mathit{edges}(\mathit{CFG})$ $\mathtt{W.push}(\langle \mathtt{before},\ \mathtt{from} \rangle)$

```
// finally, update the IN of initial nodes
```
for $\mathtt{i} \in \mathit{initial}(\mathit{CFG})$ $LV_{\mathrm{IN}}(\mathtt{i}) := (LV_{\mathrm{OUT}}(\mathtt{i}) \setminus \mathrm{kill}_{LV}(\mathtt{i})) \cup \mathrm{gen}_{LV}(\mathtt{i})$

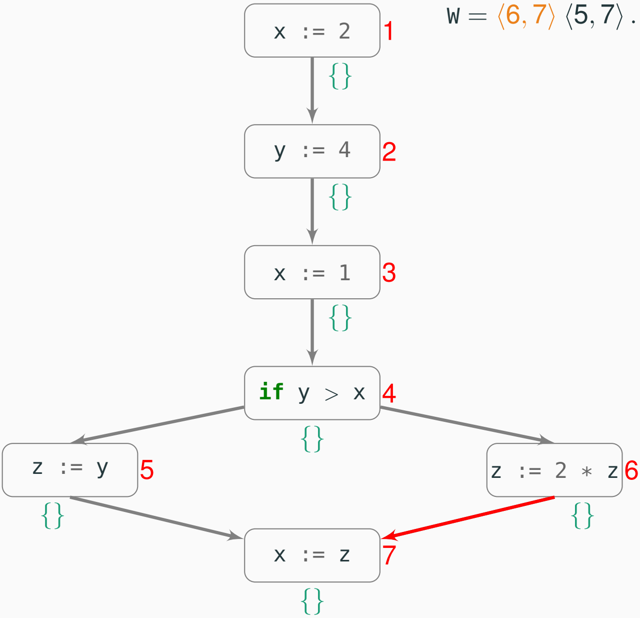
In our running example, the worklist algorithm takes 15 iterations, each evaluating one equation.

Worklist algorithm: example



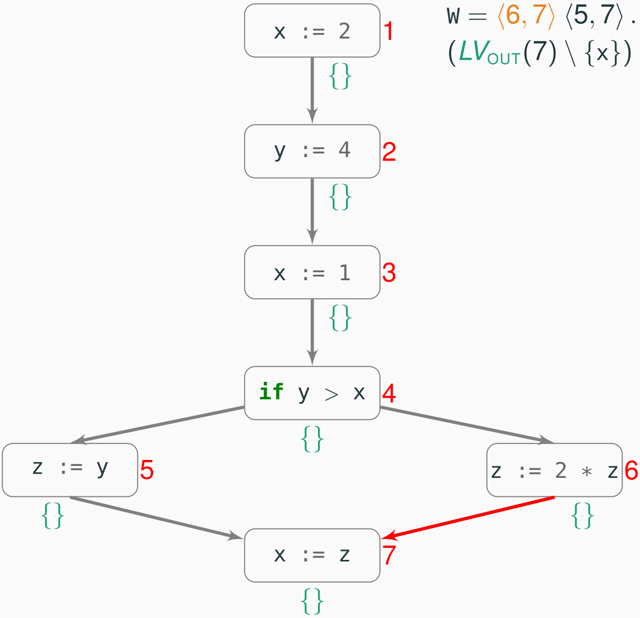
$\mathtt{W} = \langle 6, 7 \rangle \; \langle 5, 7 \rangle \ldots$

Worklist algorithm: example



$\mathtt{W} = \langle 6, 7 \rangle \ \langle 5, 7 \rangle \ldots$

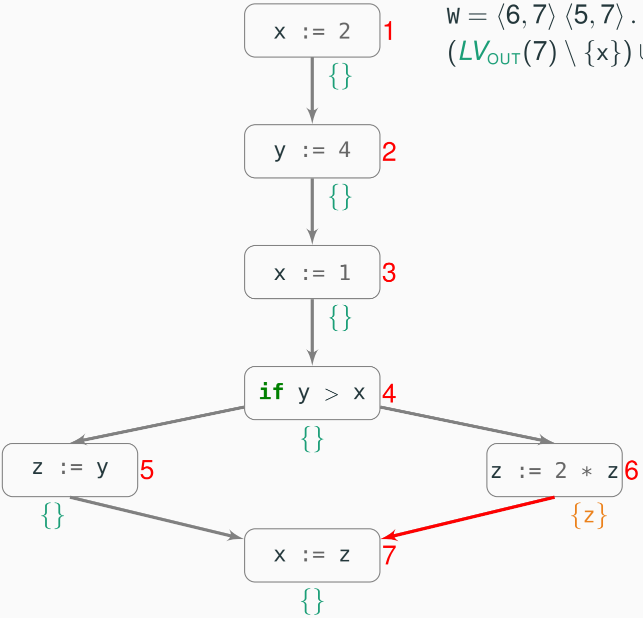
Worklist algorithm: example



$\mathtt{W} = \langle 6, 7 \rangle \; \langle 5, 7 \rangle \ldots$

$(LV_{\mathrm{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\mathrm{OUT}}(6)$

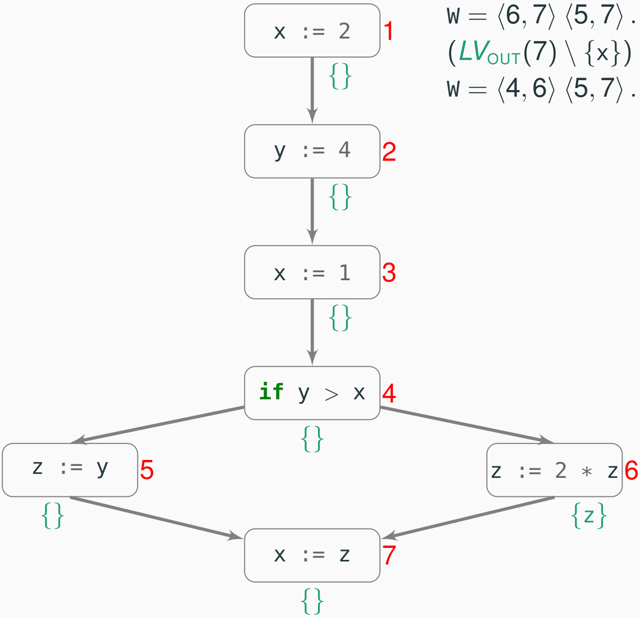
Worklist algorithm: example



$\mathtt{W} = \langle 6, 7 \rangle \, \langle 5, 7 \rangle \ldots$

$(LV_{\text{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(6)$

Worklist algorithm: example

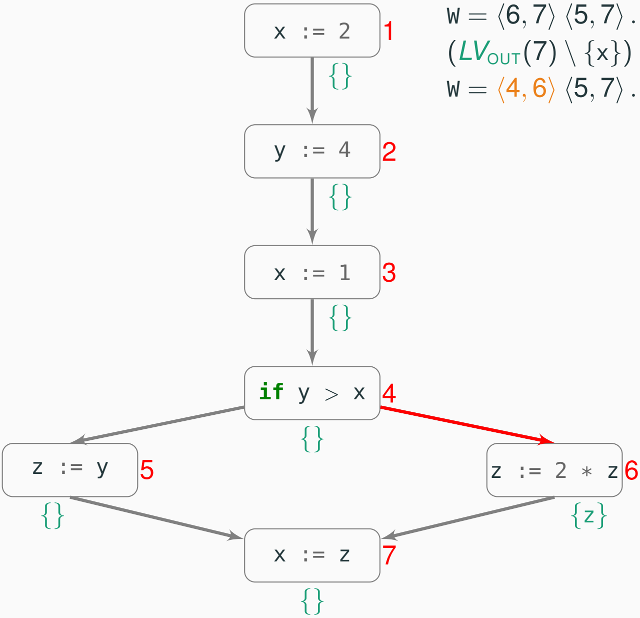


$W = \langle 6, 7\rangle\ \langle 5, 7\rangle \ldots$

$(LV_{\text{OUT}}(7) \setminus \{x\}) \cup \{z\} \not\subseteq LV_{\text{OUT}}(6)$

$W = \langle 4, 6\rangle\ \langle 5, 7\rangle \ldots$

Worklist algorithm: example

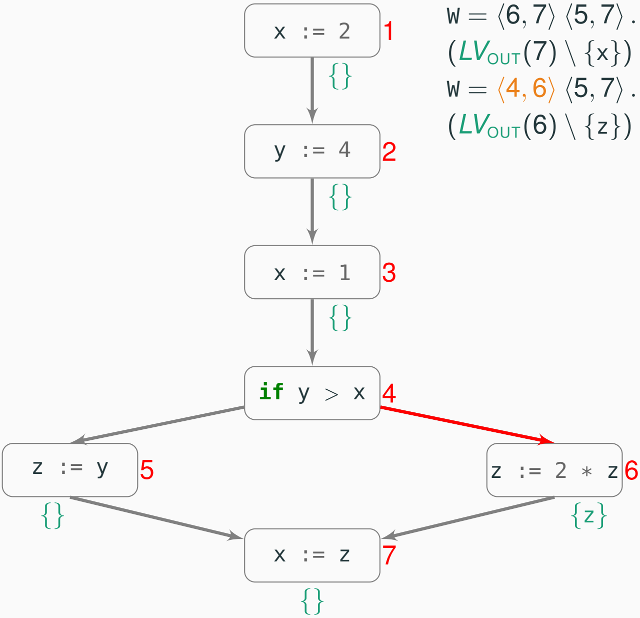


$\mathtt{W} = \langle 6,7\rangle\ \langle 5,7\rangle \ldots$

$(LV_{\text{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(6)$

$\mathtt{W} = \langle 4,6\rangle\ \langle 5,7\rangle \ldots$

Worklist algorithm: example



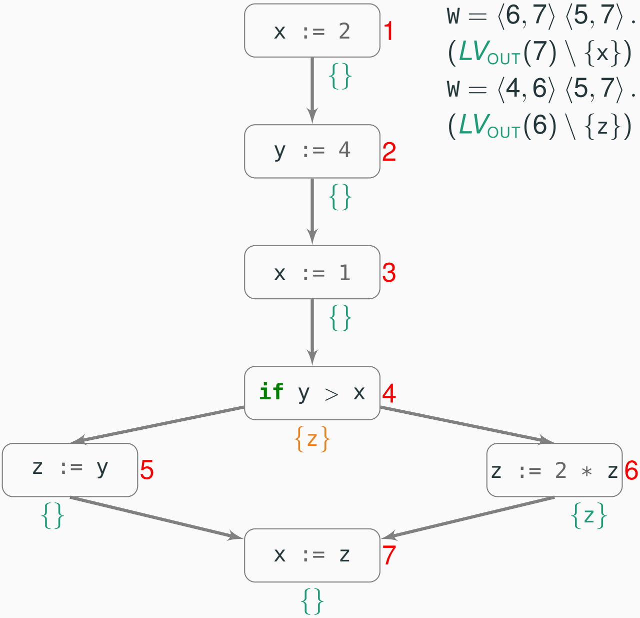
$\mathtt{W} = \langle 6, 7 \rangle \, \langle 5, 7 \rangle \ldots$

$(LV_{\text{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(6)$

$\mathtt{W} = \langle 4, 6 \rangle \, \langle 5, 7 \rangle \ldots$

$(LV_{\text{OUT}}(6) \setminus \{\mathtt{z}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(4)$

Worklist algorithm: example



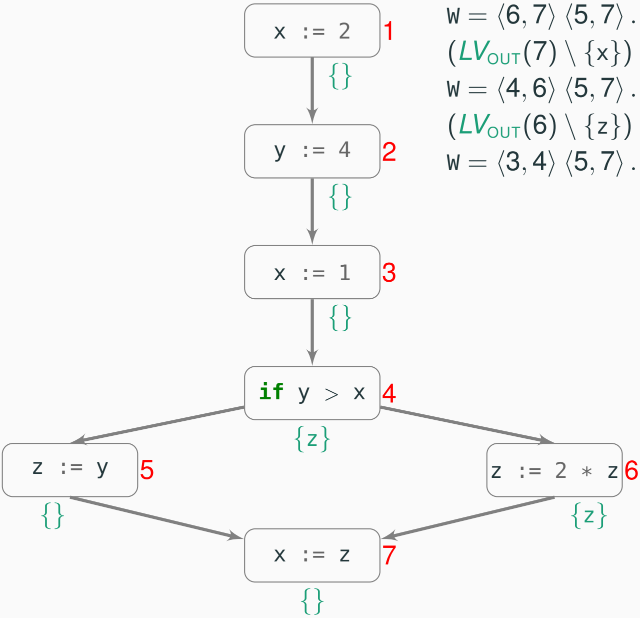
$\mathtt{W} = \langle 6, 7\rangle \, \langle 5, 7\rangle \ldots$

$(LV_{\mathrm{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\mathrm{OUT}}(6)$

$\mathtt{W} = \langle 4, 6\rangle \, \langle 5, 7\rangle \ldots$

$(LV_{\mathrm{OUT}}(6) \setminus \{\mathtt{z}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\mathrm{OUT}}(4)$

Worklist algorithm: example



$\mathtt{W} = \langle 6,7\rangle\ \langle 5,7\rangle \ldots$

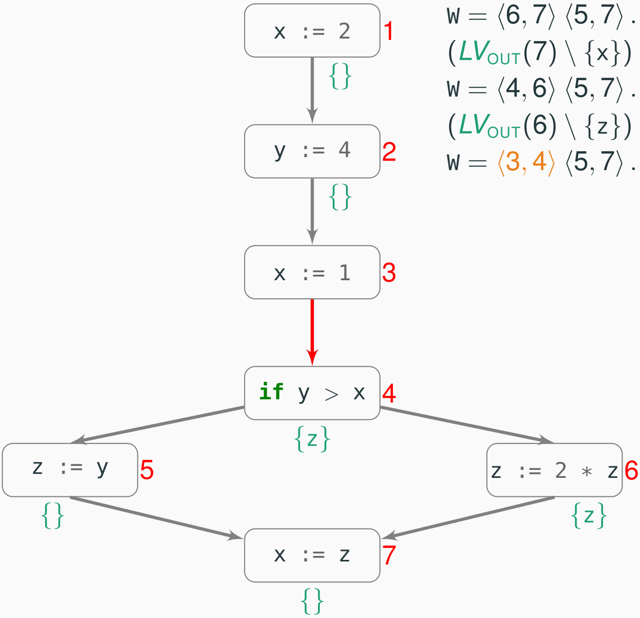
$(LV_{\mathrm{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\mathrm{OUT}}(6)$

$\mathtt{W} = \langle 4,6\rangle\ \langle 5,7\rangle \ldots$

$(LV_{\mathrm{OUT}}(6) \setminus \{\mathtt{z}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\mathrm{OUT}}(4)$

$\mathtt{W} = \langle 3,4\rangle\ \langle 5,7\rangle \ldots$

Worklist algorithm: example



$\mathtt{W} = \langle 6, 7\rangle\ \langle 5, 7\rangle \ldots$

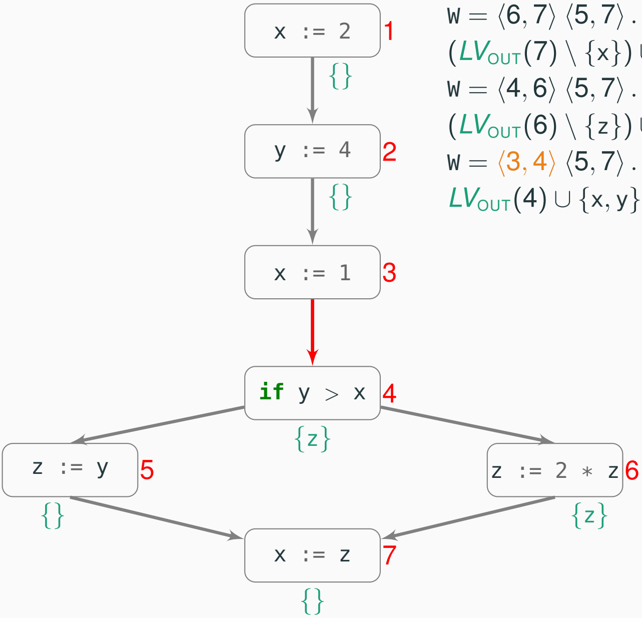
$(LV_{\text{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(6)$

$\mathtt{W} = \langle 4, 6\rangle\ \langle 5, 7\rangle \ldots$

$(LV_{\text{OUT}}(6) \setminus \{\mathtt{z}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(4)$

$\mathtt{W} = \langle 3, 4\rangle\ \langle 5, 7\rangle \ldots$

Worklist algorithm: example



$\mathtt{W} = \langle 6, 7 \rangle\ \langle 5, 7 \rangle \ldots$

$(LV_{\text{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(6)$

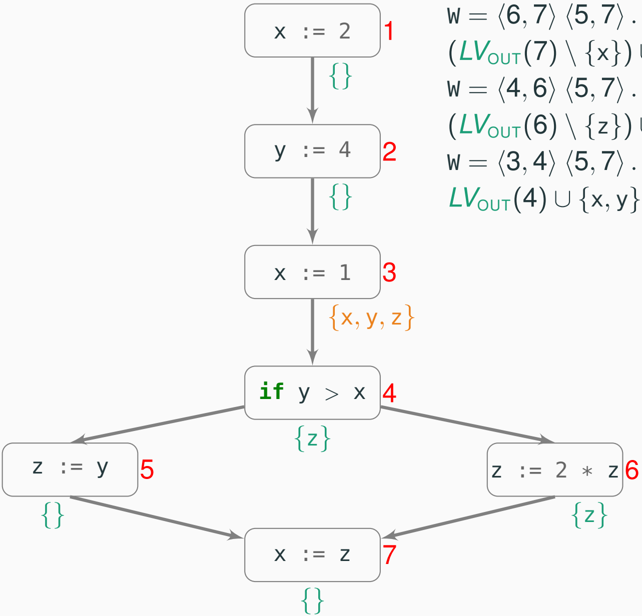
$\mathtt{W} = \langle 4, 6 \rangle\ \langle 5, 7 \rangle \ldots$

$(LV_{\text{OUT}}(6) \setminus \{\mathtt{z}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(4)$

$\mathtt{W} = \langle 3, 4 \rangle\ \langle 5, 7 \rangle \ldots$

$LV_{\text{OUT}}(4) \cup \{\mathtt{x}, \mathtt{y}\} \not\subseteq LV_{\text{OUT}}(3)$

Worklist algorithm: example



$\mathtt{W} = \langle 6, 7\rangle\ \langle 5, 7\rangle \ldots$

$(LV_{\text{OUT}}(7) \setminus \{\mathtt{x}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(6)$

$\mathtt{W} = \langle 4, 6\rangle\ \langle 5, 7\rangle \ldots$

$(LV_{\text{OUT}}(6) \setminus \{\mathtt{z}\}) \cup \{\mathtt{z}\} \not\subseteq LV_{\text{OUT}}(4)$

$\mathtt{W} = \langle 3, 4\rangle\ \langle 5, 7\rangle \ldots$

$LV_{\text{OUT}}(4) \cup \{\mathtt{x}, \mathtt{y}\} \not\subseteq LV_{\text{OUT}}(3)$

Data-flow analysis

Existence of solutions

Existence of solutions

Computing a data-flow analysis boils down to finding a least (smallest) fixed point of the vector equation:

$$\vec{X} = F(\vec{X})$$

- $\vec{X} = X_1, \ldots, X_n$ is a vector of variables, each over domain $D = \wp(\mathcal{V})$, where $\mathcal{V}$ is the set of program variables
- F is a vector function whose components are $F_1, \ldots, F_n$

Existence of solutions

Computing a data-flow analysis boils down to finding a least (smallest) fixed point of the vector equation:

$$\vec{X} = F(\vec{X})$$

- $\vec{X} = X_1, \ldots, X_n$ is a vector of variables, each over domain $D = \wp(\mathcal{V})$, where $\mathcal{V}$ is the set of program variables
- F is a vector function whose components are $F_1, \ldots, F_n$

What properties of F and D guarantee that the data-flow equations have a least fixed point?

Complete lattices

A complete lattice is a poset $\langle D, \sqsubseteq \rangle$ such that every subset $S \sqsubseteq D$ of D has:

- a least upper bound (also: lub, join, or supremum) $\sqcup S$
- a greatest lower bound (also: glb, meet, or infimum) $\sqcap S$

Complete lattices

A complete lattice is a poset $\langle D, \sqsubseteq \rangle$ such that
every subset $S \sqsubseteq D$ of D has:

- a least upper bound (also: lub, join, or supremum) $\sqcup S$
- a greatest lower bound (also: glb, meet, or infimum) $\sqcap S$

- Element $u \in D$ is an upper bound of set $S \subseteq D$ if $s \sqsubseteq u$ for all $s \in S$
- The least upper bound $\sqcup S$ of a set $S \subseteq D$ is the smallest of its upper bounds

- Element $d \in D$ is an lower bound of set $S \subseteq D$ if $d \sqsubseteq s$ for all $s \in S$
- The greatest upper bound $\sqcap S$ of a set $S \subseteq D$ is the largest of its bounds

Complete lattices

A complete lattice is a poset $\langle D, \sqsubseteq \rangle$ such that
every subset $S \sqsubseteq D$ of D has:

- a least upper bound (also: lub, join, or supremum) $\sqcup S$
- a greatest lower bound (also: glb, meet, or infimum) $\sqcap S$

- Element $u \in D$ is an upper bound of set $S \subseteq D$ if $s \sqsubseteq u$ for all $s \in S$
- The least upper bound $\sqcup S$ of a set $S \subseteq D$ is the smallest of its upper bounds

- Element $d \in D$ is an lower bound of set $S \subseteq D$ if $d \sqsubseteq s$ for all $s \in S$
- The greatest upper bound $\sqcap S$ of a set $S \subseteq D$ is the largest of its bounds

Every complete lattice is not empty, and has
a least element $\bot$ (bottom) and a greatest element $\top$ (top).

Complete lattice: example

The powerset ordered with respect to the subset $\subseteq$ relation is a complete lattice.

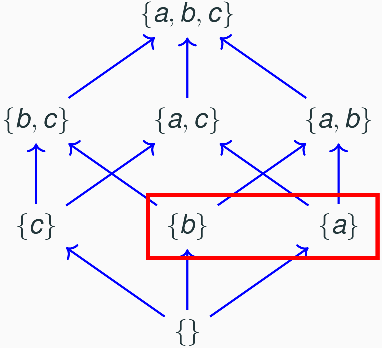
For example $\wp(\{a, b, c\})$:

$$\begin{tikzcd}
 & \{a, b, c\} & \\
\{b, c\} \arrow[ur] & \{a, c\} \arrow[u] & \{a, b\} \arrow[ul] \\
\{c\} \arrow[u] \arrow[ur] & \{b\} \arrow[ul] \arrow[ur] & \{a\} \arrow[ul] \arrow[u] \\
 & \{\} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Complete lattice: example

The powerset ordered with respect to the subset $\subseteq$ relation is a complete lattice.

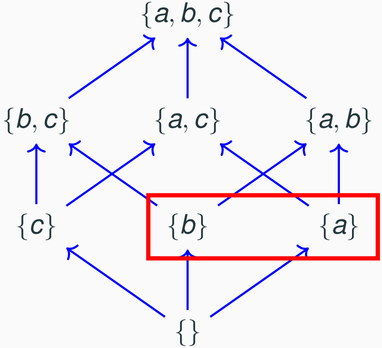
For example $\wp(\{a, b, c\})$:



Complete lattice: example

The powerset ordered with respect to the subset $\subseteq$ relation is a complete lattice.

For example $\wp(\{a, b, c\})$:



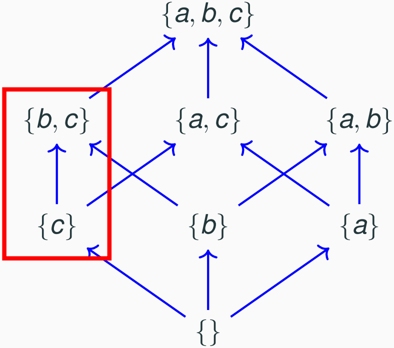
The upper bounds of $S = \{\{a\}, \{b\}\} \subseteq D = \wp(\{a, b, c\})$ are $\{a, b\}$ and $\{a, b, c\}$.

The only lower bound of $S = \{\{a\}, \{b\}\} \subseteq D = \wp(\{a, b, c\})$ is $\{\}$.

Complete lattice: example

The powerset ordered with respect to the subset $\subseteq$ relation is a complete lattice.

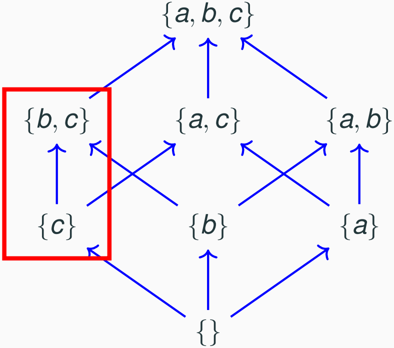
For example $\wp(\{a, b, c\})$:



Complete lattice: example

The powerset ordered with respect to the subset $\subseteq$ relation is a complete lattice.

For example $\wp(\{a, b, c\})$:



The only upper bound of $S = \{\{c\}, \{b, c\}\} \subseteq D = \wp(\{a, b, c\})$ is $\{a, b, c\}$.

The lower bounds of $S = \{\{c\}, \{b, c\}\} \subseteq D = \wp(\{a, b, c\})$ are $\{c\}$ and $\{\}$.

Complete lattice: example

The powerset ordered with respect to the subset $\subseteq$ relation is a complete lattice.

For example $\wp(\{a, b, c\})$:

$$
\begin{tikzcd}
 & \{a,b,c\} & \\
\{b,c\} \arrow[ur] & \{a,c\} \arrow[u] & \{a,b\} \arrow[ul] \\
\{c\} \arrow[u] \arrow[ur] & \{b\} \arrow[ul] \arrow[ur] & \{a\} \arrow[u] \arrow[ul] \\
 & \{\} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}
$$

The bottom (least element) is $\{\}$.

The top (greatest element) is $\{a, b, c\}$.

Tarski's fixed point theorem

A function $F\colon D \to D$ is monotonic over poset $\langle D, \sqsubseteq \rangle$ if,
for all $x, y \in D$, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$

Intuitively: monotonic means that it respects the order relation.

Tarski's fixed point theorem

A function $F\colon D \to D$ is monotonic over poset $\langle D, \sqsubseteq \rangle$ if,
for all $x, y \in D$, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$

Intuitively: monotonic means that it respects the order relation.

A value $d \in D$ is a fixed point of a function $F\colon D \to D$ if $F(d) = d$.

Tarski's fixed point theorem

A function $F\colon D \to D$ is monotonic over poset $\langle D, \sqsubseteq \rangle$ if,
for all $x, y \in D$, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$

Intuitively: monotonic means that it respects the order relation.

A value $d \in D$ is a fixed point of a function $F\colon D \to D$ if $F(d) = d$.

Tarski's fixed point theorem: let $F\colon D \to D$ be a <u>monotonic</u> function over <u>complete lattice</u> $\langle D, \sqsubseteq \rangle$. The set of all fixed points of F is also a complete lattice with respect to $\sqsubseteq$.

Tarski's fixed point theorem

A function $F\colon D \to D$ is monotonic over poset $\langle D, \sqsubseteq \rangle$ if,
for all $x, y \in D$, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$

Intuitively: monotonic means that it respects the order relation.

A value $d \in D$ is a fixed point of a function $F\colon D \to D$ if $F(d) = d$.

Tarski's fixed point theorem: let $F\colon D \to D$ be a monotonic function over complete lattice $\langle D, \sqsubseteq \rangle$. The set of all fixed points of F is also a complete lattice with respect to $\sqsubseteq$.

Implications:

- F has at least one fixed point
  (because complete lattices cannot be empty)
- F has least and greatest fixed points
  (because its fixed points are a complete lattice)

Tarski's fixed point theorem

A function $F\colon D \to D$ is monotonic over poset $\langle D, \sqsubseteq\rangle$ if,
for all $x, y \in D$, $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$

Intuitively: monotonic means that it respects the order relation.

A value $d \in D$ is a fixed point of a function $F\colon D \to D$ if $F(d) = d$.

Tarski's fixed point theorem: let $F\colon D \to D$ be a monotonic function over complete lattice $\langle D, \sqsubseteq\rangle$. The set of all fixed points of F is also a complete lattice with respect to $\sqsubseteq$.

It is Tarski who stated the result in its most general form, [but] some time earlier, Knaster and Tarski established the result for [a] special case.

Knaster-Tarski theorem on Wikipedia

Alfred Tarski

Applying Tarski's fixed point theorem

To apply Tarski's theorem to a data-flow analysis – guaranteeing the existence of a fixed point, which can then be found by iteration – we need to show:

monotonicity: the data-flow vector equation F is monotonic

complete lattice: the analysis domain D is a complete lattice

Applying Tarski's fixed point theorem

To prove that F is monotonic, we just prove that each component function F_k is monotonic.

Equations of this form:

$$LV_{\text{OUT}}(k) = \bigcup_{(k \to h) \in \text{CFG}} LV_{\text{IN}}(h)$$

$$LV_{\text{IN}}(k) = \left(LV_{\text{OUT}}(k) \setminus \text{kill}_{LV}(k)\right) \cup \text{gen}_{LV}(k)$$

are monotonic because $\setminus$ and $\cup$ are themselves monotonic.

Applying Tarski's fixed point theorem

To prove that F is monotonic, we just prove that each component function F_k is monotonic.

Equations of this form:

$$LV_{\text{OUT}}(k) = \bigcup_{(k \to h) \in \text{CFG}} LV_{\text{IN}}(h)$$

$$LV_{\text{IN}}(k) = \big(LV_{\text{OUT}}(k) \setminus \text{kill}_{LV}(k)\big) \cup \text{gen}_{LV}(k)$$

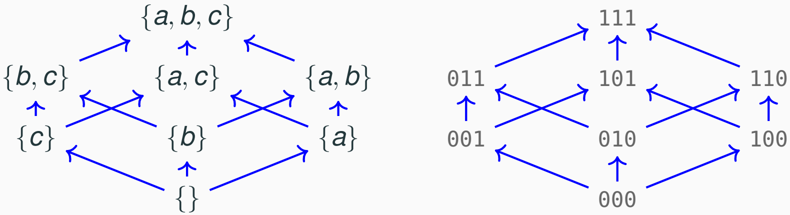
are monotonic because $\setminus$ and $\cup$ are themselves monotonic.

Whenever each variable's domain D_k is a powerset – typically the powerset $\wp(\mathcal{V})$ of the program variables – it is a complete lattice with respect to the subset relation $\subseteq$.
Then, the overall domain $D = D_1 \times \cdots \times D_n$ is also a complete lattice with respect to $\sqsubseteq$ defined as $X \sqsubseteq Y$ iff $\forall k (X_k \subseteq Y_k)$.

Bit vectors

Elements of powerset $\wp(S)$ of a finite set S can be efficiently represented using bit vectors:

- the length of the bit vector is $|S| = n$
- an element $s \in \wp(S)$ is uniquely represented by the bit string $b_1, \ldots, b_n$ where $b_k = 1$ iff the kth element of S belongs to s



Join and meet operations are then bitwise logic operations:

$$\sqcup\{\{a\},\{b\}\} = \{a, b\} \qquad \texttt{100} \vee \texttt{010} = \texttt{110}$$

$$\sqcap\{\{a\},\{b\}\} = \{\} \qquad \texttt{100} \wedge \texttt{010} = \texttt{000}$$

Data-flow analysis

Reaching definitions analysis

Reaching definitions

A definition (v, k) is an assignment to variable v at block k.

A definition (v, k) reaches block r if there is some path (on the CFG) from k to r that does not redefine v

Reaching definitions

A definition (v, k) is an assignment to variable v at block k.

A definition (v, k) reaches block r if there is some path (on the CFG) from k to r that does not redefine v

```
{ x := 5 }1
{ y := 1 }2
while ( x > 1 )3
   { y := x * y }4
   { x := x - 1 }5
```

Examples: which definitions reach (the entry of) block 5?

Reaching definitions

A definition (v, k) is an assignment to variable v at block k.

A definition (v, k) reaches block r if there is some path (on the CFG) from k to r that does not redefine v

```
{ x := 5 }¹
{ y := 1 }²
while ( x > 1 )³
  { y := x * y }⁴
  { x := x - 1 }⁵
```

Examples: which definitions reach (the entry of) block 5?

- in the first loop iteration: $(\mathsf{x}, 1)$ and $(\mathsf{y}, 4)$

Reaching definitions

A definition (v, k) is an assignment to variable v at block k.

A definition (v, k) reaches block r if there is some path (on the CFG) from k to r that does not redefine v

```
{ x := 5 }1
{ y := 1 }2
while ( x > 1 )3
   { y := x * y }4
   { x := x - 1 }5
```

Examples: which definitions reach (the entry of) block 5?

- in the first loop iteration: $(\mathsf{x}, 1)$ and $(\mathsf{y}, 4)$
- in the following iterations: $(\mathsf{x}, 5)$ and $(\mathsf{y}, 4)$

Reaching definitions analysis

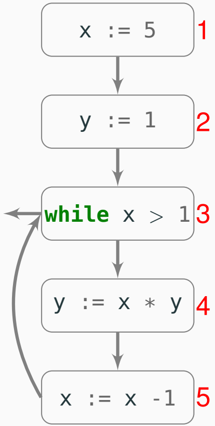
A definition $(\mathtt{v}, k)$ reaches block r if there is some path (on the CFG) from k to r that does not redefine $\mathtt{v}$

Reaching definitions analysis: for each program point, determine which definitions may reach the point.

Reaching definitions analysis

A definition (v, k) reaches block r if there is some path (on the CFG) from k to r that does not redefine v

Reaching definitions analysis: for each program point, determine which definitions may reach the point.



Reaching definitions analysis's output:

$$\vdots$$

$$RD_{\mathrm{IN}}(5) = RD_{\mathrm{OUT}}(4) = \{(\mathsf{x}, 1), (\mathsf{x}, 5), (\mathsf{y}, 4)\}$$

Reaching definitions analysis

A definition (v, k) reaches block r if there is some path (on the CFG) from k to r that does not redefine v

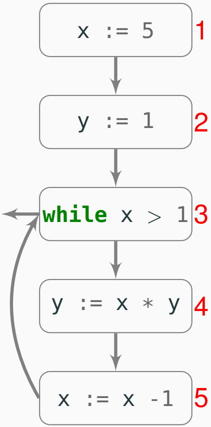
Reaching definitions analysis: for each program point, determine which definitions may reach the point.

A may analysis is an over-approximation: $RD(k)$ is a superset of the reaching definitions at k.

- if $(\mathsf{x}, \ell) \in RD_{\mathrm{IN}}(k)$, the definition of x at ℓ may or may not reach k (for example because it may be overwritten along certain paths but not along others)
- if $(\mathsf{x}, \ell) \notin RD_{\mathrm{IN}}(k)$, the definition of x at ℓ has definitely <u>not reached</u> k

Reaching definitions analysis: idea and example

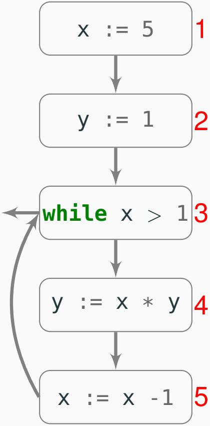
Working forward, record the reaching definitions at the entry and exit of every elementary block.



Reaching definitions analysis: idea and example

Working forward, record the reaching definitions at the entry and exit of every elementary block.

$RD_{\text{IN}}(1) = \{(x, ?), (y, ?)\}$

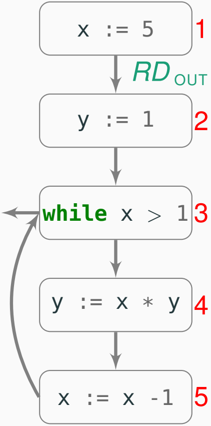


Reaching definitions analysis: idea and example

Working forward, record the reaching definitions at the entry and exit of every elementary block.

implicit initialization

$RD_{\text{IN}}(1) = \{(x, ?), (y, ?)\}$



$RD_{\text{OUT}}(1) = \{(x, 1), (y, ?)\}$

Reaching definitions analysis: idea and example

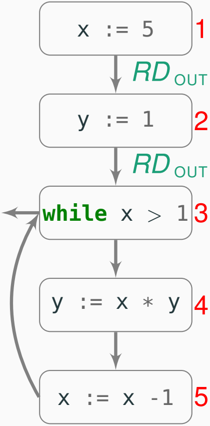
Working forward, record the reaching definitions at the entry and exit of every elementary block.

implicit initialization

$RD_{\text{IN}}(1) = \{(x, ?), (y, ?)\}$

$RD_{\text{OUT}}(1) = \{(x, 1), (y, ?)\}$

$RD_{\text{OUT}}(2) = \{(x, 1), (y, 2)\}$

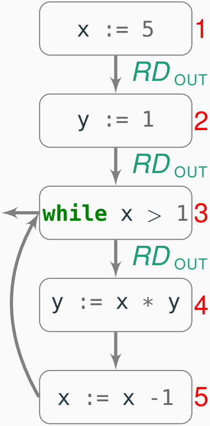


Reaching definitions analysis: idea and example

Working forward, record the reaching definitions at the entry and exit of every elementary block.

implicit initialization

$RD_{\text{IN}}(1) = \{(x, ?), (y, ?)\}$



$RD_{\text{OUT}}(1) = \{(x, 1), (y, ?)\}$

$RD_{\text{OUT}}(2) = \{(x, 1), (y, 2)\}$

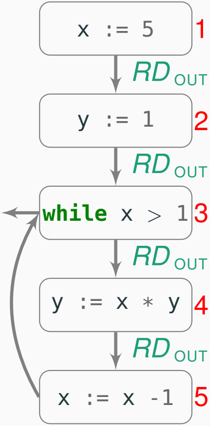
$RD_{\text{OUT}}(3) = \{(x, 1), (y, 2), (y, 4), (x, 5)\}$

Reaching definitions analysis: idea and example

Working forward, record the reaching definitions at the entry and exit of every elementary block.

implicit initialization

$$RD_{\text{IN}}(1) = \{(x, ?), (y, ?)\}$$



$$RD_{\text{OUT}}(1) = \{(x, 1), (y, ?)\}$$

$$RD_{\text{OUT}}(2) = \{(x, 1), (y, 2)\}$$

$$RD_{\text{OUT}}(3) = \{(x, 1), (y, 2), (y, 4), (x, 5)\}$$

$$RD_{\text{OUT}}(4) = \{(x, 1), (y, 4), (x, 5)\}$$

Reaching definitions analysis: idea and example

Working forward, record the reaching definitions at the entry and exit of every elementary block.

implicit initialization

$RD_{\text{IN}}(1) = \{(x, ?), (y, ?)\}$

```
x := 5
```
1

$RD_{\text{OUT}}(1) = \{(x, 1), (y, ?)\}$

```
y := 1
```
2

$RD_{\text{OUT}}(2) = \{(x, 1), (y, 2)\}$

```
while x > 1
```
3

$RD_{\text{OUT}}(3) = \{(x, 1), (y, 2), (y, 4), (x, 5)\}$

```
y := x * y
```
4

$RD_{\text{OUT}}(4) = \{(x, 1), (y, 4), (x, 5)\}$

```
x := x -1
```
5

$RD_{\text{OUT}}(5) = \{(y, 4), (x, 5)\}$

Data-flow equations

We formalize the reaching definitions analysis similarly to the live variables analysis but working forward.

For every block k:

$$RD_{\text{IN}}(k) = \bigcup_{(h \to k) \in \text{CFG}} RD_{\text{OUT}}(h)$$

$$RD_{\text{OUT}}(k) = \left(RD_{\text{IN}}(k) \setminus \text{kill}_{RD}(k)\right) \cup \text{gen}_{RD}(k)$$

Data-flow equations

We formalize the reaching definitions analysis similarly to the live variables analysis but working forward.

For every block k:

for every node h that precedes k in the CFG
(i.e., h is a direct predecessor of k)

$$RD_{\text{IN}}(k) = \bigcup_{(h \to k) \in \text{CFG}} RD_{\text{OUT}}(h)$$

$$RD_{\text{OUT}}(k) = \big(RD_{\text{IN}}(k) \setminus \text{kill}_{RD}(k)\big) \cup \text{gen}_{RD}(k)$$

other definitions of the same variables redefined at k

variables defined at k

Data-flow equations

We formalize the reaching definitions analysis similarly to the live variables analysis but working forward.

For every block k:

$$RD_{\text{IN}}(k) = \bigcup_{(h \to k) \in \text{CFG}} RD_{\text{OUT}}(h)$$

$$RD_{\text{OUT}}(k) = \big(RD_{\text{IN}}(k) \setminus \text{kill}_{RD}(k)\big) \cup \text{gen}_{RD}(k)$$

If i is an initial node, we have to set

$$RD_{\text{IN}}(i) = \{(\text{v}, ?) \mid \text{v is a program variable}\}$$

Data-flow equations

We formalize the reaching definitions analysis similarly to the live variables analysis but working forward.

For every block k:

$$RD_{\text{IN}}(k) = \bigcup_{(h \to k) \in \text{CFG}} RD_{\text{OUT}}(h)$$

$$RD_{\text{OUT}}(k) = \big(RD_{\text{IN}}(k) \setminus \text{kill}_{RD}(k)\big) \cup \text{gen}_{RD}(k)$$

If i is an initial node, we have to set

$$RD_{\text{IN}}(i) = \{(\text{v}, ?) \mid \text{v is a program variable}\}$$

We define kill_{RD} and gen_{RD} for every block type:

$$\text{kill}_{RD}(\texttt{skip}) = \{\} \qquad \text{gen}_{RD}(\texttt{skip}) = \{\}$$

$$\text{kill}_{RD}(\text{v} := E) = \{(\text{v}, p) \mid \text{for all } \underset{\substack{\uparrow \\ \text{program points or ?}}}{p}\} \qquad \text{gen}_{RD}(\{\text{v} := E\}^{k}) = \{(\text{v}, k)\}$$

$$\text{kill}_{RD}(\texttt{if}/\texttt{while}\ C) = \{\} \qquad \text{gen}_{RD}(\texttt{if}/\texttt{while}\ C) = \{\}$$

Use-Definition and Definition-Use chains

The information about which statements produce values and which use them is useful for many program optimizations. A reaching definitions analysis has this information, which can be displayed directly as links between statements.

```
{ x := 0 }1
{ x := 3 }2
if (z = x)3
  { z := 0 }4
else
  { z := x }5
{ y := x }6
{ x := y + z }7
```

Use-Definition and Definition-Use chains

The information about which statements produce values and which use them is useful for many program optimizations. A reaching definitions analysis has this information, which can be displayed directly as links between statements.

```
{ x := 0 }1
{ x := 3 }2
if (z = x)3
  { z := 0 }4
else
  { z := x }5
{ y := x }6
{ x := y + z }7
```

Use-definition chains (UD chains): link from each use of a variable to all assignments that may reach it.

Example: UD chain for x at point 6.

Use-Definition and Definition-Use chains

The information about which statements produce values and which use them is useful for many program optimizations. A reaching definitions analysis has this information, which can be displayed directly as links between statements.

```
{ x := 0 }[1]
{ x := 3 }[2]
if (z = x)[3]
  { z := 0 }[4]
else
  { z := x }[5]
{ y := x }[6]
{ x := y + z }[7]
```

Use-definition chains (UD chains): link from each use of a variable to all assignments that may reach it.

Example: UD chain for x at point 6.

Use-Definition and Definition-Use chains

The information about which statements produce values and which use them is useful for many program optimizations. A reaching definitions analysis has this information, which can be displayed directly as links between statements.

```
{ x := 0 }1
{ x := 3 }2
if (z = x)3
  { z := 0 }4
else
  { z := x }5
{ y := x }6
{ x := y + z }7
```

Use-definition chains (UD chains): link from each use of a variable to all assignments that may reach it.

Example: UD chain for x at point 6.

Definition-use chains (DU chains): link from each assignment of a variable to all expressions that may use it.

Example: DU chain for x at point 2.

Use-Definition and Definition-Use chains

The information about which statements produce values and which use them is useful for many program optimizations. A reaching definitions analysis has this information, which can be displayed directly as links between statements.

```
{ x := 0 }1
{ x := 3 }2
if (z = x)3
  { z := 0 }4
else
  { z := x }5
{ y := x }6
{ x := y + z }7
```

Use-definition chains (UD chains): link from each use of a variable to all assignments that may reach it.

Example: UD chain for x at point 6.

Definition-use chains (DU chains): link from each assignment of a variable to all expressions that may use it.

Example: DU chain for x at point 2.

UD chains: definition

location of implicit initialization

Use-definition chains (UD chains): link from each use of a variable to all assignments that may reach it.

$$\mathrm{UD}(\mathtt{v}, k) \;=\; \{q \mid \{\mathtt{v} := E\}^{q} \text{ and } \mathit{clear}(\mathtt{v}, q, k)\} \cup \{? \mid \mathit{clear}(\mathtt{v}, ?, k)\}$$

UD chains: all $p \to q$ such that p uses some $\mathtt{x}$ and $q \in \mathrm{UD}(\mathtt{x}, p)$

Predicate *clear*$(\mathtt{x}, p, q)$ holds iff there is a definition-clear path from p to q: a path such that no block strictly between p and q redefines $\mathtt{x}$.

UD chains: definition

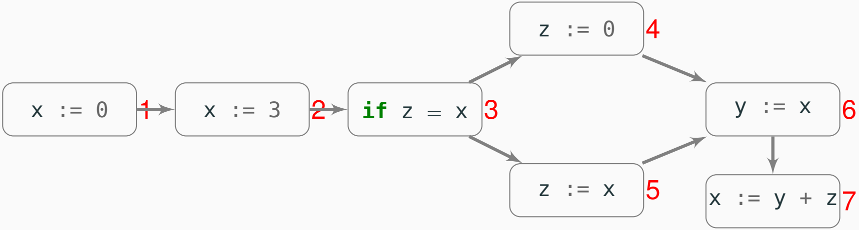
location of implicit initialization

Use-definition chains (UD chains): link from each use of a variable <u>to all assignments</u> that may reach it.

$$\mathrm{UD}(\mathtt{v}, k) \;=\; \{q \mid \{\mathtt{v} := E\}^{q} \text{ and } \mathit{clear}(\mathtt{v}, q, k)\} \cup \{? \mid \mathit{clear}(\mathtt{v}, ?, k)\}$$

UD chains: all $p \rightarrow q$ such that p uses some $\mathtt{x}$ and $q \in \mathrm{UD}(\mathtt{x}, p)$

Predicate $\mathit{clear}(\mathtt{x}, p, q)$ holds iff there is a definition-clear path from p to q: a path such that no block strictly between p and q redefines $\mathtt{x}$.



UD chains: definition

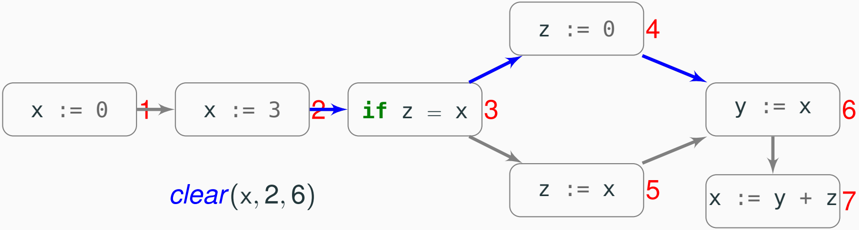
location of implicit initialization

Use-definition chains (UD chains): link from each use of a variable to all assignments that may reach it.

$$\mathrm{UD}(\mathtt{v}, k) \;=\; \{q \mid \{\mathtt{v} := E\}^{q} \text{ and } \mathit{clear}(\mathtt{v}, q, k)\} \cup \{? \mid \mathit{clear}(\mathtt{v}, ?, k)\}$$

UD chains: all $p \rightarrow q$ such that p uses some $\mathtt{x}$ and $q \in \mathrm{UD}(\mathtt{x}, p)$

Predicate $\mathit{clear}(\mathtt{x}, p, q)$ holds iff there is a definition-clear path from p to q: a path such that no block strictly between p and q redefines $\mathtt{x}$.



$\mathit{clear}(\mathtt{x}, 2, 6)$

UD chains from reaching definitions

Use-definition chains (UD chains): link from each use of a variable to all assignments that may reach it.

$$\mathrm{UD}(\mathtt{v}, k) \;=\; \{q \mid \{\mathtt{v} := E\}^{\mathtt{q}} \text{ and } \mathit{clear}(\mathtt{v}, q, k)\} \cup \{? \mid \mathit{clear}(\mathtt{v}, ?, k)\}$$

UD chains: all $p \rightarrow q$ such that p uses some $\mathtt{x}$ and $q \in \mathrm{UD}(\mathtt{x}, p)$

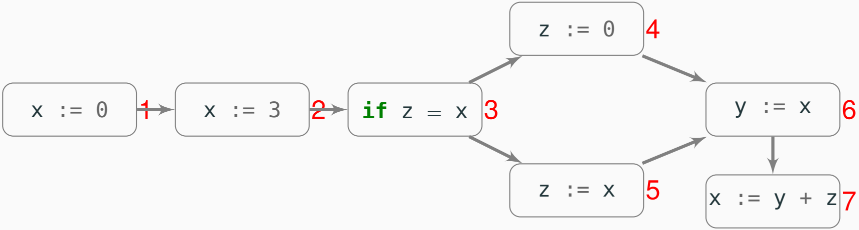
Set $\mathrm{UD}(\mathtt{v}, k)$ can be computed from the information collected by a reaching definitions analysis:

$$\mathrm{UD}(\mathtt{v}, k) \;=\; \begin{cases} \{q \mid (\mathtt{v}, q) \in \mathit{RD}_{\mathrm{IN}}(k)\} & \text{if } \mathtt{v} \text{ is used in block } k \\ \{\,\} & \text{otherwise} \end{cases}$$

UD chains from reaching definitions

Set UD(v, k) can be computed from the information collected by a reaching definitions analysis:

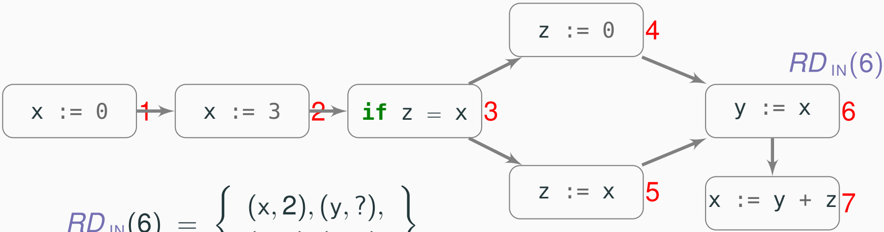
$$\mathrm{UD}(\mathrm{v}, k) = \begin{cases} \{q \mid (\mathrm{v}, q) \in RD_{\mathrm{IN}}(k)\} & \text{if v is used in block } k \\ \{\,\} & \text{otherwise} \end{cases}$$



UD chains from reaching definitions

Set UD(v, k) can be computed from the information collected by a reaching definitions analysis:

$$\mathrm{UD}(\mathtt{v}, k) = \begin{cases} \{q \mid (\mathtt{v}, q) \in RD_{\mathrm{IN}}(k)\} & \text{if } \mathtt{v} \text{ is used in block } k \\ \{\,\} & \text{otherwise} \end{cases}$$

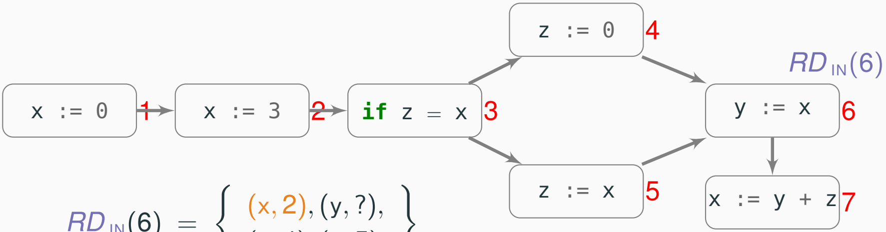


$$RD_{\mathrm{IN}}(6) = \left\{ \begin{array}{l} (\mathtt{x}, 2), (\mathtt{y}, ?), \\ (\mathtt{z}, 4), (\mathtt{z}, 5) \end{array} \right\}$$

UD chains from reaching definitions

Set UD(v, k) can be computed from the information collected by a reaching definitions analysis:

$$\mathrm{UD}(\mathtt{v}, k) = \begin{cases} \{q \mid (\mathtt{v}, q) \in RD_{\mathrm{IN}}(k)\} & \text{if } \mathtt{v} \text{ is used in block } k \\ \{\,\} & \text{otherwise} \end{cases}$$



$$RD_{\mathrm{IN}}(6) = \left\{ \begin{array}{l} (\mathtt{x}, 2), (\mathtt{y}, ?), \\ (\mathtt{z}, 4), (\mathtt{z}, 5) \end{array} \right\}$$

DU chains from reaching definitions

Definition-use chains (DU chains): link from each assignment of a variable to all expressions that may use it.

$$\mathrm{DU}(\mathsf{v}, k) \;=\; \{q \mid q \text{ uses } \mathsf{v} \text{ and } (\mathsf{v}, k) \text{ reaches } q\}$$

DU chains: all $p \to q$ such that $q \in \mathrm{DU}(\mathsf{x}, p)$

DU chains from reaching definitions

Definition-use chains (DU chains): link from each assignment of a variable to all expressions that may use it.

$$\mathrm{DU}(\mathtt{v}, k) \;=\; \{q \mid q \text{ uses } \mathtt{v} \text{ and } (\mathtt{v}, k) \text{ reaches } q\}$$

DU chains: all $p \to q$ such that $q \in \mathrm{DU}(\mathtt{x}, p)$

Set $\mathrm{DU}(\mathtt{v}, k)$ can be computed as the “inverse” of UD:

$$\mathrm{DU}(\mathtt{v}, k) \;=\; \{q \mid k \in \mathrm{UD}(\mathtt{v}, q)\}$$

DU chains from reaching definitions

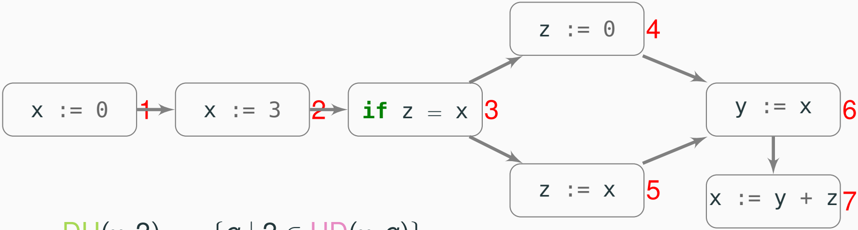
Definition-use chains (DU chains): link from each assignment of a variable to all expressions that may use it.

$$\mathrm{DU}(\mathsf{v}, k) \;=\; \{q \mid q \text{ uses } \mathsf{v} \text{ and } (\mathsf{v}, k) \text{ reaches } q\}$$

DU chains: all $p \to q$ such that $q \in \mathrm{DU}(\mathsf{x}, p)$

Set $\mathrm{DU}(\mathsf{v}, k)$ can be computed as the "inverse" of UD:

$$\mathrm{DU}(\mathsf{v}, k) \;=\; \{q \mid k \in \mathrm{UD}(\mathsf{v}, q)\}$$



$$\mathrm{DU}(\mathsf{x}, 2) \;=\; \{q \mid 2 \in \mathrm{UD}(\mathsf{x}, q)\}$$

DU chains from reaching definitions

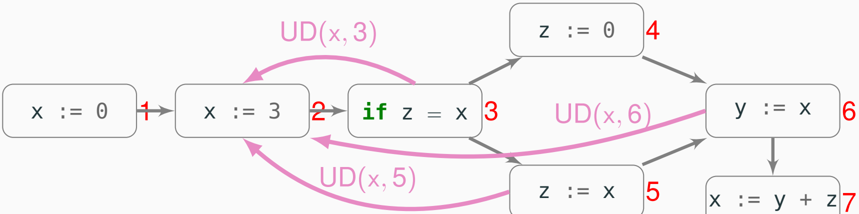
Definition-use chains (DU chains): link from each assignment of a variable to all expressions that may use it.

$$\mathrm{DU}(\mathtt{v}, k) = \{q \mid q \text{ uses } \mathtt{v} \text{ and } (\mathtt{v}, k) \text{ reaches } q\}$$

DU chains: all $p \to q$ such that $q \in \mathrm{DU}(\mathtt{x}, p)$

Set $\mathrm{DU}(\mathtt{v}, k)$ can be computed as the “inverse” of UD:

$$\mathrm{DU}(\mathtt{v}, k) = \{q \mid k \in \mathrm{UD}(\mathtt{v}, q)\}$$



$$\mathrm{DU}(\mathtt{x}, 2) = \{q \mid 2 \in \mathrm{UD}(\mathtt{x}, q)\}$$

DU chains from reaching definitions

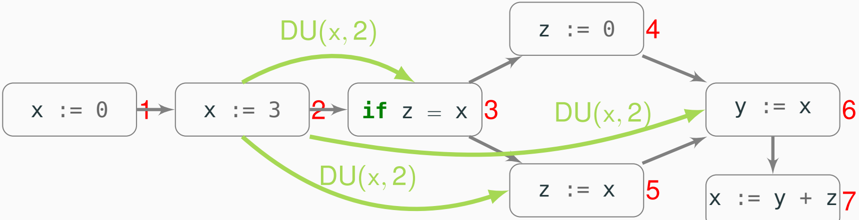
Definition-use chains (DU chains): link from each assignment of a variable to all expressions that may use it.

$$\mathrm{DU}(\mathtt{v}, k) \;=\; \{q \mid q \text{ uses } \mathtt{v} \text{ and } (\mathtt{v}, k) \text{ reaches } q\}$$

DU chains: all $p \to q$ such that $q \in \mathrm{DU}(\mathtt{x}, p)$

Set $\mathrm{DU}(\mathtt{v}, k)$ can be computed as the “inverse” of UD:

$$\mathrm{DU}(\mathtt{v}, k) \;=\; \{q \mid k \in \mathrm{UD}(\mathtt{v}, q)\}$$



$$\mathrm{DU}(\mathtt{x}, 2) \;=\; \{q \mid 2 \in \mathrm{UD}(\mathtt{x}, q)\}$$

Data-flow analysis

Slicing

Program slicing: the idea

What statements potentially affect the value of `sum` printed at line 8?

```
1 sum := 0
2 prod := 1
3 k := 0
4 while k < y
5   sum := sum + x
6   prod := prod * x
7   k := k + 1
8 print(sum)
9 print(prod)
```

The program that only includes the statements that affect `sum` at 8 is called a program slice.

Program slicing: the idea

What statements potentially affect the value of `sum` printed at line 8?

```
1 sum := 0
2 prod := 1
3 k := 0
4 while k < y
5   sum := sum + x
6   prod := prod * x
7   k := k + 1
8 print(sum)
9 print(prod)
```

```
1 sum := 0
```

The program that only includes the statements that affect `sum` at 8 is called a program slice.

Program slicing: the idea

What statements potentially affect the value of `sum` printed at line 8?

```
1 sum := 0
2 prod := 1
3 k := 0
4 while k < y
5   sum := sum + x
6   prod := prod * x
7   k := k + 1
8 print(sum)
9 print(prod)
```

```
1 sum := 0

3 k := 0
```

The program that only includes the statements that affect `sum` at 8 is called a program slice.

Program slicing: the idea

What statements potentially affect the value of `sum` printed at line 8?

```
1 sum := 0
2 prod := 1
3 k := 0
4 while k < y
5   sum := sum + x
6   prod := prod * x
7   k := k + 1
8 print(sum)
9 print(prod)
```

```
1 sum := 0

3 k := 0
4 while k < y
```

The program that only includes the statements that affect `sum` at 8 is called a program slice.

Program slicing: the idea

What statements potentially affect the value of `sum` printed at line 8?

```
1 sum := 0
2 prod := 1
3 k := 0
4 while k < y
5   sum := sum + x
6   prod := prod * x
7   k := k + 1
8 print(sum)
9 print(prod)
```

```
1 sum := 0

3 k := 0
4 while k < y
5   sum := sum + x
```

The program that only includes the statements that affect `sum` at 8 is called a program slice.

Program slicing: the idea

What statements potentially affect the value of `sum` printed at line 8?

```
1 sum := 0
2 prod := 1
3 k := 0
4 while k < y
5   sum := sum + x
6   prod := prod * x
7   k := k + 1
8 print(sum)
9 print(prod)
```

```
1 sum := 0

3 k := 0
4 while k < y
5   sum := sum + x

7   k := k + 1
```

The program that only includes the statements that affect `sum` at 8 is called a program slice.

Program slicing: the idea

What statements potentially affect the value of `sum` printed at line 8?

```
1 sum := 0
2 prod := 1
3 k := 0
4 while k < y
5   sum := sum + x
6   prod := prod * x
7   k := k + 1
8 print(sum)
9 print(prod)
```

```
1 sum := 0

3 k := 0
4 while k < y
5   sum := sum + x

7   k := k + 1
8 print(sum)
```

The program that only includes the statements that affect `sum` at 8 is called a program slice.

Program slicing

The program slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that may affect the values of variables at ℓ

If we only observe variables at location ℓ, we cannot distinguish a run of P from a run of its slice according to ℓ.

Program slicing

The program slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that may affect the values of variables at ℓ

If we only observe variables at location ℓ, we cannot distinguish a run of P from a run of its slice according to ℓ.

```
sum := 0
prod := 1
k := 0
while k < y
  sum := sum + x
  prod := prod * x
  k := k + 1
print(sum)
print(prod)
```

```
sum := 0
k := 0
while k < y
  sum := sum + x
  k := k + 1
print(sum)
```

Applications of program slicing

Several program analyses and optimizations are based on slicing:

debugging: by using the location of failure as slicing criterion, programmers can winnow statements where the error originates from the others

testing: slicing a failing test is a way of shrinking its size without losing its failure-triggering capability

parallelization: statements in separate slices can be executed in parallel without running into race conditions

Different approaches to slicing

Slicing can be done statically or dynamically.

Static slicing is based on general dependencies between statements, and hence it does not depend on particular inputs.

Dynamic slicing is based on the dependencies between statements that occur with specific inputs, and hence it is in general more precise (smaller slices).

Slicing can work forward or backward.

Backward slicing: given a statement ℓ, find which other (previous) statements affect ℓ.

Forward slicing: given a statement ℓ, find which other (following) statements are affected by ℓ.

Program slicing: rigorous definition

The backward program slice of program P according to slicing criterion ℓ is a program S with the following properties:

- S is obtained by deleting zero or more statements from P
- if P halts on some input X, then the values of variables at ℓ are the same in $P(X)$ and in $S(X)$ every time execution reaches ℓ

If we only observe variables at location $\underline{\ell}$, we cannot distinguish a run of P from a run of its slice according to ℓ.

Program slicing: rigorous definition

The backward program slice of program P according to slicing criterion ℓ is a program S with the following properties:

- S is obtained by deleting zero or more statements from P
- if P halts on some input X, then the values of variables at ℓ are the same in $P(X)$ and in $S(X)$ every time execution reaches ℓ

If we only observe variables at location ℓ, we cannot distinguish a run of P from a run of its slice according to ℓ.

To construct S we analyze any possible dependencies between ℓ and other statements:

data dependencies: corresponding to reaching definitions of variables used at ℓ

control-flow dependencies: corresponding to branching statements that may determine if execution reaches ℓ

Data dependence graph

The data dependence graph captures definition-usage dependencies between any pairs of nodes a and b in the CFG:

$$a \longrightarrow b \quad \text{iff} \quad b \in \mathrm{DU}(\mathsf{v}, a)$$

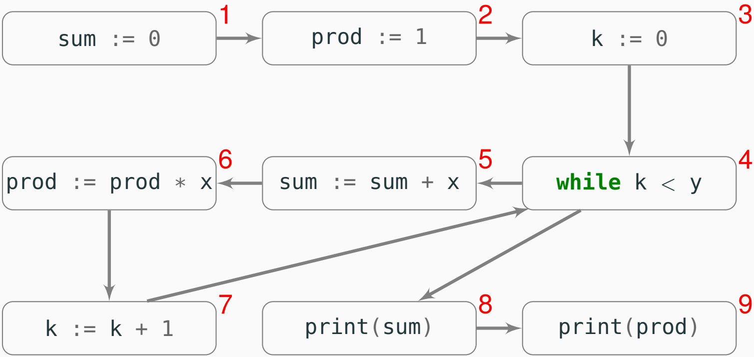
(We omit self-loops for simplicity.)

Data dependence graph

The data dependence graph captures definition-usage dependencies between any pairs of nodes *a* and *b* in the CFG:

$$a \longrightarrow b \quad \text{iff} \quad b \in \mathrm{DU}(\mathrm{v}, a)$$

(We omit self-loops for simplicity.)

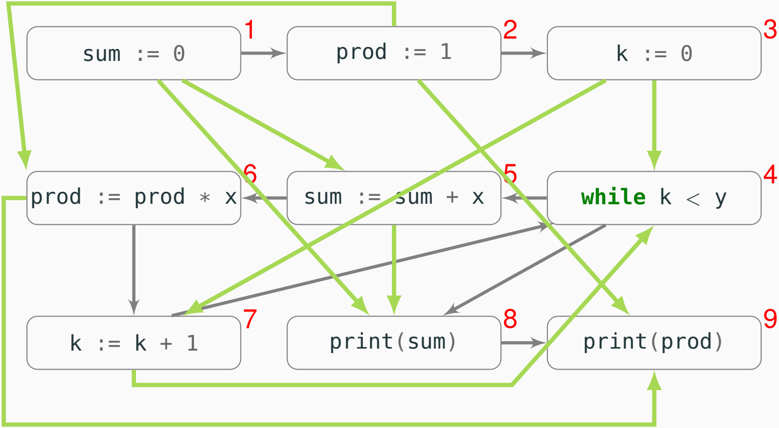


Data dependence graph

The data dependence graph captures definition-usage dependencies between any pairs of nodes *a* and *b* in the CFG:

$$a \longrightarrow b \quad \text{iff} \quad b \in \mathrm{DU}(\mathtt{v}, a)$$

(We omit self-loops for simplicity.)



Control dependence graph

The control dependence graph captures dependencies between branching statements and statements that may or may not execute according to which branch was taken.

$$a \dashrightarrow b \quad \text{iff} \quad \begin{array}{c} \text{block } a \text{ is a branch} \\ \text{and} \\ \text{branch } a\text{'s outcome determines whether } b \text{ executes} \end{array}$$

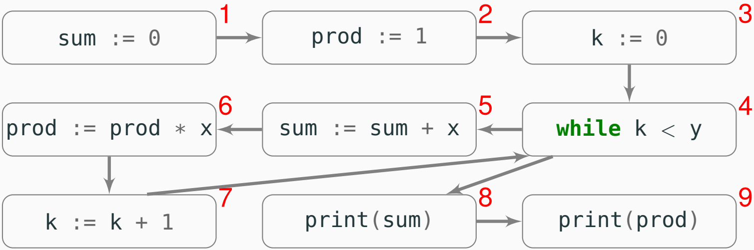
(We also add a special ENTRY node on which all statements not within a control structure depend.)

Control dependence graph

The control dependence graph captures dependencies between branching statements and statements that may or may not execute according to which branch was taken.

$a \dashrightarrow b$ iff block a is a branch and branch a's outcome determines whether b executes

(We also add a special ENTRY node on which all statements not within a control structure depend.)

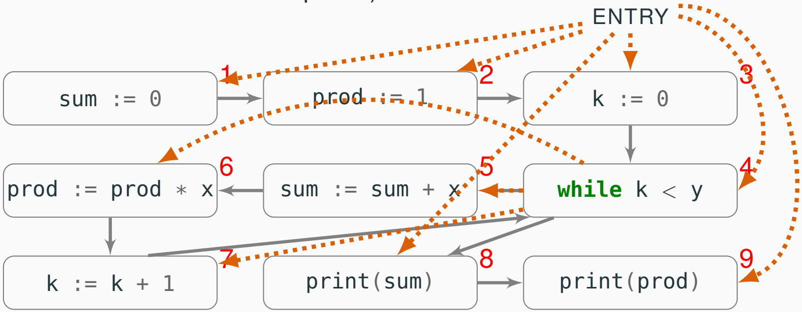


Control dependence graph

The control dependence graph captures dependencies between branching statements and statements that may or may not execute according to which branch was taken.

$a \dashrightarrow b$ iff block a is a branch and branch a's outcome determines whether b executes

(We also add a special ENTRY node on which all statements not within a control structure depend.)

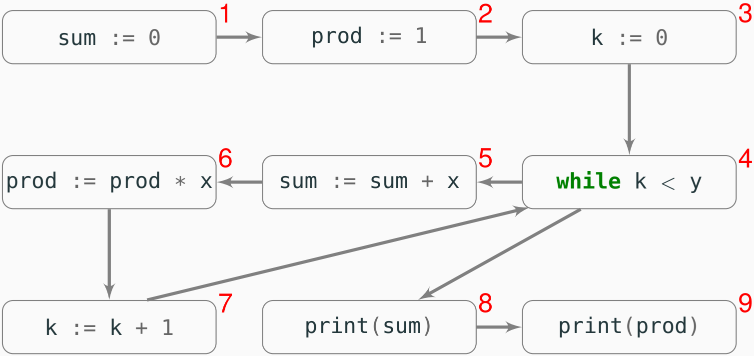


Program dependence graph

The program dependence graph (PDG) combines the data dependence and control dependence graphs.

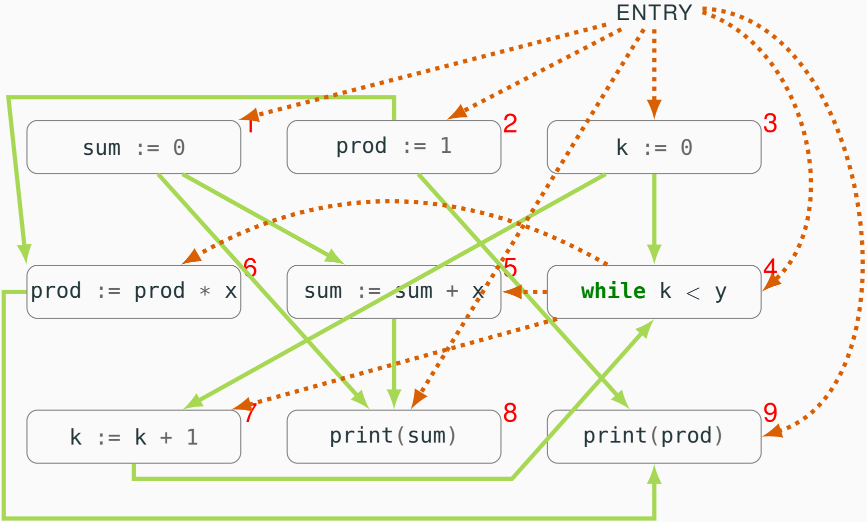
Program dependence graph

The program dependence graph (PDG) combines the data dependence and control dependence graphs.



Program dependence graph

The program dependence graph (PDG) combines the data dependence and control dependence graphs.



Slicing using the PDG

To build a backward slice S using the PDG:

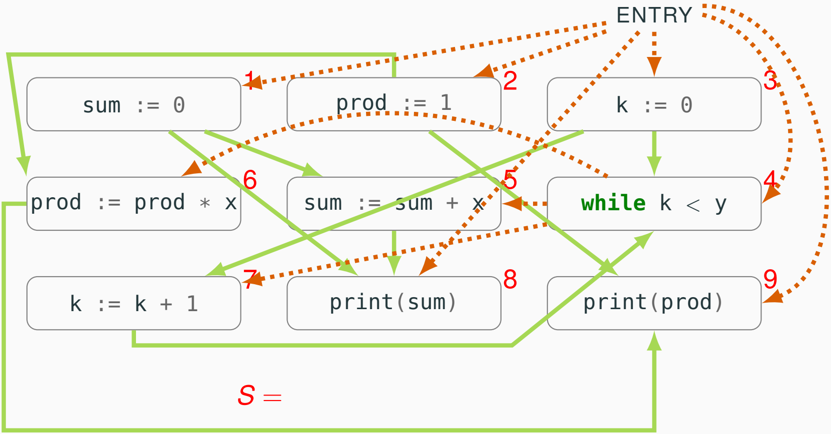
1. initially: $S = \{\ell\}$, where ℓ is the slicing criterion
2. add to S all nodes on which nodes in S transitively depend (data or control dependencies)

This corresponds to all nodes s such that $\ell \leftarrow^{+} s$, where $\leftarrow^{+}$ is the transitive closure of the inverse edge relation $\leftarrow$ in the PDG.

Slicing using the PDG

To build a backward slice S using the PDG:

1. initially: $S = \{\ell\}$, where ℓ is the slicing criterion
2. add to S all nodes on which nodes in S transitively depend (data or control dependencies)

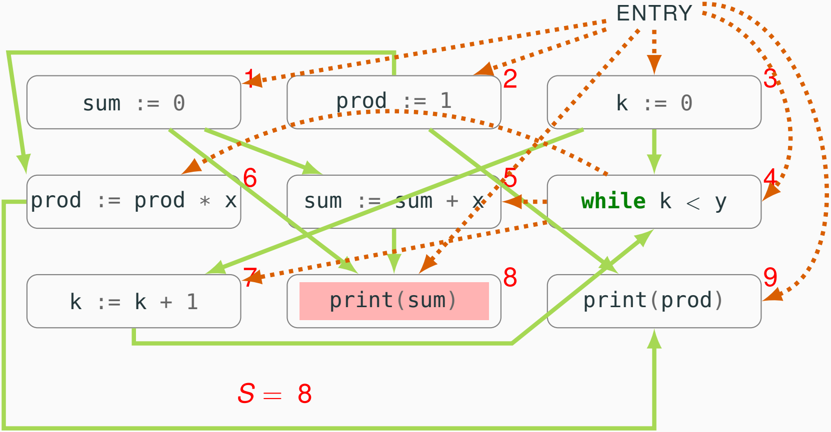


$S =$

Slicing using the PDG

To build a backward slice S using the PDG:

1. initially: $S = \{\ell\}$, where ℓ is the slicing criterion
2. add to S all nodes on which nodes in S transitively depend (data or control dependencies)

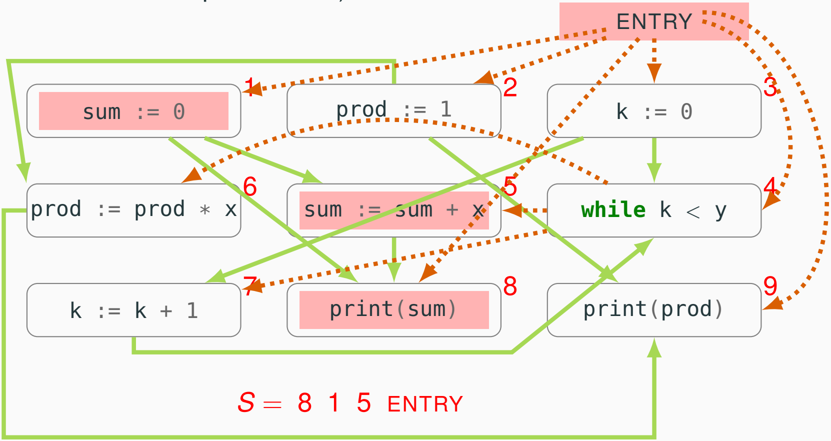


$S =$ 8

Slicing using the PDG

To build a backward slice S using the PDG:

1. initially: $S = \{\ell\}$, where ℓ is the slicing criterion
2. add to S all nodes on which nodes in S transitively depend (data or control dependencies)

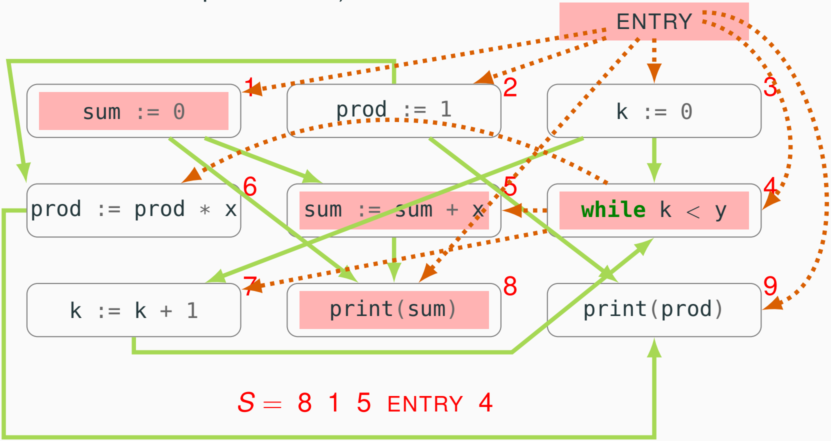


$S =$ 8 1 5 ENTRY

Slicing using the PDG

To build a backward slice S using the PDG:

1. initially: $S = \{\ell\}$, where ℓ is the slicing criterion
2. add to S all nodes on which nodes in S transitively depend (data or control dependencies)

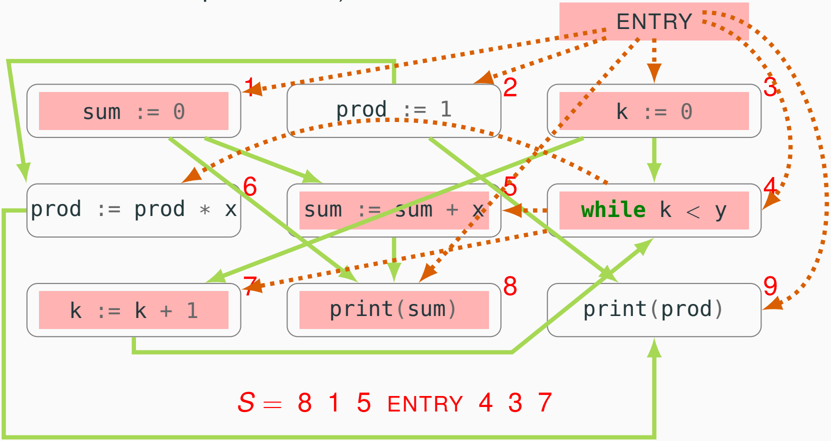


$S =$ 8 1 5 ENTRY 4

Slicing using the PDG

To build a backward slice S using the PDG:

1. initially: $S = \{\ell\}$, where ℓ is the slicing criterion
2. add to S all nodes on which nodes in S transitively depend (data or control dependencies)



$S =$ 8 1 5 ENTRY 4 3 7

Data-flow analysis

Static analysis tools example: Frama-C

A mini demo of Frama-C

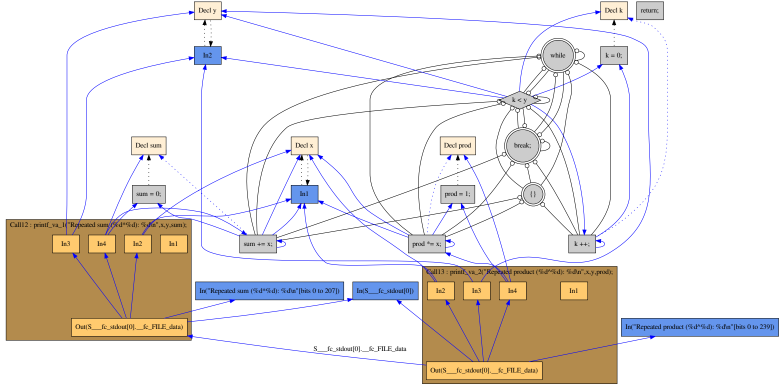
Let's perform some static analyses of the following example (already used to demonstrate slicing):

```c
// print x*y and x^y
void sum_prod(int x, int y)
{
  int sum, prod, k;
  sum = 0;
  prod = 1;
  k = 0;
  while (k < y) {
        sum = sum + x;
        prod = prod * x;
        k = k + 1;
  }
  printf("Repeated sum (%d*%d): %d\n", x, y, sum);
  printf("Repeated product (%d^%d): %d\n", x, y, prod);
}
```

The simplest way to use Frama-C is through its GUI: `frama-c-gui` (open a new project, and load the source file).

Frama-C: Program dependence graph

```
> frama-c -pdg -pdg-dot="pdg" example.c
# generate PDG and store it as DOT file 'pdg.sum_prod.dot'
```



Frama-C: Program dependence graph

```
> frama-c -pdg -pdg-dot="pdg" example.c
# generate PDG and store it as DOT file 'pdg.sum_prod.dot'
```

In the program dependence graph generated by Frama-C:

- Blue arrows go from a variable's usage to its reaching definitions
- Edges with an empty circle as arrowhead go from a statement to its control dependences
- Nodes `while` and `break` denote the loop's entry and exit points

Frama-C: Slicing

Slice using, as slicing criterion, variable `sum` at the exit of `sum_prod`:

```
> frama-c -main="sum_prod" -lib-entry -slice-value="sum" example.c \
            -then-on 'Slicing export' -print -ocode sum_exit_slice.c
```

```c
void sum_prod(int x, int y)
{
  int sum;
  int k;
  sum = 0;
  k = 0;
  while (k < y) {
    sum += x;
    k ++;
  }
  return;
}
```

Frama-C: Slicing

Slice using, as slicing criterion, variable `sum` at the exit of `sum_prod`:

```
> frama-c -main="sum_prod" -lib-entry -slice-value="sum" example.c \
            -then-on 'Slicing export' -print -ocode sum_exit_slice.c
```

Another way of specifying the slicing criterion is adding:

```
            /* slice pragma stmt; */
```

before the statement representing the slicing criterion. Then, call the analysis with `-slice-pragma="sum_prod"` instead of `-slice-value`.

Frama-C: Value analysis

An analysis of the range of values that variables may take, and the possible overflows that may result:

```
> frama-c -eva example.c

[eva:alarm] example.c:11: Warning:
  signed overflow. assert sum + x <= 2147483647;
[eva:alarm] example.c:12: Warning:
  signed overflow. assert prod * x <= 2147483647;

[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function sum_prod:
  sum in [0..2147483646]
  prod in [1..2147483647]
  k in [0..2147483647]
```

Data-flow analysis

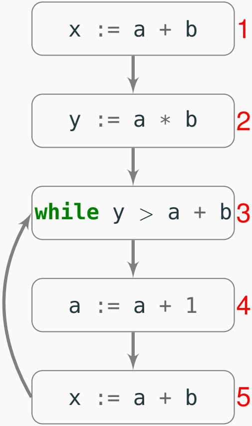
Available expressions analysis

Available expressions

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.

Available expressions

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.

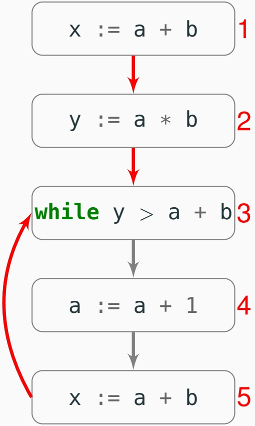


Expressions: `a + b`, `a * b`, `a + 1`.

Which of these expressions are available at (the entry of) 3?

Available expressions

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.



Expressions: `a + b`, `a * b`, `a + 1`.

Which of these expressions are available at (the entry of) 3? `a + b`

Available expressions analysis

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.

Available expressions analysis: for each program point, determine which expressions must be available at the point.

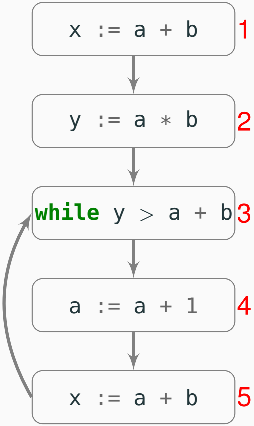
before the point/
at the entry of the block

Available expressions analysis

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.

Available expressions analysis: for each program point, determine which expressions must be available at the point.

before the point/
at the entry of the block



Available expressions analysis's output:

$$AE(1) = \{\}$$
$$AE(2) = \{\texttt{a + b}, \texttt{a}, \texttt{b}\}$$
$$AE(3) = \{\texttt{a + b}, \texttt{a}, \texttt{b}\}$$
$$AE(4) = \{\texttt{a + b}, \texttt{a}, \texttt{b}\}$$
$$AE(5) = \{\texttt{b}\}$$

Available expressions analysis

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.

Available expressions analysis: for each program point, determine which expressions must be available at the point.

under-approximation

before the point/
at the entry of the block

A must analysis is an under-approximation:
$AE(k)$ is a subset of the available expressions at k.

- if $E \in AE(k)$, E is definitely available at k
- if $E \notin AE(k)$, E may or may not be available at k (for example because it is available along certain paths but not along others)

The analysis has to be sound, and then as precise as possible given the information available in the CFG.

Available expressions analysis: applications

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.

If an expression E is available it needs not be recomputed; thus, we can save the value in its first computation in each path, and then read the saved value instead if computing it again. This improves performance the more computing E is expensive.

Available expressions analysis: applications

An expression *E* is available at block *k*
if *E* was evaluated and not later modified on all paths that reach *k*.

If an expression *E* is available it needs not be recomputed; thus, we can save the value in its first computation in each path, and then read the saved value instead if computing it again. This improves performance the more computing *E* is expensive.

```
x := f(a, b)
y := a * b
while y > f(a, b)
  a := a + 1
  x := f(a, b)
```

Available expressions analysis: applications

An expression E is available at block k
if E was evaluated and not later modified on all paths that reach k.

If an expression E is available it needs not be recomputed; thus, we can save the value in its first computation in each path, and then read the saved value instead if computing it again. This improves performance the more computing E is expensive.

```
1  x := f(a, b)
2  y := a * b
3  while y > f(a, b)
4    a := a + 1
5    x := f(a, b)
```

```
fab := f(a, b) ; x := fab
y := a * b
while y > fab
  a := a + 1
  fab := f(a, b) ; x := fab
```

Expression `f(a, b)` is available at 3. Hence, we can save its value in a fresh variable `fab` and read it at 3. Assuming the computation of `f` is expensive, this avoids repeating it when not necessary. Typically `f` has to be side-effect free for this optimization to be safe.

Formalizing available expressions analysis

We formalize the idea of available expressions analysis as an equation system:

- $AE_{\text{IN}}(k)$ and $AE_{\text{OUT}}(k)$ are variables over domain $\wp(\mathcal{E})$, where $\mathcal{E}$ is the set of all program expressions
- the equations formalize the relations:

$$AE_{\text{IN}}(k) = \bigcap_{h \text{ direct predecessor of } k} AE_{\text{OUT}}(h)$$

$$AE_{\text{OUT}}(k) = (AE_{\text{IN}}(k) \setminus \text{“changed at } k\text{”}) \cup \text{“not changed at } k\text{”}$$

Formalizing available expressions analysis

We formalize the idea of available expressions analysis as an equation system:

- $AE_{\mathrm{IN}}(k)$ and $AE_{\mathrm{OUT}}(k)$ are variables over domain $\wp(\mathcal{E})$, where $\mathcal{E}$ is the set of all program expressions
- the equations formalize the relations:

$$
\begin{aligned}
AE_{\mathrm{IN}}(k) &= \bigcap_{h \text{ direct predecessor of } k} AE_{\mathrm{OUT}}(h) \\
AE_{\mathrm{OUT}}(k) &= (AE_{\mathrm{IN}}(k) \setminus \text{“changed at } k\text{”}) \cup \text{“not changed at } k\text{”}
\end{aligned}
$$

must analysis: available along all paths

The analysis result is the greatest solution of the equation system – greatest so that the under-approximation is as precise as possible.

Data-flow equations

for every node h that precedes k in the CFG (i.e., h is a direct predecessor of k)

For every block k:

$$AE_{\text{IN}}(k) = \bigcap_{(h \to k) \in \text{CFG}} AE_{\text{OUT}}(h)$$

$$AE_{\text{OUT}}(k) = \left(AE_{\text{IN}}(k) \setminus \text{kill}_{AE}(k)\right) \cup \text{gen}_{AE}(k)$$

Data-flow equations

For every block k:

for every node h that precedes k in the CFG (i.e., h is a direct predecessor of k)

$$AE_{\text{IN}}(k) = \bigcap_{(h \to k) \in \text{CFG}} AE_{\text{OUT}}(h)$$

$$AE_{\text{OUT}}(k) = \left(AE_{\text{IN}}(k) \setminus \text{kill}_{AE}(k)\right) \cup \text{gen}_{AE}(k)$$

If j is an initial node, it has no predecessors, and hence $AE_{\text{IN}}(j) = \{\}$.

Data-flow equations

for every node h that precedes k in the CFG
(i.e., h is a direct predecessor of k)

For every block k:

$$AE_{\text{IN}}(k) = \bigcap_{(h \to k) \in \text{CFG}} AE_{\text{OUT}}(h)$$

$$AE_{\text{OUT}}(k) = \big(AE_{\text{IN}}(k) \setminus \text{kill}_{AE}(k)\big) \cup \text{gen}_{AE}(k)$$

If j is an initial node, it has no predecessors, and hence $AE_{\text{IN}}(j) = \{\}$.

all expressions containing v

We define kill_{AE} and gen_{AE} for every block type:

all subexpressions of E
not containing v

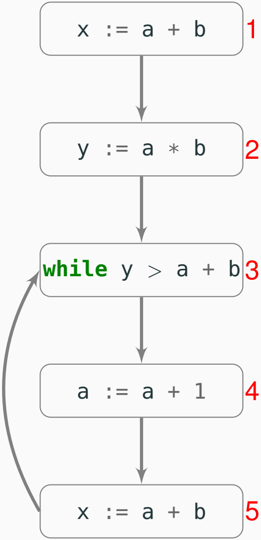
$$\text{kill}_{AE}(\texttt{skip}) = \{\} \qquad \text{gen}_{AE}(\texttt{skip}) = \{\}$$

$$\text{kill}_{AE}(\text{v} := E) = \{e \mid \text{v} \in e\} \qquad \text{gen}_{AE}(\text{v} := E) = \{e \mid e \in E \text{ and } \text{v} \notin e\}$$

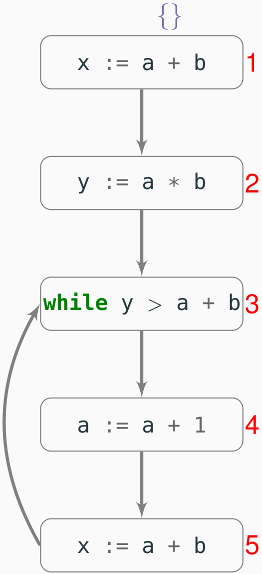
$$\text{kill}_{AE}(\texttt{if}/\texttt{while}\ C) = \{\} \qquad \text{gen}_{AE}(\texttt{if}/\texttt{while}\ C) = \{e \mid e \in C\}$$

all subexpressions of C

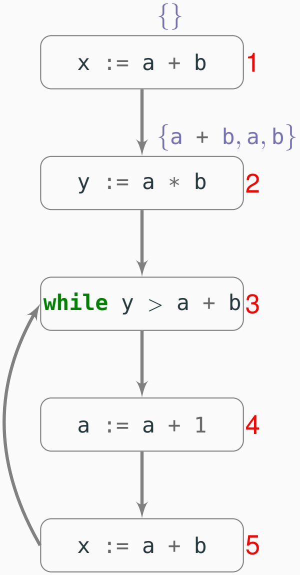
Available expressions analysis: example



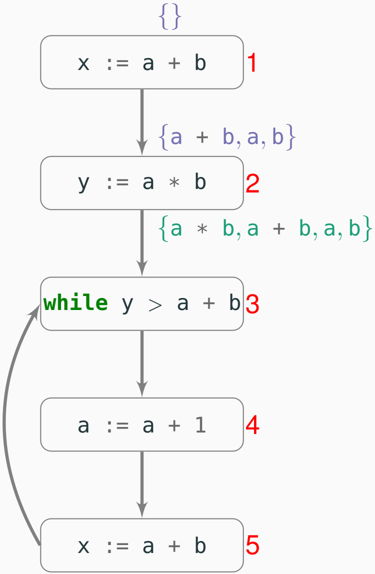
Available expressions analysis: example



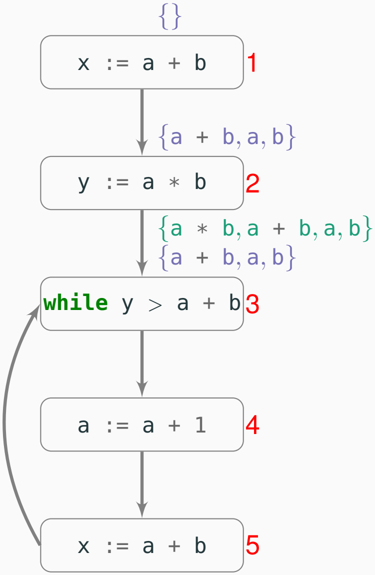
Available expressions analysis: example



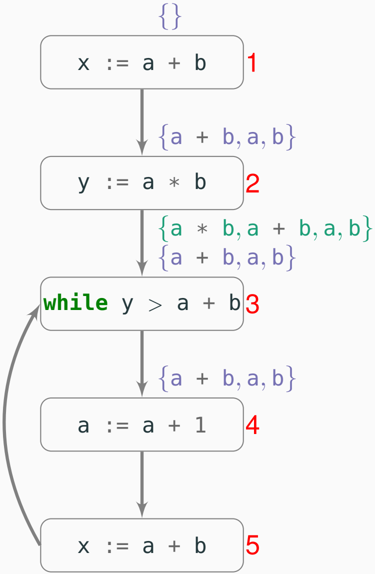
Available expressions analysis: example



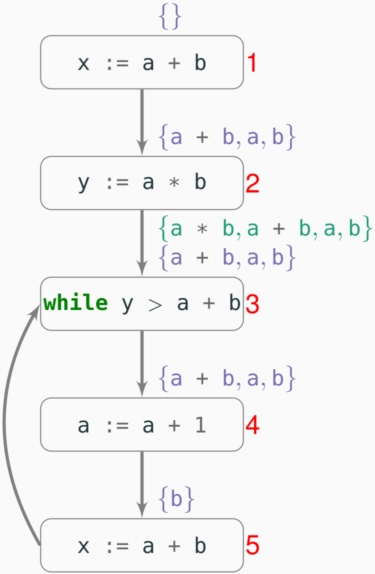
Available expressions analysis: example



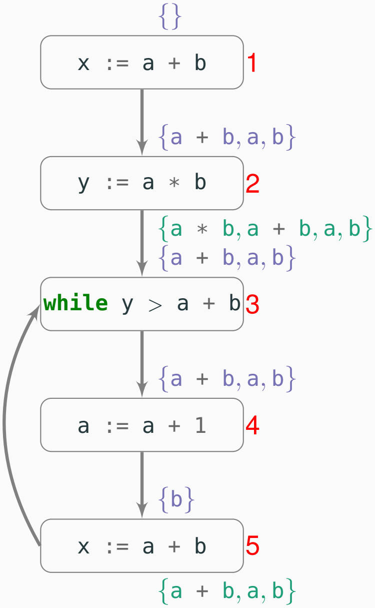
Available expressions analysis: example



Available expressions analysis: example



Available expressions analysis: example



May vs. must analyses

May analyses (such as LV) and must analyses (such as AE) are dual.

Accordingly, the notions of soundness and precision are formulated in a way that matches the way the analysis's results are used.

May vs. must analyses

May analyses (such as LV) and must analyses (such as AE) are dual.

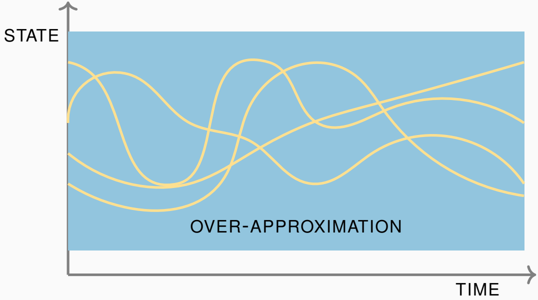
Accordingly, the notions of soundness and precision are formulated in a way that matches the way the analysis's results are used.

MAY ANALYSIS	MUST ANALYSIS
analysis approximates property P	
analysis output MAY	analysis ouput: $MUST$
example: $P =$ live variables	example $P =$ available expressions
property P is an error property	property P is a correctness property
if v is not live, then I can eliminate an assignment	if E is available, then I can eliminate an evaluation
over-approximation: $P \subseteq MAY$	under-approximation: $MUST \subseteq P$
sound: $x \notin MAY \Longrightarrow x \notin P$	sound: $x \in MUST \Longrightarrow x \in P$
imprecise: $x \in MAY \not\Longrightarrow x \in P$	imprecise: $x \notin MUST \not\Longrightarrow x \notin P$
most precise: least fixed point	most precise: greatest fixed point

Abstract interpretation

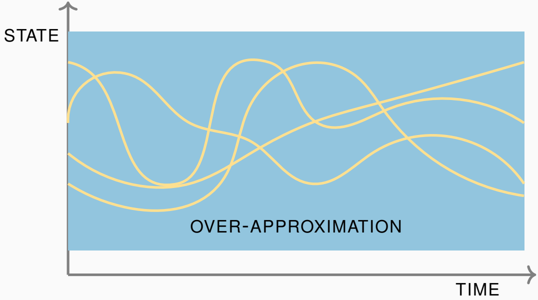
One framework to rule them all

The basic idea behind the data-flow analyses we have seen – as well as many other kinds of static analysis – is to abstract computations by keeping track of partial, simpler information – such as the variables that may be live.



One framework to rule them all

The basic idea behind the data-flow analyses we have seen – as well as many other kinds of static analysis – is to abstract computations by keeping track of partial, simpler information – such as the variables that may be live.



A crucial concern is correctness: how to ensure that a particular analysis is sound.

Abstract interpretation provides a general framework to construct program analyses and to establish their correctness.

Cousot & Cousot

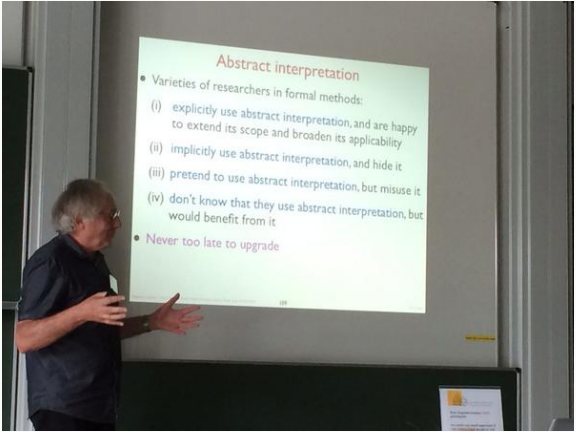
Abstract interpretation was invented by Patrick and Radhia Cousot in a seminal POPL paper published in 1977.

ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

Abstract interpretation was invented by Patrick and Radhia Cousot in a seminal POPL paper published in 1977.

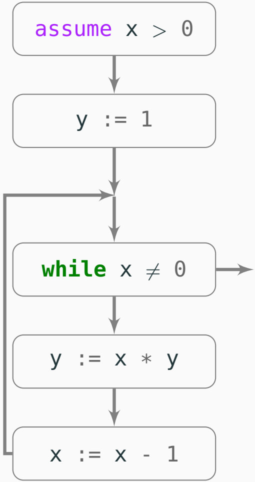


Abstract interpretation

Concrete and abstract computations

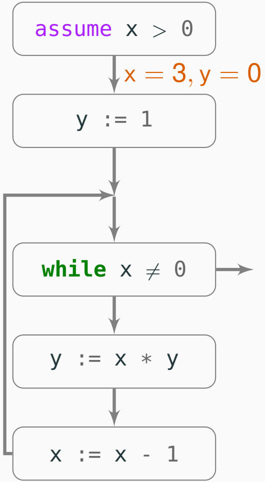
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



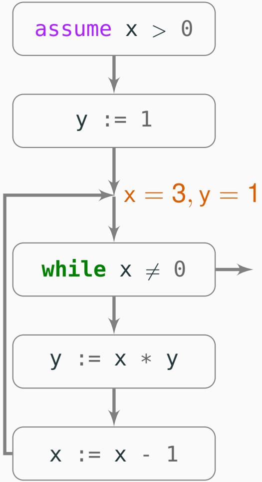
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



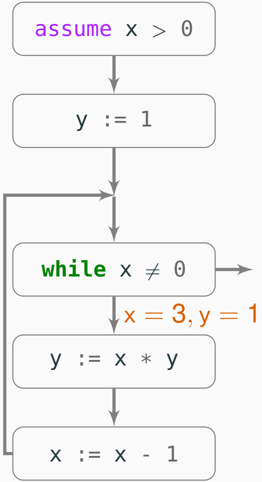
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



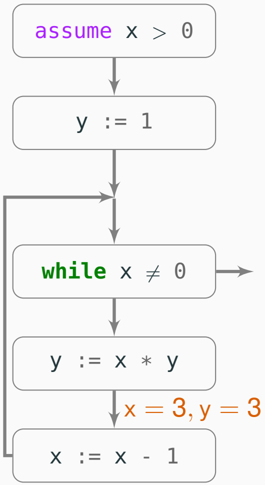
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



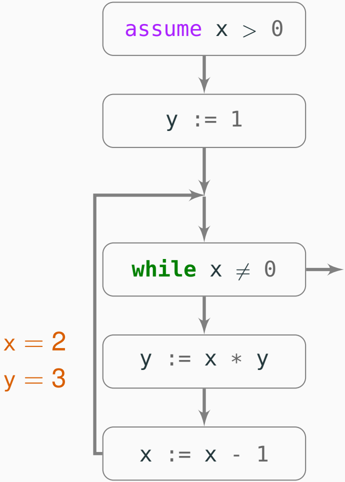
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



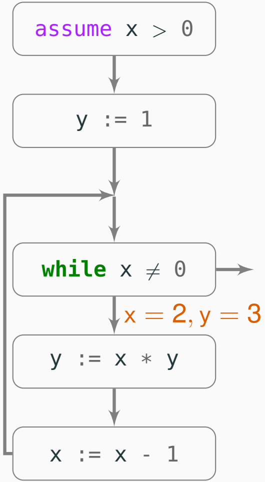
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



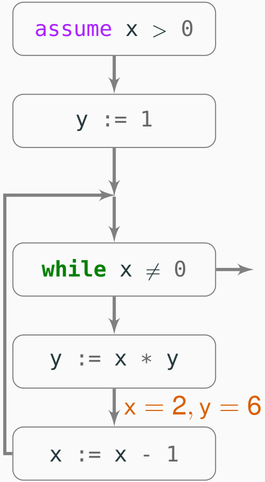
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



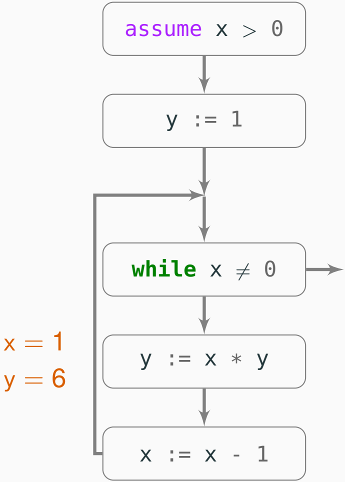
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



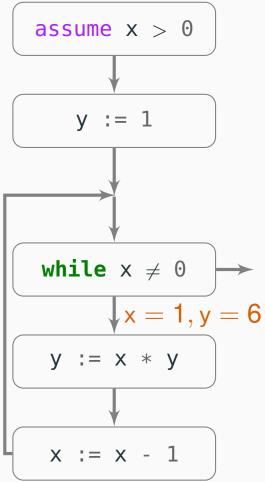
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



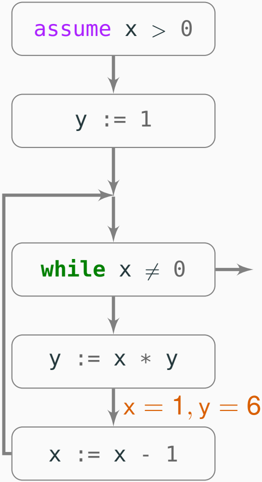
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



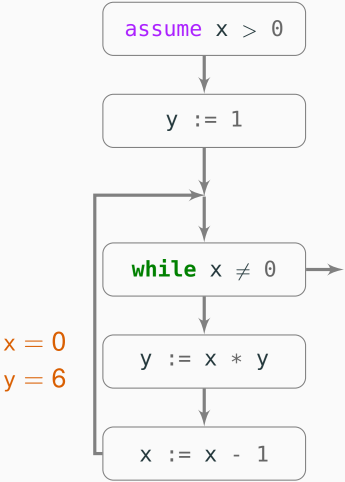
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



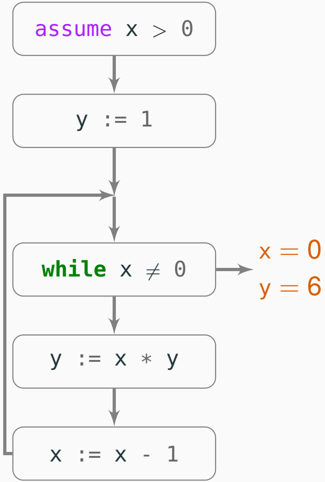
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



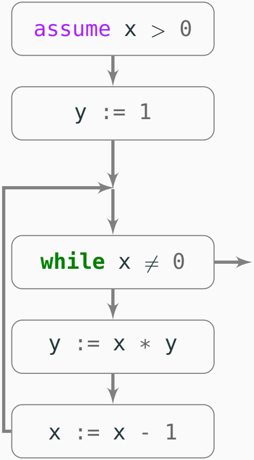
Concrete computations

A program defines a set of possible computations as sequences of states over a concrete domain according to its concrete semantics – for example, the programming language's operational semantics.



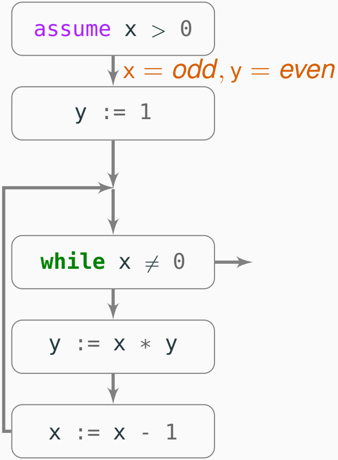
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



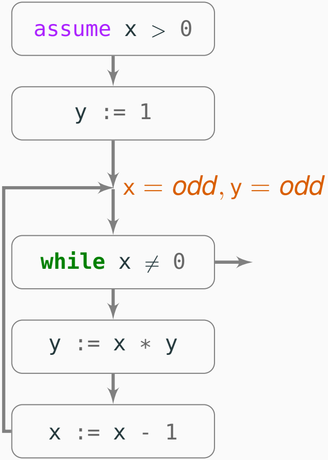
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



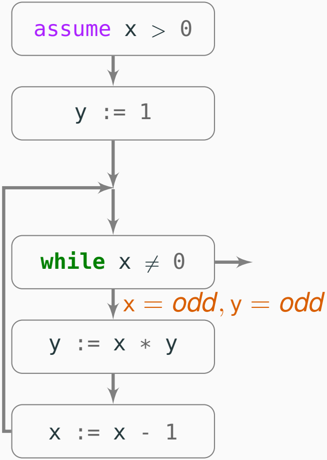
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



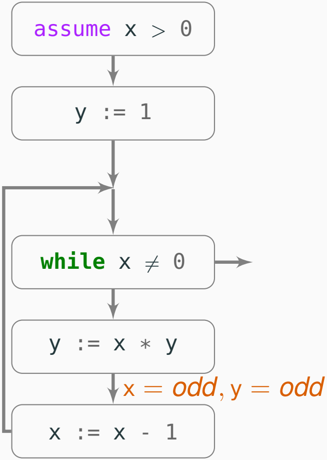
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



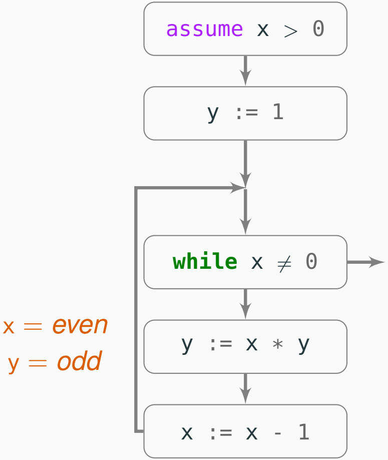
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



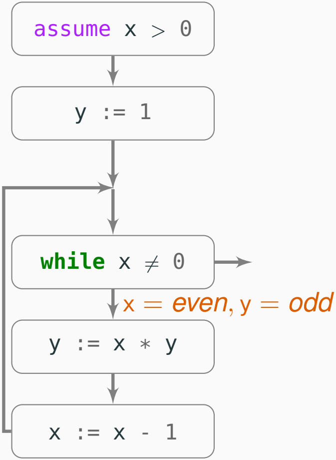
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



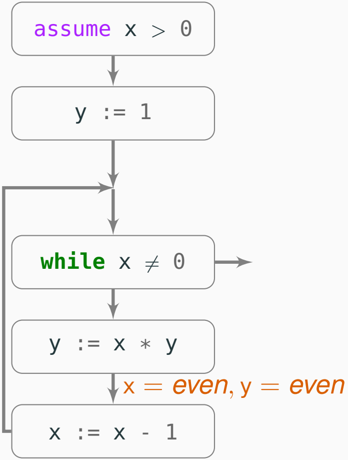
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



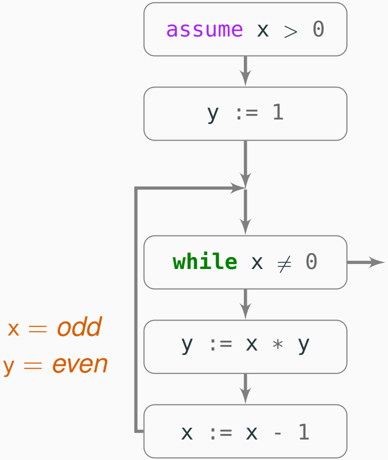
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



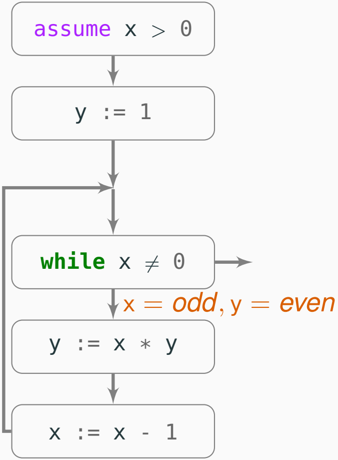
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



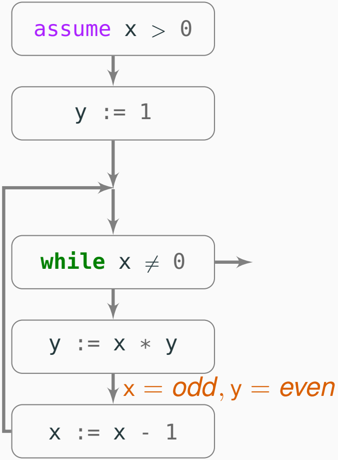
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



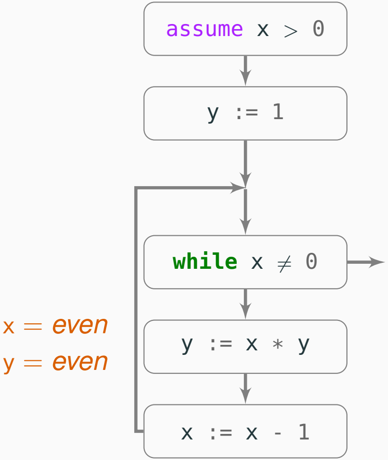
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



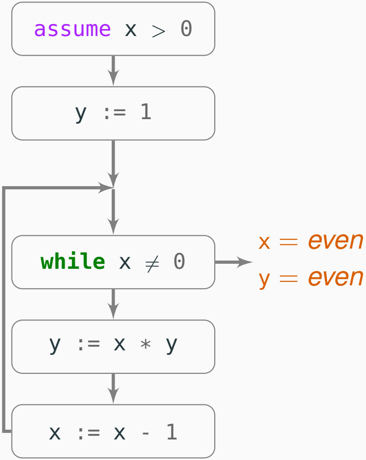
Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



Abstract computations

A static analysis defines a form of abstract semantics where computations are sequences of states over an abstract domain that keeps track of partial information about properties of a program's concrete state – for example, whether a variable is even or odd.



Abstract interpretation: main idea

Abstract interpretation is a framework for constructing abstract semantics and proving that they are sound with respect to the concrete semantics.

Contrast this to the a posteriori approach of data-flow analysis: first define an analysis, then prove that it is correct.

Abstract interpretation: main idea

As in the data-flow analyses, computations are captured by the possible values of variables at each program point.

Abstract interpretation: main idea

As in the data-flow analyses, computations are captured by the possible values of variables at each program point.

Now we label edges of the CFG (instead of nodes) because we want to express the possible values of variables before or after executing a statement.

Abstract interpretation: main idea

As in the data-flow analyses, computations are captured by the possible values of variables at each program point.

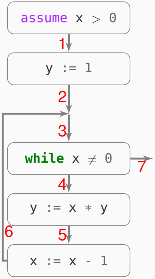
Now we label edges of the CFG (instead of nodes) because we want to express the possible values of variables before or after executing a statement.

Concrete state domain:

$$\mathit{State}\colon \mathit{Vars} \to \mathbb{Z}$$

Concrete semantics: set of possible concrete states at every program point

$$C\colon \mathit{Labels} \to \wp(\mathit{State})$$



Abstract state domain:

$$\mathit{AbstractState}\colon \mathit{Vars} \to \left\{ \begin{array}{l} \mathit{odd}, \\ \mathit{even} \end{array} \right\}$$

Abstract semantics: set of possible abstract states at every program point

$$A\colon \mathit{Labels} \to \wp(\mathit{AbstractState})$$

Collecting semantics

The collecting semantics C is a concrete semantics in data-flow fashion, giving the set of possible concrete states at every edge label:

$$C: \mathit{Labels} \to \wp(\mathit{State})$$

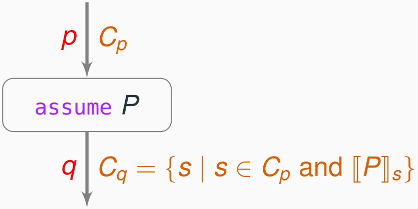
We define C for every block type in a CFG
(for brevity, we omit `skip` and `assert`, which are easy to add).

Collecting semantics

The collecting semantics C is a concrete semantics in data-flow fashion, giving the set of possible concrete states at every edge label:

$$C\colon \mathit{Labels} \to \wp(\mathit{State})$$

We define C for every block type in a CFG
(for brevity, we omit `skip` and `assert`, which are easy to add).

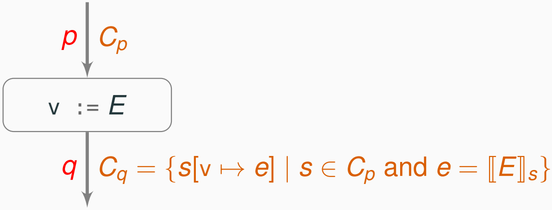


Collecting semantics

The collecting semantics C is a concrete semantics in data-flow fashion, giving the set of possible concrete states at every edge label:

$$C\colon \mathit{Labels} \to \wp(\mathit{State})$$

We define C for every block type in a CFG
(for brevity, we omit `skip` and `assert`, which are easy to add).

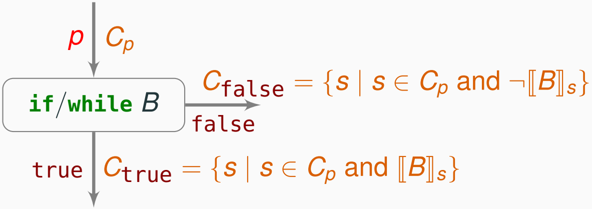


Collecting semantics

The collecting semantics C is a concrete semantics in data-flow fashion, giving the set of possible concrete states at every edge label:

$$C\colon \mathit{Labels} \to \wp(\mathit{State})$$

We define C for every block type in a CFG
(for brevity, we omit **`skip`** and `assert`, which are easy to add).

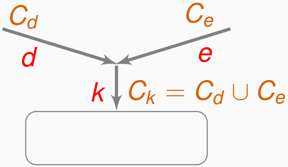


Collecting semantics

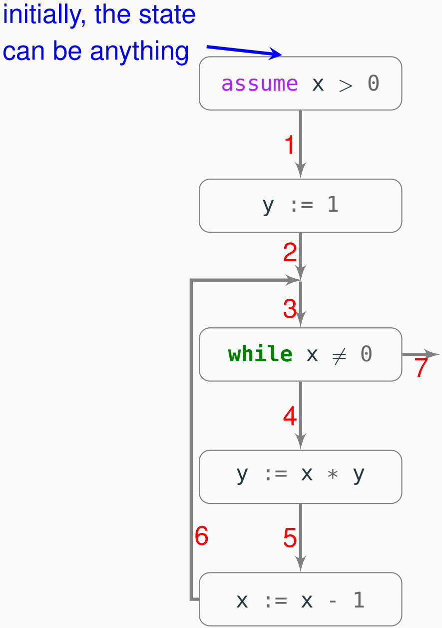
The collecting semantics C is a concrete semantics in data-flow fashion, giving the set of possible concrete states at every edge label:

$$C \colon \mathit{Labels} \to \wp(\mathit{State})$$

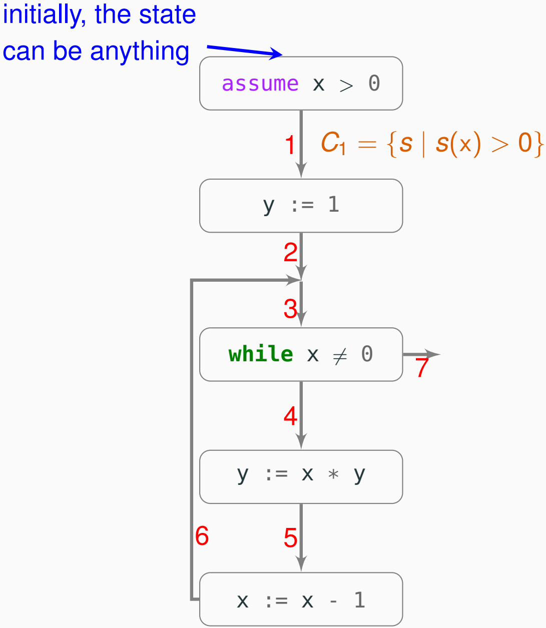
We define C for every block type in a CFG
(for brevity, we omit `skip` and `assert`, which are easy to add).



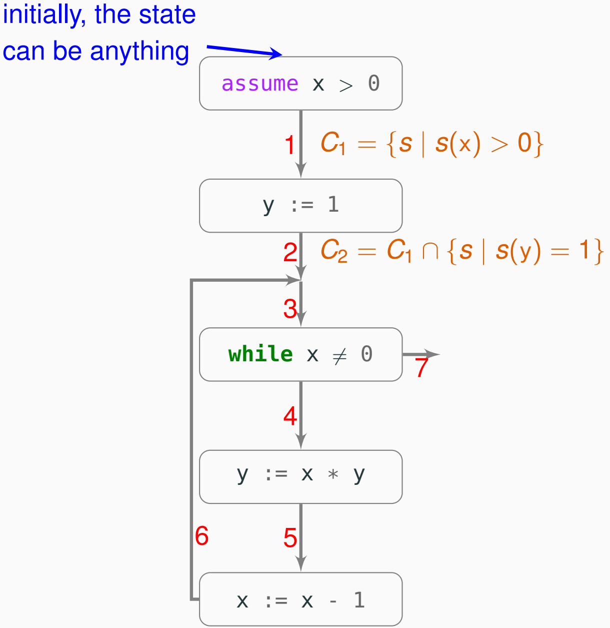
Collecting semantics: example



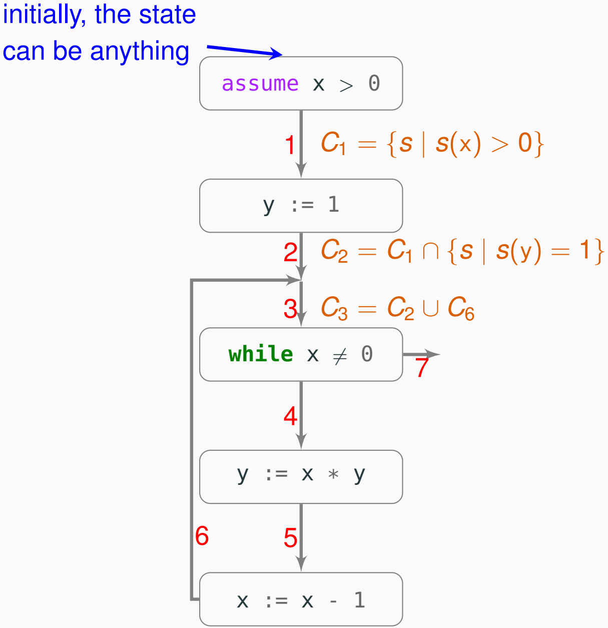
Collecting semantics: example



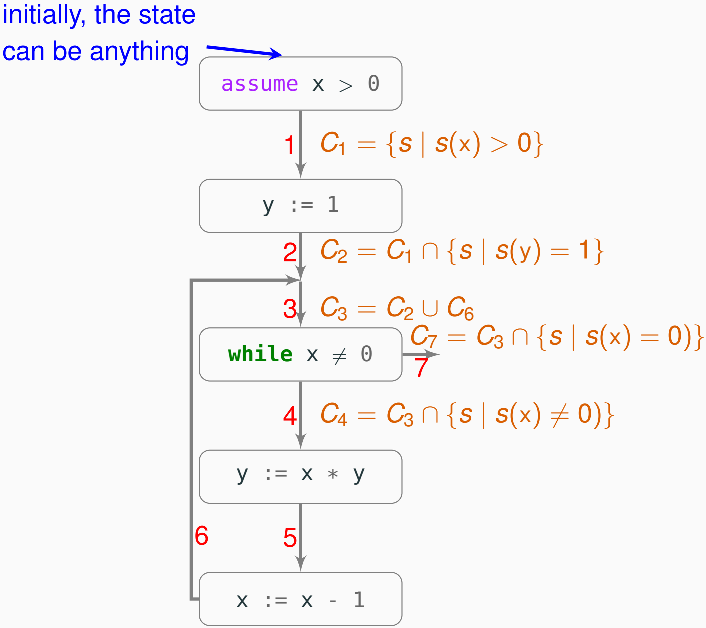
Collecting semantics: example



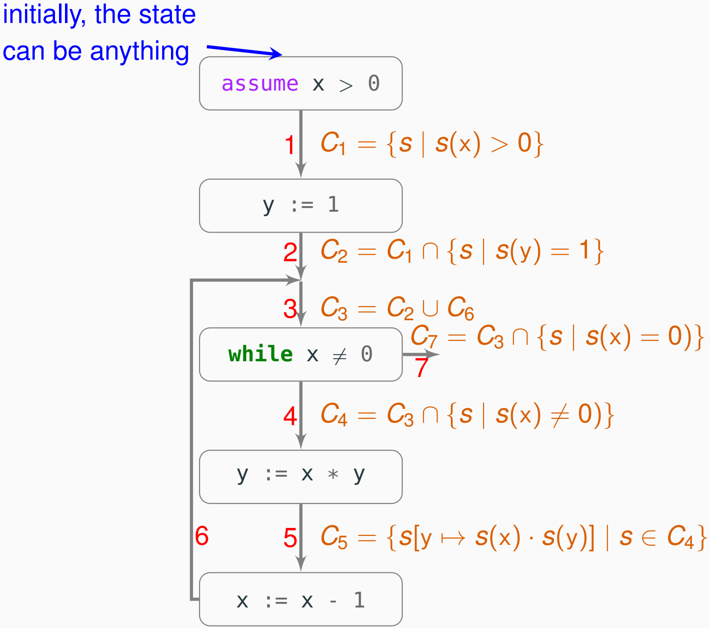
Collecting semantics: example



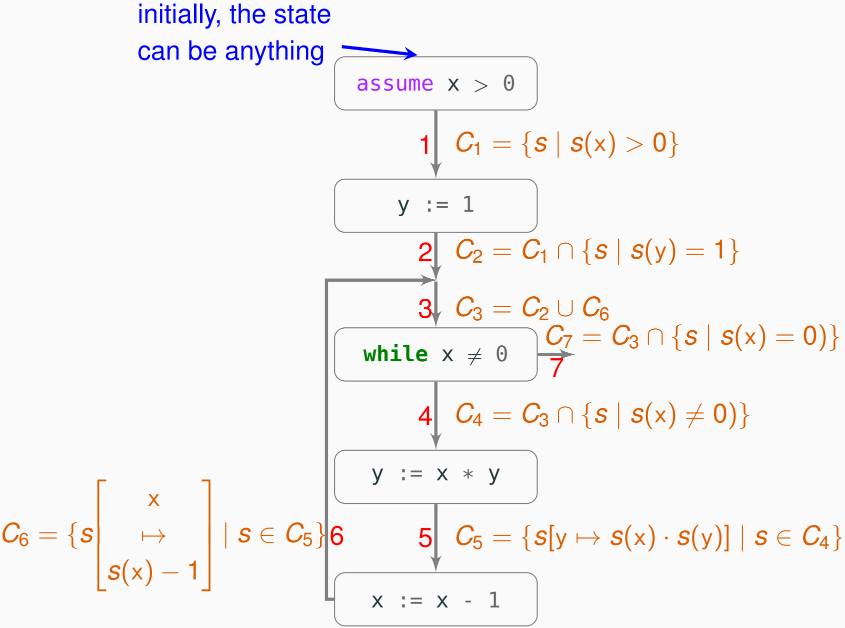
Collecting semantics: example



Collecting semantics: example



Collecting semantics: example



Collecting semantics: equation solving

The collecting semantics gives a set of equations that look a lot like data-flow equations – except for minor details such as that we have labels on edges instead of entry and exit of blocks.

$$
\begin{aligned}
C_1 &= \{s \mid s(\mathrm{x}) > 0\} \\
C_2 &= C_1 \cap \{s \mid s(\mathrm{y}) = 1\} \\
C_3 &= C_2 \cup C_6 \\
C_4 &= C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
C_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\} \\
C_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\} \\
C_7 &= C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

Collecting semantics: equation solving

The collecting semantics gives a set of equations that look a lot like data-flow equations – except for minor details such as that we have labels on edges instead of entry and exit of blocks.

$$
\begin{aligned}
C_1 &= \{s \mid s(\mathrm{x}) > 0\} \\
C_2 &= C_1 \cap \{s \mid s(\mathrm{y}) = 1\} \\
C_3 &= C_2 \cup C_6 \\
C_4 &= C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
C_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\} \\
C_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\} \\
C_7 &= C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

These equations satisfy the conditions of Tarski's fixed point theorem:

monotonicity: every function C_k is monotonic

lattice: the analysis domain is $\wp(\mathit{State})^7$, a complete lattice

Collecting semantics: equation solving

The collecting semantics gives a set of equations that look a lot like data-flow equations – except for minor details such as that we have labels on edges instead of entry and exit of blocks.

$$
\begin{aligned}
C_1 &= \{s \mid s(\mathrm{x}) > 0\} \\
C_2 &= C_1 \cap \{s \mid s(\mathrm{y}) = 1\} \\
C_3 &= C_2 \cup C_6 \\
C_4 &= C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
C_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\} \\
C_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\} \\
C_7 &= C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

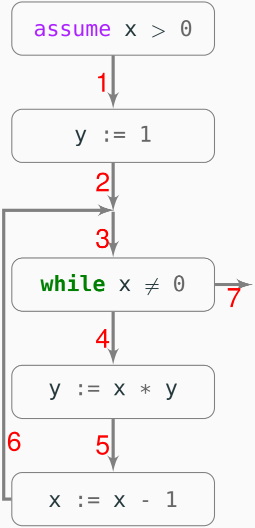
These equations satisfy the conditions of Tarski's fixed point theorem:

monotonicity: every function C_k is monotonic

lattice: the analysis domain is $\wp(State)^7$, a complete lattice

We can compute the concrete semantics by evaluating the equations starting from $\{\} \times \cdots \times \{\}$ until we reach a fixed point.

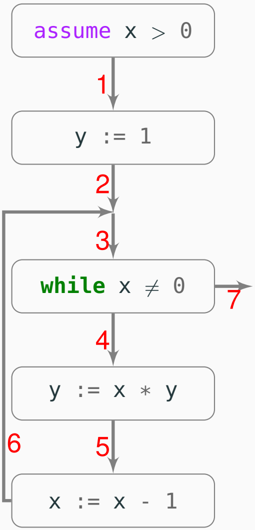
Fixed point concrete computation: example



$$
\begin{aligned}
C_1 &= \\
C_2 &= \\
C_3 &= \\
\\
C_4 &= \\
C_5 &= \\
C_6 &= \\
C_7 &=
\end{aligned}
$$

$$
\begin{aligned}
C_1 &= \{s \mid s(\mathrm{x}) > 0\} \\
C_2 &= C_1 \cap \{s \mid s(\mathrm{y}) = 1\} \\
C_3 &= C_2 \cup C_6 \\
C_4 &= C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
C_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\} \\
C_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\} \\
C_7 &= C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

Fixed point concrete computation: example

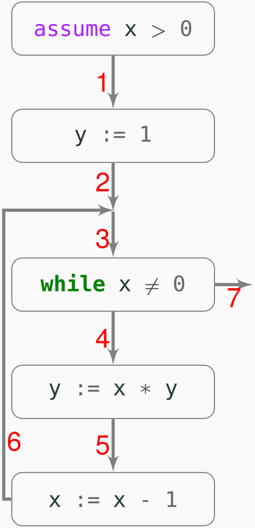


$$C_1 = \{\}$$
$$C_2 = \{\}$$
$$C_3 = \{\}$$

$$C_4 = \{\}$$
$$C_5 = \{\}$$
$$C_6 = \{\}$$
$$C_7 = \{\}$$

$$C_1 = \{s \mid s(\mathrm{x}) > 0\}$$
$$C_2 = C_1 \cap \{s \mid s(\mathrm{y}) = 1\}$$
$$C_3 = C_2 \cup C_6$$
$$C_4 = C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\}$$
$$C_5 = \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\}$$
$$C_6 = \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\}$$
$$C_7 = C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}$$

Fixed point concrete computation: example

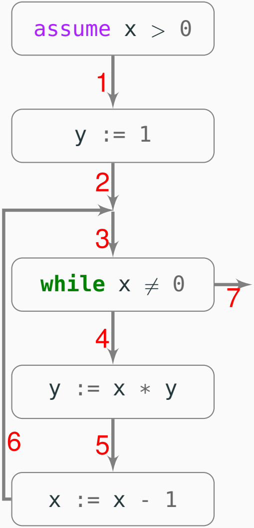


$$C_1 = \quad \{\mathrm{x} = m, \mathrm{y} = n \mid m > 0\}$$
$$C_2 = \{\}$$
$$C_3 = \{\}$$

$$C_4 = \{\}$$
$$C_5 = \{\}$$
$$C_6 = \{\}$$
$$C_7 = \{\}$$

$$C_1 = \{s \mid s(\mathrm{x}) > 0\}$$
$$C_2 = C_1 \cap \{s \mid s(\mathrm{y}) = 1\}$$
$$C_3 = C_2 \cup C_6$$
$$C_4 = C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\}$$
$$C_5 = \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\}$$
$$C_6 = \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\}$$
$$C_7 = C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}$$

Fixed point concrete computation: example

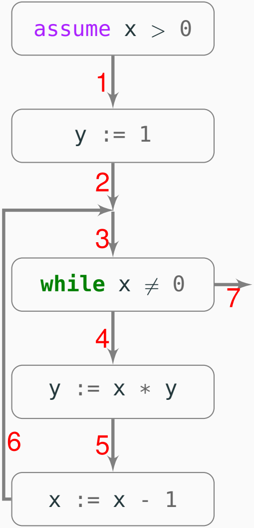


$$C_1 = \quad \{\mathrm{x} = m, \mathrm{y} = n \mid m > 0\}$$
$$C_2 = \quad \{\mathrm{x} = m, \mathrm{y} = 1 \mid m > 0\}$$
$$C_3 = \{\}$$

$$C_4 = \{\}$$
$$C_5 = \{\}$$
$$C_6 = \{\}$$
$$C_7 = \{\}$$

$$C_1 = \{s \mid s(\mathrm{x}) > 0\}$$
$$C_2 = C_1 \cap \{s \mid s(\mathrm{y}) = 1\}$$
$$C_3 = C_2 \cup C_6$$
$$C_4 = C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\}$$
$$C_5 = \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\}$$
$$C_6 = \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\}$$
$$C_7 = C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}$$

Fixed point concrete computation: example

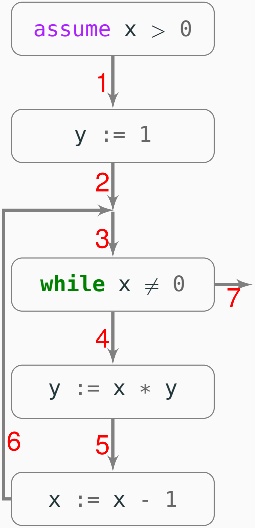


$$C_1 = \{x = m, y = n \mid m > 0\}$$
$$C_2 = \{x = m, y = 1 \mid m > 0\}$$
$$C_3 = \{x = m, y = 1 \mid m > 0\}$$

$$C_4 = \{\}$$
$$C_5 = \{\}$$
$$C_6 = \{\}$$
$$C_7 = \{\}$$

$$C_1 = \{s \mid s(x) > 0\}$$
$$C_2 = C_1 \cap \{s \mid s(y) = 1\}$$
$$C_3 = C_2 \cup C_6$$
$$C_4 = C_3 \cap \{s \mid s(x) \neq 0)\}$$
$$C_5 = \{s[y \mapsto s(x) \cdot s(y)] \mid s \in C_4\}$$
$$C_6 = \{s[x \mapsto s(x) - 1] \mid s \in C_5\}$$
$$C_7 = C_3 \cap \{s \mid s(x) = 0)\}$$

Fixed point concrete computation: example

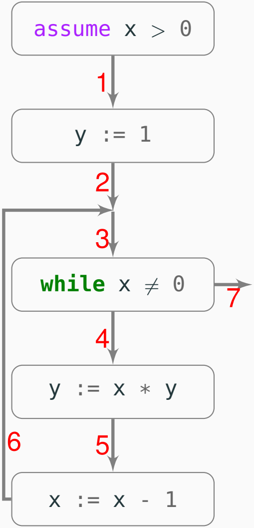


$$
\begin{aligned}
C_1 &= \quad \{\mathrm{x} = m, \mathrm{y} = n \mid m > 0\}\\
C_2 &= \quad \{\mathrm{x} = m, \mathrm{y} = 1 \mid m > 0\}\\
C_3 &= \quad \{\mathrm{x} = m, \mathrm{y} = 1 \mid m > 0\}
\end{aligned}
$$

$$
\begin{aligned}
C_4 &= \quad \{\mathrm{x} = m, \mathrm{y} = 1 \mid m > 0\}\\
C_5 &= \{\}\\
C_6 &= \{\}\\
C_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
C_1 &= \{s \mid s(\mathrm{x}) > 0\}\\
C_2 &= C_1 \cap \{s \mid s(\mathrm{y}) = 1\}\\
C_3 &= C_2 \cup C_6\\
C_4 &= C_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\}\\
C_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \cdot s(\mathrm{y})] \mid s \in C_4\}\\
C_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) - 1] \mid s \in C_5\}\\
C_7 &= C_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

Fixed point concrete computation: example

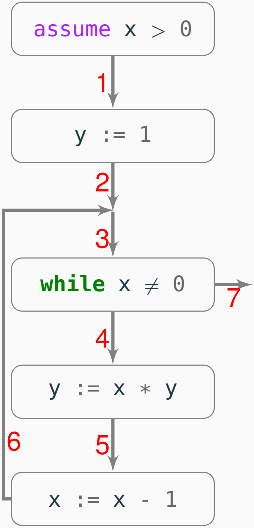


$$
\begin{aligned}
C_1 &= \{x = m, y = n \mid m > 0\}\\
C_2 &= \{x = m, y = 1 \mid m > 0\}\\
C_3 &= \{x = m, y = 1 \mid m > 0\}
\end{aligned}
$$

$$
\begin{aligned}
C_4 &= \{x = m, y = 1 \mid m > 0\}\\
C_5 &= \{x = m, y = m \mid m > 0\}\\
C_6 &= \{\}\\
C_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
C_1 &= \{s \mid s(x) > 0\}\\
C_2 &= C_1 \cap \{s \mid s(y) = 1\}\\
C_3 &= C_2 \cup C_6\\
C_4 &= C_3 \cap \{s \mid s(x) \neq 0)\}\\
C_5 &= \{s[y \mapsto s(x) \cdot s(y)] \mid s \in C_4\}\\
C_6 &= \{s[x \mapsto s(x) - 1] \mid s \in C_5\}\\
C_7 &= C_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

Fixed point concrete computation: example

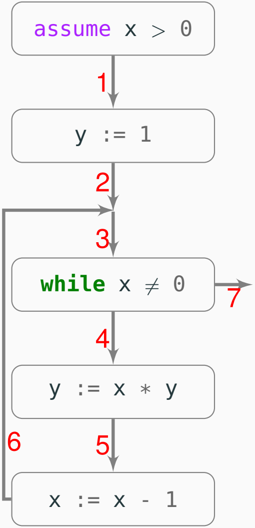


$$
\begin{aligned}
C_1 &= \quad \{x = m, y = n \mid m > 0\} \\
C_2 &= \quad \{x = m, y = 1 \mid m > 0\} \\
C_3 &= \quad \{x = m, y = 1 \mid m > 0\}
\end{aligned}
$$

$$
\begin{aligned}
C_4 &= \quad \{x = m, y = 1 \mid m > 0\} \\
C_5 &= \quad \{x = m, y = m \mid m > 0\} \\
C_6 &= \quad \{x = m - 1, y = m \mid m > 0\} \\
C_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
C_1 &= \{s \mid s(x) > 0\} \\
C_2 &= C_1 \cap \{s \mid s(y) = 1\} \\
C_3 &= C_2 \cup C_6 \\
C_4 &= C_3 \cap \{s \mid s(x) \neq 0)\} \\
C_5 &= \{s[y \mapsto s(x) \cdot s(y)] \mid s \in C_4\} \\
C_6 &= \{s[x \mapsto s(x) - 1] \mid s \in C_5\} \\
C_7 &= C_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

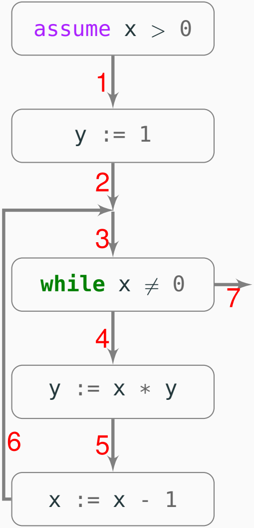
Fixed point concrete computation: example



$$
\begin{aligned}
C_1 &= \quad \{x = m, y = n \mid m > 0\}\\
C_2 &= \quad \{x = m, y = 1 \mid m > 0\}\\
C_3 &= \quad \{x = m, y = 1 \mid m > 0\}\\
&\qquad \cup \{x = m - 1, y = m \mid m > 0\}\\
C_4 &= \quad \{x = m, y = 1 \mid m > 0\}\\
C_5 &= \quad \{x = m, y = m \mid m > 0\}\\
C_6 &= \quad \{x = m - 1, y = m \mid m > 0\}\\
C_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
C_1 &= \{s \mid s(x) > 0\}\\
C_2 &= C_1 \cap \{s \mid s(y) = 1\}\\
C_3 &= C_2 \cup C_6\\
C_4 &= C_3 \cap \{s \mid s(x) \neq 0)\}\\
C_5 &= \{s[y \mapsto s(x) \cdot s(y)] \mid s \in C_4\}\\
C_6 &= \{s[x \mapsto s(x) - 1] \mid s \in C_5\}\\
C_7 &= C_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

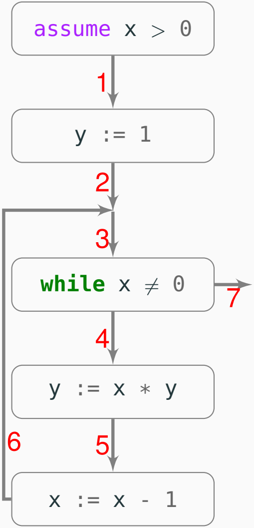
Fixed point concrete computation: example



$$
\begin{aligned}
C_1 &= \quad \{x = m, y = n \mid m > 0\} \\
C_2 &= \quad \{x = m, y = 1 \mid m > 0\} \\
C_3 &= \quad \{x = m, y = 1 \mid m > 0\} \\
&\quad\ \cup \{x = m - 1, y = m \mid m > 0\} \\
C_4 &= \quad \{x = m, y = 1 \mid m > 0\} \\
C_5 &= \quad \{x = m, y = m \mid m > 0\} \\
C_6 &= \quad \{x = m - 1, y = m \mid m > 0\} \\
C_7 &= \quad \{x = 0, y = 1 \mid m > 0\}
\end{aligned}
$$

$$
\begin{aligned}
C_1 &= \{s \mid s(x) > 0\} \\
C_2 &= C_1 \cap \{s \mid s(y) = 1\} \\
C_3 &= C_2 \cup C_6 \\
C_4 &= C_3 \cap \{s \mid s(x) \neq 0)\} \\
C_5 &= \{s[y \mapsto s(x) \cdot s(y)] \mid s \in C_4\} \\
C_6 &= \{s[x \mapsto s(x) - 1] \mid s \in C_5\} \\
C_7 &= C_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

Fixed point concrete computation: example



$$
\begin{aligned}
C_1 &= \quad \{x = m, y = n \mid m > 0\} \\
C_2 &= \quad \{x = m, y = 1 \mid m > 0\} \\
C_3 &= \quad \{x = m, y = 1 \mid m > 0\} \\
&\quad\ \cup \{x = m - 1, y = m \mid m > 0\} \\
C_4 &= \quad \{x = m, y = 1 \mid m > 0\} \\
C_5 &= \quad \{x = m, y = m \mid m > 0\} \\
C_6 &= \quad \{x = m - 1, y = m \mid m > 0\} \\
C_7 &= \quad \{x = 0, y = 1 \mid m > 0\}
\end{aligned}
$$

and so on...

$$
\begin{aligned}
C_1 &= \{s \mid s(x) > 0\} \\
C_2 &= C_1 \cap \{s \mid s(y) = 1\} \\
C_3 &= C_2 \cup C_6 \\
C_4 &= C_3 \cap \{s \mid s(x) \neq 0)\} \\
C_5 &= \{s[y \mapsto s(x) \cdot s(y)] \mid s \in C_4\} \\
C_6 &= \{s[x \mapsto s(x) - 1] \mid s \in C_5\} \\
C_7 &= C_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

Sign semantics

The sign semantics A is an abstract semantics that only keeps track of the sign of integer variables, giving the set of possible abstract states at every edge label:

$$A \colon \mathit{Labels} \to \wp(\underbrace{\mathit{Vars} \to \mathit{Sign}}_{\mathit{AbstractState}})$$

Domain $\mathit{Sign} = \{\top, +, 0, -, \bot\}$ is a finite set ordered according to $\leq$:

$$\begin{tikzcd}
& \top & \\
- \arrow[ur] & 0 \arrow[u] & + \arrow[ul] \\
& \bot \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

- $\top$ represents all integers
- $+$ represents all positive integers
- 0 represents the singleton $\{0\}$
- $-$ represents all negative integers
- $\top$ represents the empty set

The poset $\langle \mathit{Sign}, \leq \rangle$ is a

Sign semantics

The sign semantics A is an abstract semantics that only keeps track of the sign of integer variables, giving the set of possible abstract states at every edge label:

$$A\colon \mathit{Labels} \to \wp(\underbrace{\mathit{Vars} \to \mathit{Sign}}_{\mathit{AbstractState}})$$

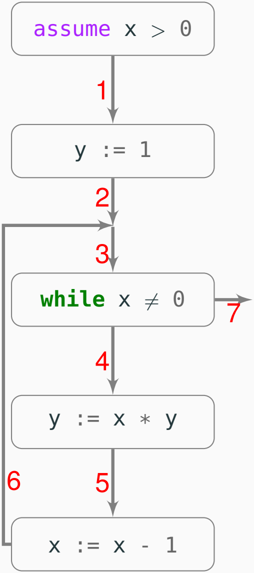
Domain $\mathit{Sign} = \{\top, +, 0, -, \bot\}$ is a finite set ordered according to $\leq$:

$$\begin{tikzcd}
 & \top & \\
- \arrow[ur] & 0 \arrow[u] & + \arrow[ul] \\
 & \bot \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

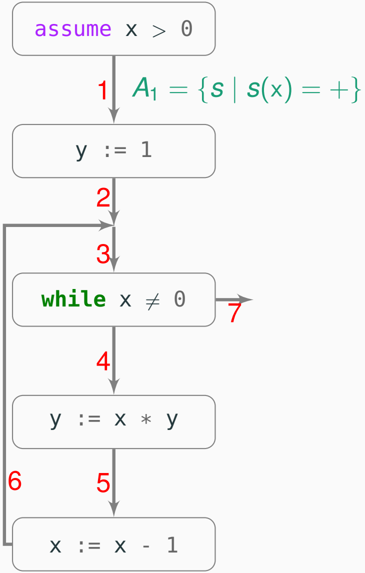
- $\top$ represents all integers
- $+$ represents all positive integers
- 0 represents the singleton $\{0\}$
- $-$ represents all negative integers
- $\top$ represents the empty set

The poset $\langle \mathit{Sign}, \leq \rangle$ is a complete lattice.

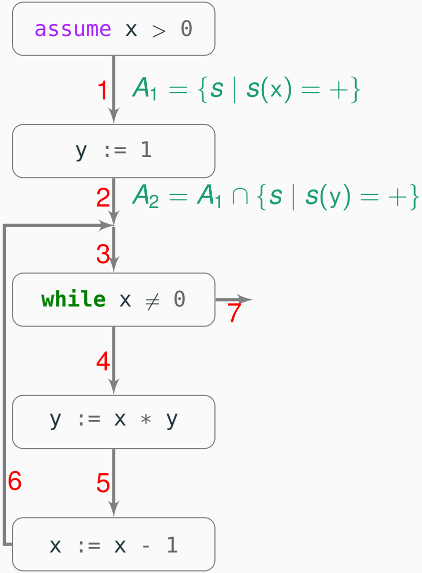
Sign semantics: example



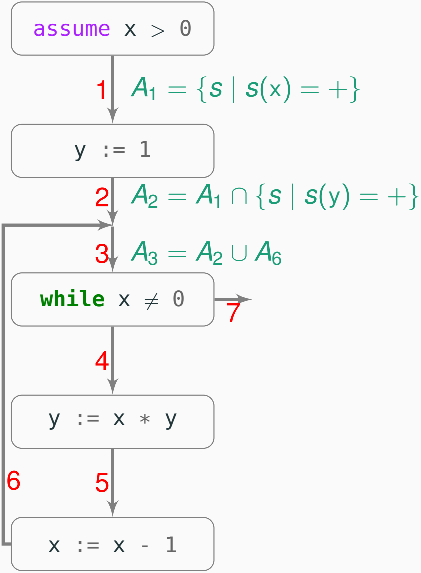
Sign semantics: example



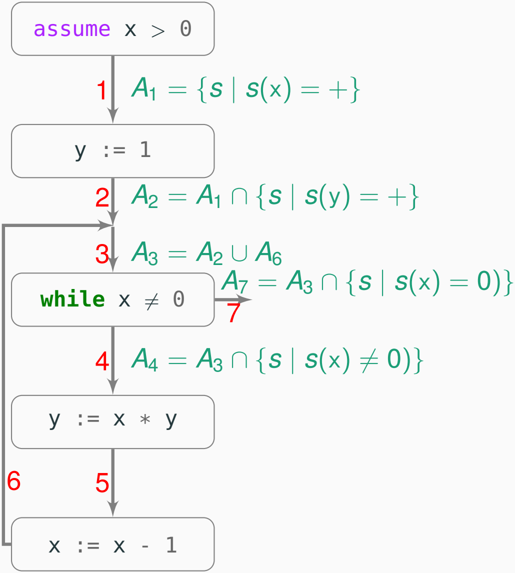
Sign semantics: example



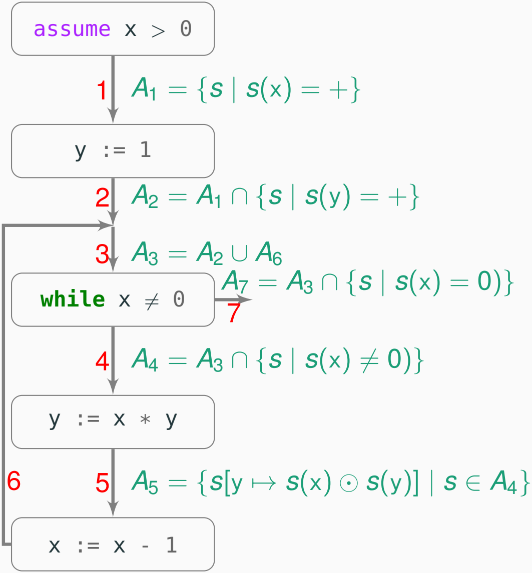
Sign semantics: example



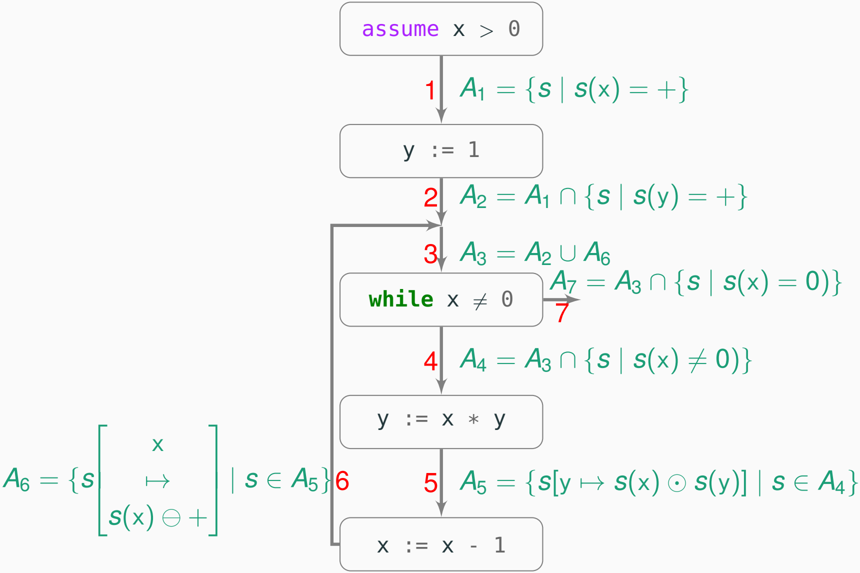
Sign semantics: example



Sign semantics: example



Sign semantics: example



Sign semantics: equation solving

The sign semantics gives a set of equations that are structurally identical to those of the collecting semantics – except that variables range over *Sign* in the sign semantics, and hence we need to express arithmetic operations $\cdot$ and $-$ as operations $\odot$ and $\ominus$ over the abstract domain.

$$
\begin{aligned}
A_1 &= \{s \mid s(\mathrm{x}) = +\} \\
A_2 &= A_1 \cap \{s \mid s(\mathrm{y}) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
A_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \odot s(\mathrm{y})] \mid s \in A_4\} \\
A_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

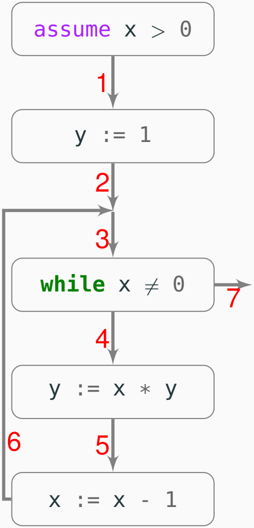
Sign semantics: equation solving

The sign semantics gives a set of equations that are structurally identical to those of the collecting semantics – except that variables range over *Sign* in the sign semantics, and hence we need to express arithmetic operations $\cdot$ and $-$ as operations $\odot$ and $\ominus$ over the abstract domain.

$$
\begin{aligned}
A_1 &= \{s \mid s(\mathrm{x}) = +\} \\
A_2 &= A_1 \cap \{s \mid s(\mathrm{y}) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
A_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \odot s(\mathrm{y})] \mid s \in A_4\} \\
A_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

Since these equations still satisfy the conditions of Tarski's fixed point theorem, we can compute the abstract semantics by evaluating the equations starting from $\{\} \times \cdots \times \{\}$ until we reach a fixed point.

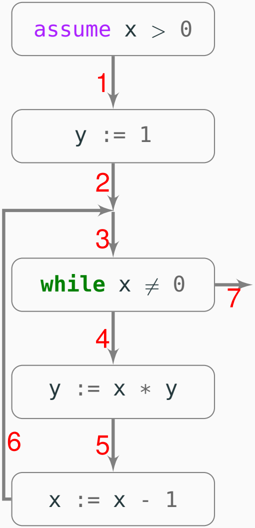
Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \\
A_2 &= \\
A_3 &= \\
A_4 &= \\
A_5 &= \\
A_6 &= \\
A_7 &=
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= \{s \mid s(\mathrm{x}) = +\} \\
A_2 &= A_1 \cap \{s \mid s(\mathrm{y}) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
A_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \odot s(\mathrm{y})] \mid s \in A_4\} \\
A_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

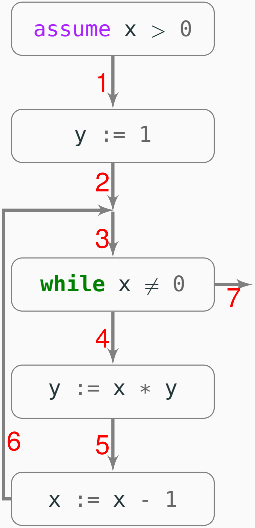
Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \{\}\\
A_2 &= \{\}\\
A_3 &= \{\}\\
A_4 &= \{\}\\
A_5 &= \{\}\\
A_6 &= \{\}\\
A_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= \{s \mid s(\mathrm{x}) = +\}\\
A_2 &= A_1 \cap \{s \mid s(\mathrm{y}) = +\}\\
A_3 &= A_2 \cup A_6\\
A_4 &= A_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\}\\
A_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \odot s(\mathrm{y})] \mid s \in A_4\}\\
A_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) \ominus +] \mid s \in A_5\}\\
A_7 &= A_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

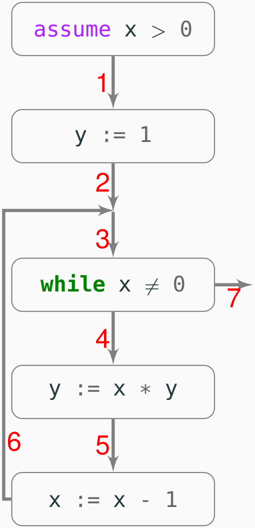
Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \quad \{x = +, y = \top\}\\
A_2 &= \{\}\\
A_3 &= \{\}\\
A_4 &= \{\}\\
A_5 &= \{\}\\
A_6 &= \{\}\\
A_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= \{s \mid s(x) = +\}\\
A_2 &= A_1 \cap \{s \mid s(y) = +\}\\
A_3 &= A_2 \cup A_6\\
A_4 &= A_3 \cap \{s \mid s(x) \neq 0)\}\\
A_5 &= \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\}\\
A_6 &= \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\}\\
A_7 &= A_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

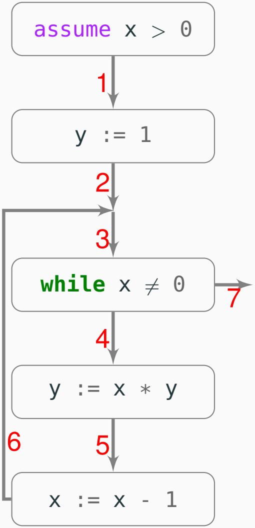
Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \quad \{x = +, y = \top\} \\
A_2 &= \quad \{x = +, y = +\} \\
A_3 &= \{\} \\
A_4 &= \{\} \\
A_5 &= \{\} \\
A_6 &= \{\} \\
A_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= \{s \mid s(x) = +\} \\
A_2 &= A_1 \cap \{s \mid s(y) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(x) \neq 0)\} \\
A_5 &= \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\} \\
A_6 &= \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

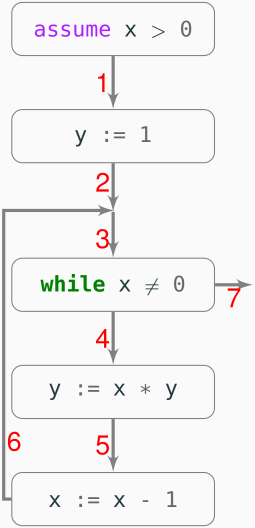
Fixed point abstract computation: example



$$A_1 = \quad \{x = +, y = \top\}$$
$$A_2 = \quad \{x = +, y = +\}$$
$$A_3 = \quad \{x = +, y = +\}$$
$$A_4 = \{\}$$
$$A_5 = \{\}$$
$$A_6 = \{\}$$
$$A_7 = \{\}$$

$$A_1 = \{s \mid s(x) = +\}$$
$$A_2 = A_1 \cap \{s \mid s(y) = +\}$$
$$A_3 = A_2 \cup A_6$$
$$A_4 = A_3 \cap \{s \mid s(x) \neq 0)\}$$
$$A_5 = \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\}$$
$$A_6 = \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\}$$
$$A_7 = A_3 \cap \{s \mid s(x) = 0)\}$$

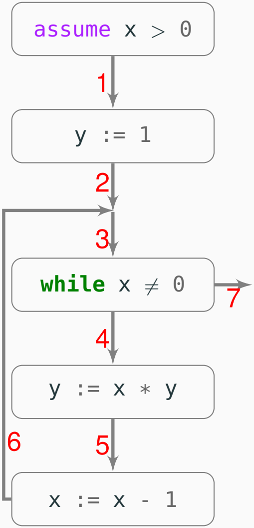
Fixed point abstract computation: example



$$\begin{aligned}
A_1 &= \quad \{x = +, y = \top\} \\
A_2 &= \quad \{x = +, y = +\} \\
A_3 &= \quad \{x = +, y = +\} \\
A_4 &= \quad \{x = +, y = +\} \\
A_5 &= \{\} \\
A_6 &= \{\} \\
A_7 &= \{\}
\end{aligned}$$

$$\begin{aligned}
A_1 &= \{s \mid s(x) = +\} \\
A_2 &= A_1 \cap \{s \mid s(y) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(x) \neq 0)\} \\
A_5 &= \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\} \\
A_6 &= \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}$$

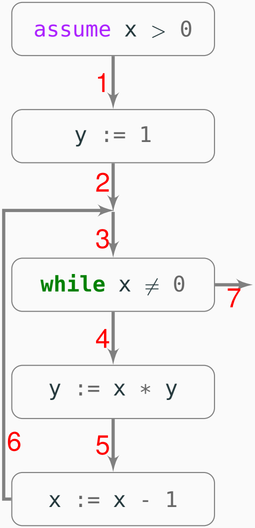
Fixed point abstract computation: example



$$A_1 = \{x = +, y = \top\}$$
$$A_2 = \{x = +, y = +\}$$
$$A_3 = \{x = +, y = +\}$$
$$A_4 = \{x = +, y = +\}$$
$$A_5 = \{x = +, y = +\}$$
$$A_6 = \{\}$$
$$A_7 = \{\}$$

$$A_1 = \{s \mid s(x) = +\}$$
$$A_2 = A_1 \cap \{s \mid s(y) = +\}$$
$$A_3 = A_2 \cup A_6$$
$$A_4 = A_3 \cap \{s \mid s(x) \neq 0)\}$$
$$A_5 = \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\}$$
$$A_6 = \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\}$$
$$A_7 = A_3 \cap \{s \mid s(x) = 0)\}$$

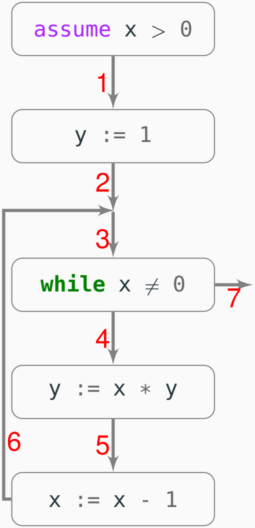
Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \quad \{x = +, y = \top\} \\
A_2 &= \quad \{x = +, y = +\} \\
A_3 &= \quad \{x = +, y = +\} \\
A_4 &= \quad \{x = +, y = +\} \\
A_5 &= \quad \{x = +, y = +\} \\
A_6 &= \quad \{x \in \{0, +\}, y = +\} \\
A_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= \{s \mid s(x) = +\} \\
A_2 &= A_1 \cap \{s \mid s(y) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(x) \neq 0)\} \\
A_5 &= \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\} \\
A_6 &= \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

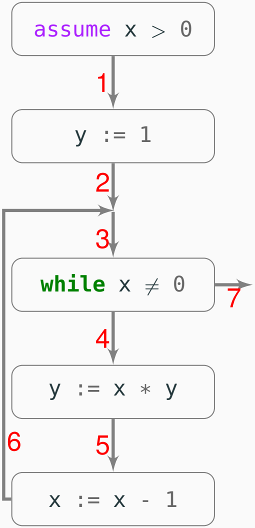
Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \quad \{x = +, y = \top\}\\
A_2 &= \quad \{x = +, y = +\}\\
A_3 &= \quad \{x \in \{0, +\}, y = +\}\\
A_4 &= \quad \{x = +, y = +\}\\
A_5 &= \quad \{x = +, y = +\}\\
A_6 &= \quad \{x \in \{0, +\}, y = +\}\\
A_7 &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= \{s \mid s(x) = +\}\\
A_2 &= A_1 \cap \{s \mid s(y) = +\}\\
A_3 &= A_2 \cup A_6\\
A_4 &= A_3 \cap \{s \mid s(x) \neq 0)\}\\
A_5 &= \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\}\\
A_6 &= \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\}\\
A_7 &= A_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

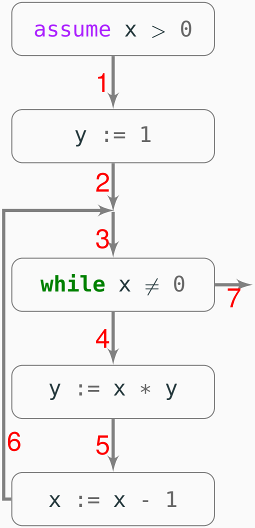
Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \quad \{x = +, y = \top\} \\
A_2 &= \quad \{x = +, y = +\} \\
A_3 &= \quad \{x \in \{0, +\}, y = +\} \\
A_4 &= \quad \{x = +, y = +\} \\
A_5 &= \quad \{x = +, y = +\} \\
A_6 &= \quad \{x \in \{0, +\}, y = +\} \\
A_7 &= \quad \{x = 0, y = +\}
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= \{s \mid s(x) = +\} \\
A_2 &= A_1 \cap \{s \mid s(y) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(x) \neq 0)\} \\
A_5 &= \{s[y \mapsto s(x) \odot s(y)] \mid s \in A_4\} \\
A_6 &= \{s[x \mapsto s(x) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(x) = 0)\}
\end{aligned}
$$

Fixed point abstract computation: example



$$
\begin{aligned}
A_1 &= \quad \{\mathrm{x} = +, \mathrm{y} = \top\} \\
A_2 &= \quad \{\mathrm{x} = +, \mathrm{y} = +\} \\
A_3 &= \quad \{\mathrm{x} \in \{0, +\}, \mathrm{y} = +\} \\
A_4 &= \quad \{\mathrm{x} = +, \mathrm{y} = +\} \\
A_5 &= \quad \{\mathrm{x} = +, \mathrm{y} = +\} \\
A_6 &= \quad \{\mathrm{x} \in \{0, +\}, \mathrm{y} = +\} \\
A_7 &= \quad \{\mathrm{x} = 0, \mathrm{y} = +\}
\end{aligned}
$$

fixed point!

$$
\begin{aligned}
A_1 &= \{s \mid s(\mathrm{x}) = +\} \\
A_2 &= A_1 \cap \{s \mid s(\mathrm{y}) = +\} \\
A_3 &= A_2 \cup A_6 \\
A_4 &= A_3 \cap \{s \mid s(\mathrm{x}) \neq 0)\} \\
A_5 &= \{s[\mathrm{y} \mapsto s(\mathrm{x}) \odot s(\mathrm{y})] \mid s \in A_4\} \\
A_6 &= \{s[\mathrm{x} \mapsto s(\mathrm{x}) \ominus +] \mid s \in A_5\} \\
A_7 &= A_3 \cap \{s \mid s(\mathrm{x}) = 0)\}
\end{aligned}
$$

Abstract interpretation

Correctness

Very simple integer expressions

To illustrate in detail how abstract interpretation supports reasoning about the correctness (soundness) of abstract computations, let us focus on a simple example: integer expressions.

Initially, we only consider the <u>product</u> as possible operation.

Very simple integer expressions

To illustrate in detail how abstract interpretation supports reasoning about the correctness (soundness) of abstract computations, let us focus on a simple example: integer expressions.

Initially, we only consider the product as possible operation.

Syntax of very simple integer expressions VE:

$$
\begin{aligned}
VE &\ni n && \text{for } n \in \mathbb{Z} \\
VE &\ni e_1 \times e_2 && \text{for } e_1, e_2 \in VE
\end{aligned}
$$

Very simple integer expressions

To illustrate in detail how abstract interpretation supports reasoning about the correctness (soundness) of abstract computations, let us focus on a simple example: integer expressions.

Initially, we only consider the product as possible operation.

Syntax of very simple integer expressions VE:

$$VE \ni n \qquad \text{for } n \in \mathbb{Z}$$

$$VE \ni e_1 \times e_2 \qquad \text{for } e_1, e_2 \in VE$$

only operation: product

Very simple integer expressions

To illustrate in detail how abstract interpretation supports reasoning about the correctness (soundness) of abstract computations, let us focus on a simple example: integer expressions.

Initially, we only consider the product as possible operation.

Syntax of very simple integer expressions *VE*:

$$
\begin{aligned}
VE &\ni n && \text{for } n \in \mathbb{Z} \\
VE &\ni e_1 \times e_2 && \text{for } e_1, e_2 \in VE
\end{aligned}
$$

only operation: product

Let us define a concrete semantics (to evaluate the integer value of any expression) and an abstract semantics (to evaluate the sign of any expression).

Expressions: concrete semantics

The concrete semantics C assigns integer values to expressions:

$$C\colon \mathit{VE} \to \mathit{State} \qquad\qquad \text{where } \mathit{State} = \mathbb{Z}$$

Expressions: concrete semantics

The concrete semantics C assigns integer values to expressions:

$$C\colon VE \to State \qquad\qquad \text{where } State = \mathbb{Z}$$

The definition of C is straightforward.

$$C[n] = n$$
$$C[e_1 \times e_2] = C[e_1] \cdot C[e_2]$$

For notational clarity, we will use square brackets to define the evaluation of semantics.

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon VE \to AbstractState \qquad \text{where } AbstractState = Sign$$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A \colon VE \to AbstractState \qquad \text{where } AbstractState = Sign$$

The definition of A:

$$A[n] = \mathrm{sign}(n) \qquad \text{where } \mathrm{sign}(n) = \begin{cases} + & n > 0 \\ 0 & n = 0 \\ - & n < 0 \end{cases}$$

$$A[e_1 \times e_2] = A[e_1] \otimes A[e_2]$$

is based on properties of how product and sign are related.

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon VE \to AbstractState \qquad \text{where } AbstractState = Sign$$

The definition of A:

$$A[n] = \mathrm{sign}(n) \qquad \text{where } \mathrm{sign}(n) = \begin{cases} + & n > 0 \\ 0 & n = 0 \\ - & n < 0 \end{cases}$$

$$A[e_1 \times e_2] = A[e_1] \otimes A[e_2]$$

is based on properties of how product and sign are related.

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon VE \to AbstractState \qquad \text{where } AbstractState = Sign$$

The definition of A:

$$A[n] = \mathrm{sign}(n) \qquad \text{where } \mathrm{sign}(n) = \begin{cases} + & n > 0 \\ 0 & n = 0 \\ - & n < 0 \end{cases}$$

$$A[e_1 \times e_2] = A[e_1] \otimes A[e_2]$$

is based on properties of how product and sign are related.

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

For example: $A[-3 \times 2 \times -5] =$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon VE \to AbstractState \qquad \text{where } AbstractState = Sign$$

The definition of A:

$$A[n] = \mathrm{sign}(n) \qquad \text{where } \mathrm{sign}(n) = \begin{cases} + & n > 0 \\ 0 & n = 0 \\ - & n < 0 \end{cases}$$

$$A[e_1 \times e_2] = A[e_1] \otimes A[e_2]$$

is based on properties of how product and sign are related.

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

For example: $A[-3 \times 2 \times -5] = +$

Representation function

We want to check that the abstract semantics is sound – that is, it correctly represents the sign of the concrete semantics.

To this end we frame the two semantics using the notation of abstract interpretation.

Representation function

We want to check that the abstract semantics is sound – that is, it correctly represents the sign of the concrete semantics.

To this end we frame the two semantics using the notation of abstract interpretation.

The representation function β links each concrete value to an abstract value:

$$\beta : \mathit{State} \to \mathit{AbstractState} \qquad\qquad \text{that is: } \mathbb{Z} \to \mathit{Sign}$$

Representation function

We want to check that the abstract semantics is sound – that is, it correctly represents the sign of the concrete semantics.

To this end we frame the two semantics using the notation of abstract interpretation.

The representation function β links each concrete value to an abstract value:

$$\beta : \mathit{State} \to \mathit{AbstractState} \qquad\qquad \text{that is: } \mathbb{Z} \to \mathit{Sign}$$

For very simple integer expressions β is sign:

$$\beta(n) = \begin{cases} + & n > 0 \\ 0 & n = 0 \\ - & n < 0 \end{cases}$$

Concretization function

We want to check that the abstract semantics is sound – that is, it correctly represents the sign of the concrete semantics.

To this end we frame the two semantics using the notation of abstract interpretation.

The concretization function γ links each abstract value to the possible concrete values it may represent:

$$\gamma : \mathit{AbstractState} \to \wp(\mathit{State}) \qquad \text{that is: } \mathit{Sign} \to \wp(\mathbb{Z})$$

Concretization function

We want to check that the abstract semantics is sound – that is, it correctly represents the sign of the concrete semantics.

To this end we frame the two semantics using the notation of abstract interpretation.

The concretization function γ links each abstract value to the possible concrete values it may represent:

$$\gamma\colon \mathit{AbstractState} \to \wp(\mathit{State}) \qquad \text{that is: } \mathit{Sign} \to \wp(\mathbb{Z})$$

For very simple integer expressions γ identifies subsets of $\mathbb{Z}$:

$$\gamma(s) = \begin{cases} \{n \in \mathbb{Z} \mid n > 0\} & s = + \\ \{0\} & s = 0 \\ \{n \in \mathbb{Z} \mid n < 0\} & s = - \end{cases}$$

Basic soundness condition

We have two semantics and the concretization function:

$$\begin{aligned} C &: VE \to State \\ A &: VE \to AbstractState \\ \gamma &: AbstractState \to \wp(State) \end{aligned} \qquad\qquad \begin{aligned} C &: VE \to \mathbb{Z} \\ A &: VE \to Sign \\ \gamma &: Sign \to \wp(\mathbb{Z}) \end{aligned}$$

Basic soundness condition

We have two semantics and the concretization function:

$$C\colon VE \to State \qquad\qquad C\colon VE \to \mathbb{Z}$$
$$A\colon VE \to AbstractState \qquad\qquad A\colon VE \to Sign$$
$$\gamma\colon AbstractState \to \wp(State) \qquad\qquad \gamma\colon Sign \to \wp(\mathbb{Z})$$

Soundness requires that the concrete semantics is compatible with the abstract semantics. Formally, $C[e]$ should give one of the possible concretizations of $A[e]$ – as in an over-approximation.

$$C[e] \in \gamma(A[e]) \qquad\qquad \text{for all } e \in VE$$

Basic soundness condition

We have two semantics and the concretization function:

$$C\colon VE \to State \qquad\qquad C\colon VE \to \mathbb{Z}$$

$$A\colon VE \to AbstractState \qquad\qquad A\colon VE \to Sign$$

$$\gamma\colon AbstractState \to \wp(State) \qquad\qquad \gamma\colon Sign \to \wp(\mathbb{Z})$$

Soundness requires that the concrete semantics is compatible with the abstract semantics. Formally, $C[e]$ should give one of the possible concretizations of $A[e]$ – as in an <u>over-approximation</u>.

$$C[e] \in \gamma(A[e]) \qquad\qquad \text{for all } e \in VE$$

For example:

$$A[-3 \times 2 \times -5] = + \qquad\qquad C[-3 \times 2 \times -5] = 30$$

Basic soundness condition

We have two semantics and the concretization function:

$$C\colon VE \to State \qquad C\colon VE \to \mathbb{Z}$$
$$A\colon VE \to AbstractState \qquad A\colon VE \to Sign$$
$$\gamma\colon AbstractState \to \wp(State) \qquad \gamma\colon Sign \to \wp(\mathbb{Z})$$

Soundness requires that the concrete semantics is compatible with the abstract semantics. Formally, $C[e]$ should give one of the possible concretizations of $A[e]$ – as in an <u>over-approximation</u>.

$$C[e] \in \gamma(A[e]) \qquad \text{for all } e \in VE$$

For example:

$$A[-3 \times 2 \times -5] = + \qquad C[-3 \times 2 \times -5] = 30$$
$$\{n \in \mathbb{Z} \mid n > 0\} = \gamma(+) \qquad \{n \in \mathbb{Z} \mid n > 0\} \ni 30$$

Simple integer expressions

To see a more interesting example, let's add the sum as possible operation between integers:

Simple integer expressions

To see a more interesting example, let's add the sum as possible operation between integers:

Syntax of simple integer expressions SE:

$$SE \ni n \qquad \text{for } n \in \mathbb{Z}$$

$$SE \ni e_1 \times e_2 \,,\; e_1 + e_2 \,,\; -e_1 \qquad \text{for } e_1, e_2 \in VE$$

unary minus

Once again, let us define a concrete semantics and an abstract semantics.

Expressions: concrete semantics

The concrete semantics C assigns integer values to expressions:

$$C : SE \rightarrow State \qquad\qquad \text{where } State = \mathbb{Z}$$

Expressions: concrete semantics

The concrete semantics C assigns integer values to expressions:

$$C\colon SE \to State \qquad\qquad \text{where } State = \mathbb{Z}$$

The definition of C is straightforward.

$$\begin{aligned} C[n] &= n \\ C[e_1 \times e_2] &= C[e_1] \cdot C[e_2] \\ C[e_1 + e_2] &= C[e_1] + C[e_2] \\ C[-e] &= -C[e] \end{aligned}$$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon \mathit{SE} \to \mathit{AbstractState} \qquad \text{where } \mathit{AbstractState} = \mathit{Sign}$$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon SE \to AbstractState \qquad \text{where } AbstractState = Sign$$

$$\begin{aligned} A[n] &= \mathrm{sign}(n) \\ A[e_1 \times e_2] &= A[e_1] \otimes A[e_2] \\ A[e_1 + e_2] &= A[e_1] \oplus A[e_2] \\ A[-e] &= \ominus A[e] \end{aligned}$$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon SE \to AbstractState \qquad \text{where } AbstractState = Sign$$

$$\begin{aligned}
A[n] &= \mathrm{sign}(n)\\
A[e_1 \times e_2] &= A[e_1] \otimes A[e_2]\\
A[e_1 + e_2] &= A[e_1] \oplus A[e_2]\\
A[-e] &= \ominus A[e]
\end{aligned}$$

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon SE \to AbstractState \qquad \text{where } AbstractState = Sign$$

$$\begin{aligned} A[n] &= \mathrm{sign}(n) \\ A[e_1 \times e_2] &= A[e_1] \otimes A[e_2] \\ A[e_1 + e_2] &= A[e_1] \oplus A[e_2] \\ A[-e] &= \ominus A[e] \end{aligned}$$

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

$\ominus$	
$-$	$+$
0	0
$+$	$-$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon SE \to \mathit{AbstractState} \qquad \text{where } \mathit{AbstractState} = \mathit{Sign}$$

$$
\begin{aligned}
A[n] &= \mathrm{sign}(n) \\
A[e_1 \times e_2] &= A[e_1] \otimes A[e_2] \\
A[e_1 + e_2] &= A[e_1] \oplus A[e_2] \\
A[-e] &= \ominus A[e]
\end{aligned}
$$

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

$\ominus$	
$-$	$+$
0	0
$+$	$-$

$\oplus$	$-$	0	$+$
$-$	$-$	$-$	?
0	$-$	0	$+$
$+$	?	$+$	$+$

Expressions: abstract semantics

The abstract semantics A assigns to expressions values in the sign domain:

$$A\colon SE \to AbstractState \qquad \text{where } AbstractState = Sign$$

$$A[n] = \mathrm{sign}(n)$$
$$A[e_1 \times e_2] = A[e_1] \otimes A[e_2]$$
$$A[e_1 + e_2] = A[e_1] \oplus A[e_2]$$
$$A[-e] = \ominus A[e]$$

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

$\ominus$	
$-$	$+$
0	0
$+$	$-$

$\oplus$	$-$	0	$+$
$-$	$-$	$-$	?
0	$-$	0	$+$
$+$	?	$+$	$+$

The abstract domain $\{+, 0, 0\}$ is not closed under the interpretation of addition $\oplus$.

Extending the abstract domain

To ensure that the abstract domain is closed under $\oplus$ we include value $\top$ (top), corresponding to “any value”. When the abstract value is $\top$ it means that we have no information about the sign.

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

Extending the abstract domain

To ensure that the abstract domain is closed under $\oplus$ we include value $\top$ (top), corresponding to "any value". When the abstract value is $\top$ it means that we have no information about the sign.

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

To have a complete lattice, let's also add to the abstract domain value $\bot$ (bottom), corresponding to "no value" (the empty set).

$A\colon \mathit{SE} \to \mathit{AbstractState}$ where $\mathit{AbstractState} = \mathit{Sign} = \{-, 0, +, \top, \bot\}$

Extending the abstract domain

To ensure that the abstract domain is closed under $\oplus$ we include value $\top$ (top), corresponding to "any value". When the abstract value is $\top$ it means that we have no information about the sign.

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

To have a complete lattice, let's also add to the abstract domain value $\perp$ (bottom), corresponding to "no value" (the empty set).

$A\colon \mathit{SE} \to \mathit{AbstractState}$ where $\mathit{AbstractState} = \mathit{Sign} = \{-, 0, +, \top, \perp\}$

$\otimes$	$-$	0	$+$	$\top$
$-$	$+$	0	$-$	$\top$
0	0	0	0	0
$+$	$-$	0	$+$	$\top$
$\top$	$\top$	0	$\top$	$\top$

Extending the abstract domain

To ensure that the abstract domain is closed under $\oplus$ we include value $\top$ (top), corresponding to "any value". When the abstract value is $\top$ it means that we have no information about the sign.

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

To have a complete lattice, let's also add to the abstract domain value $\perp$ (bottom), corresponding to "no value" (the empty set).

$A\colon \mathit{SE} \to \mathit{AbstractState} \quad$ where $\mathit{AbstractState} = \mathit{Sign} = \{-, 0, +, \top, \perp\}$

$\otimes$	$-$	0	$+$	$\top$
$-$	$+$	0	$-$	$\top$
0	0	0	0	0
$+$	$-$	0	$+$	$\top$
$\top$	$\top$	0	$\top$	$\top$

$\ominus$	
$-$	$+$
0	0
$+$	$-$
$\top$	$\top$

Extending the abstract domain

To ensure that the abstract domain is closed under $\oplus$ we include value $\top$ (top), corresponding to "any value". When the abstract value is $\top$ it means that we have no information about the sign.

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

To have a complete lattice, let's also add to the abstract domain value $\bot$ (bottom), corresponding to "no value" (the empty set).

$A\colon SE \to AbstractState$ where $AbstractState = Sign = \{-, 0, +, \top, \bot\}$

$\otimes$	$-$	0	$+$	$\top$
$-$	$+$	0	$-$	$\top$
0	0	0	0	0
$+$	$-$	0	$+$	$\top$
$\top$	$\top$	0	$\top$	$\top$

$\ominus$	
$-$	$+$
0	0
$+$	$-$
$\top$	$\top$

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

Extending the abstract domain

To ensure that the abstract domain is closed under $\oplus$ we include value $\top$ (top), corresponding to "any value". When the abstract value is $\top$ it means that we have no information about the sign.

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

To have a complete lattice, let's also add to the abstract domain value $\bot$ (bottom), corresponding to "no value" (the empty set).

$A\colon \mathit{SE} \to \mathit{AbstractState}$ where $\mathit{AbstractState} = \mathit{Sign} = \{-, 0, +, \top, \bot\}$

$\otimes$	$-$	0	$+$	$\top$
$-$	$+$	0	$-$	$\top$
0	0	0	0	0
$+$	$-$	0	$+$	$\top$
$\top$	$\top$	0	$\top$	$\top$

$\ominus$	
$-$	$+$
0	0
$+$	$-$
$\top$	$\top$

The definition of abstract operations $\otimes$, $\ominus$, and $\oplus$ applied to $\bot$ does not matter, since this value will never appear in a specific abstract computation.

Concretization function for *Sign*

We extend the concretization function γ to *Sign*

$$\gamma : \mathit{AbstractState} \to \wp(\mathit{State}) \qquad \text{that is: } \mathit{Sign} \to \wp(\mathbb{Z})$$

For simple integer expressions γ identifies subsets of $\mathbb{Z}$:

$$\gamma(s) = \begin{cases} \{n \in \mathbb{Z} \mid n > 0\} & s = + \\ \{0\} & s = 0 \\ \{n \in \mathbb{Z} \mid n < 0\} & s = - \\ \mathbb{Z} & s = \top \\ \{\} & s = \bot \end{cases}$$

Concretization function for *Sign*

We extend the concretization function γ to *Sign*

$$\gamma\colon \mathit{AbstractState} \to \wp(\mathit{State}) \qquad \text{that is: } \mathit{Sign} \to \wp(\mathbb{Z})$$

For simple integer expressions γ identifies subsets of $\mathbb{Z}$:

$$\gamma(s) = \begin{cases} \{n \in \mathbb{Z} \mid n > 0\} & s = + \\ \{0\} & s = 0 \\ \{n \in \mathbb{Z} \mid n < 0\} & s = - \\ \mathbb{Z} & s = \top \\ \{\} & s = \bot \end{cases}$$

We can see that $\langle \mathit{Sign}, \leq \rangle$ is a partial order induced by

$$a \leq b \quad \text{iff} \quad \gamma(a) \subseteq \gamma(b)$$

$$\begin{tikzcd} & \top & \\ - \arrow[ur] & 0 \arrow[u] & + \arrow[ul] \\ & \bot \arrow[ul] \arrow[u] \arrow[ur] & \end{tikzcd}$$

Abstract interpretation: the framework

To define a static analysis in the framework of abstract interpretation, we start from the concrete domain C:

1. Define an abstract domain A as a poset $\langle A, \sqsubseteq \rangle$ that must be a complete lattice
2. Define a representation function $\beta \colon C \to A$ that maps each concrete value to its "best" abstract value
3. The concretization function $\gamma \colon A \to \wp(C)$ can then be defined as

$$\gamma(a) \; = \; \{c \in C \mid \beta(c) \sqsubseteq a\}$$

4. The abstraction function $\alpha \colon \wp(C) \to A$ can then be defined as

$$\alpha(C) \; = \; \bigsqcup \{\beta(c) \mid c \in C\}$$

Abstract interpretation of simple integer expressions

Concrete domain	Abstract domain	Representation function
$C = \mathbb{Z}$	$\langle A, \sqsubseteq \rangle = \langle Sign, \leq \rangle$	$\beta = \mathsf{sign}$

Abstract interpretation of simple integer expressions

Concrete domain	Abstract domain	Representation function
$C = \mathbb{Z}$	$\langle A, \sqsubseteq \rangle = \langle Sign, \leq \rangle$	$\beta = \text{sign}$

Concretization function:

$$\gamma(a) = \{c \in \mathbb{Z} \mid \text{sign}(c) \leq a\} = \begin{cases} \{c \in \mathbb{Z} \mid c > 0\} & a = + \\ \{0\} & a = 0 \\ \{c \in \mathbb{Z} \mid c < 0\} & a = - \\ \mathbb{Z} & a = \top \\ \{\} & a = \bot \end{cases}$$

Abstract interpretation of simple integer expressions

Concrete domain	Abstract domain	Representation function
$C = \mathbb{Z}$	$\langle A, \sqsubseteq \rangle = \langle Sign, \leq \rangle$	$\beta = \text{sign}$

Concretization function:

$$\gamma(a) = \{c \in \mathbb{Z} \mid \text{sign}(c) \leq a\} = \begin{cases} \{c \in \mathbb{Z} \mid c > 0\} & a = + \\ \{0\} & a = 0 \\ \{c \in \mathbb{Z} \mid c < 0\} & a = - \\ \mathbb{Z} & a = \top \\ \{\} & a = \bot \end{cases}$$

Abstraction function ($C \subseteq C$):

$$\alpha(C) = \bigsqcup \{\beta(c) \mid c \in C\} = \begin{cases} + & C = \{1, 2, 3\} \\ \top & C = \{0, 1, 2\} \\ \top & C = \{-1, 1\} \\ \ldots \end{cases}$$

Galois connections

The concretization and abstraction functions have the following properties by construction:

monotonicity α and γ are monotonic functions

Galois connection: α and γ satisfy

$$C \subseteq \gamma(\alpha(C)) \qquad \text{for all } C \in \wp(C)$$

$$a \sqsupseteq \alpha(\gamma(a)) \qquad \text{for all } a \in A$$

Under these conditions, α and γ over their respective domains are said to form a Galois connection.

Galois connection: γ and α map between posets C and A in a way that γ and α are "almost inverses" of each other.

Galois connections

The concretization and abstraction functions have the following properties by construction:

monotonicity α and γ are monotonic functions

Galois connection: α and γ satisfy

$$C \subseteq \gamma(\alpha(C)) \qquad \text{for all } C \in \wp(C)$$
$$a \sqsupseteq \alpha(\gamma(a)) \qquad \text{for all } a \in A$$

Under these conditions, α and γ over their respective domains are said to form a Galois connection.

> Galois connection: γ and α map between posets C and A in a way that γ and α are "almost inverses" of each other.

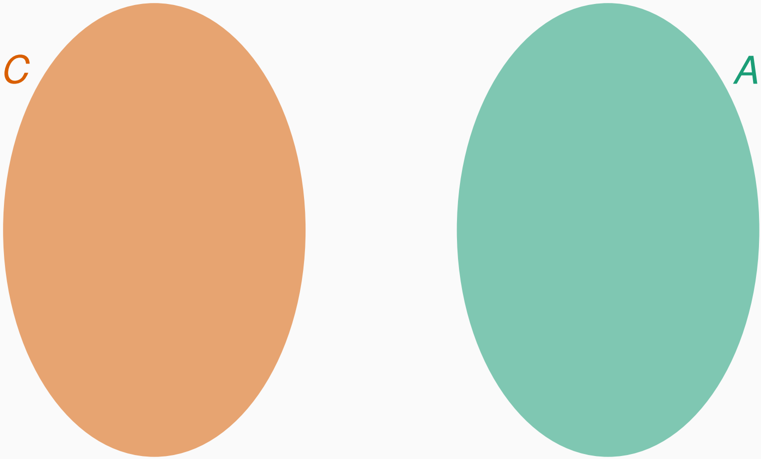
Galois connections capture the notion of correctness: the abstraction $\alpha(C)$ is a superset (over-approximation) of the concrete semantics.

Galois connections

$$C \subseteq \gamma(\alpha(C)) \qquad \text{for all } C \in \wp(C)$$

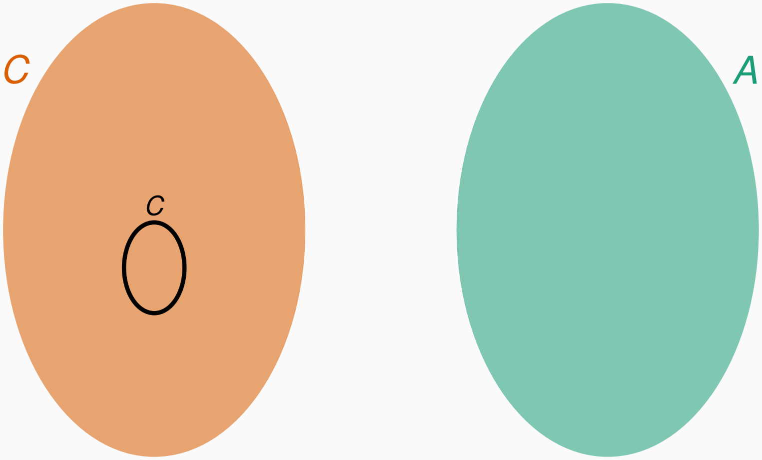
Galois connections

$C \subseteq \gamma(\alpha(C))$ for all $C \in \wp(C)$



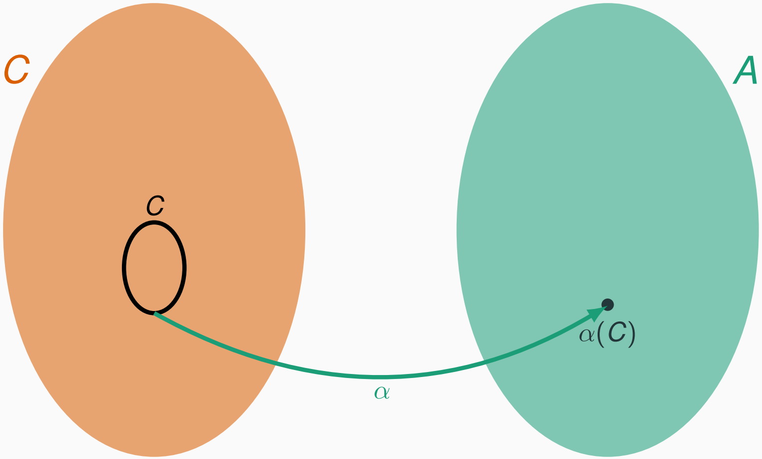
Galois connections

$C \subseteq \gamma(\alpha(C))$ for all $C \in \wp(C)$



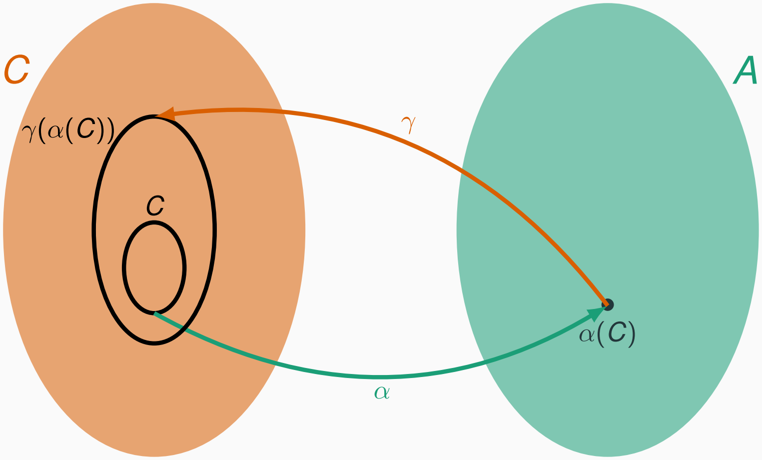
Galois connections

$C \subseteq \gamma(\alpha(C))$ for all $C \in \wp(C)$



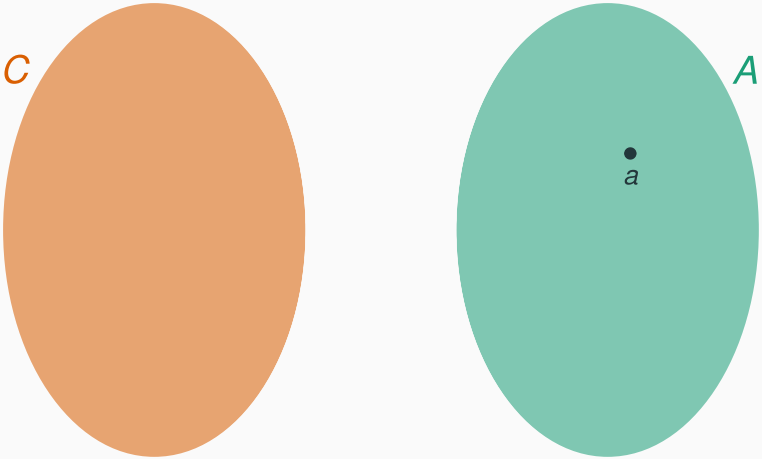
Galois connections

$C \subseteq \gamma(\alpha(C)) \qquad \text{for all } C \in \wp(C)$



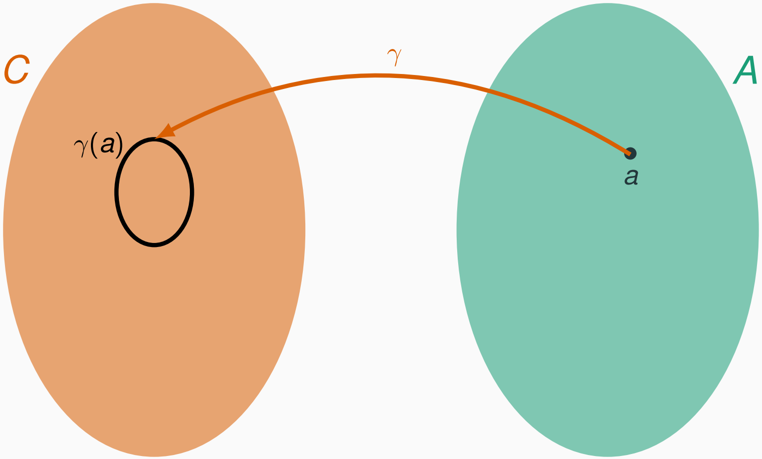
Galois connections

$a \sqsupseteq \alpha(\gamma(a))$ for all $a \in A$



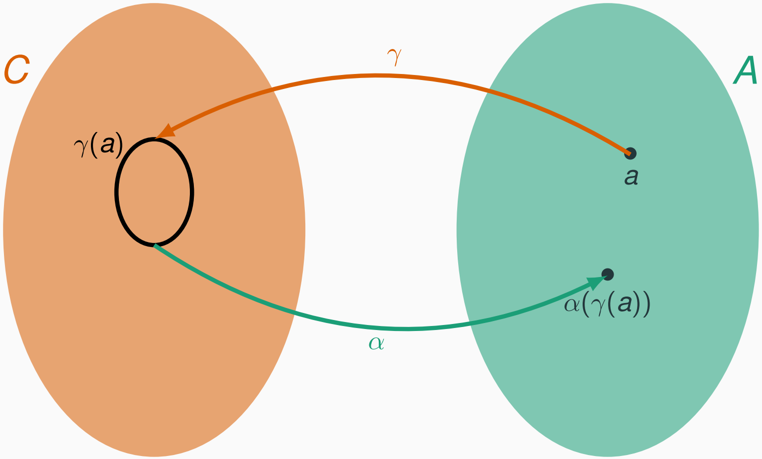
Galois connections

$a \sqsupseteq \alpha(\gamma(a))$ for all $a \in A$



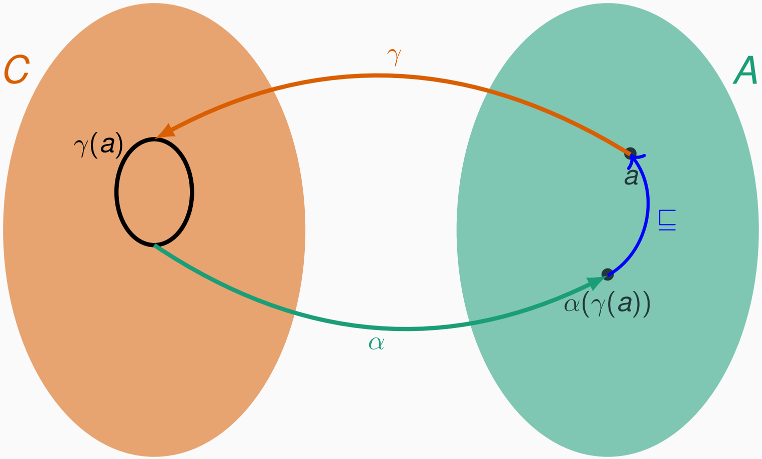
Galois connections

$a \sqsupseteq \alpha(\gamma(a))$ for all $a \in A$



Galois connections

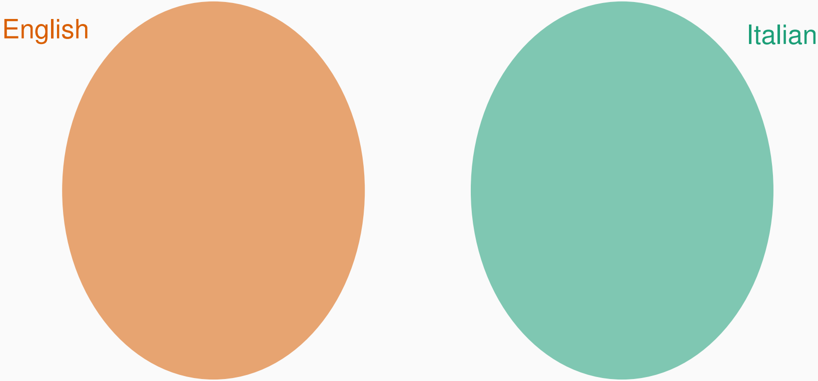
$a \sqsupseteq \alpha(\gamma(a))$ for all $a \in A$



Dictionaries

Bilingual dictionaries behave somewhat like Galois connections:

$$e \in \mathit{english}(\mathit{italian}(\{e\})) \qquad \text{for all } e \in \text{English}$$

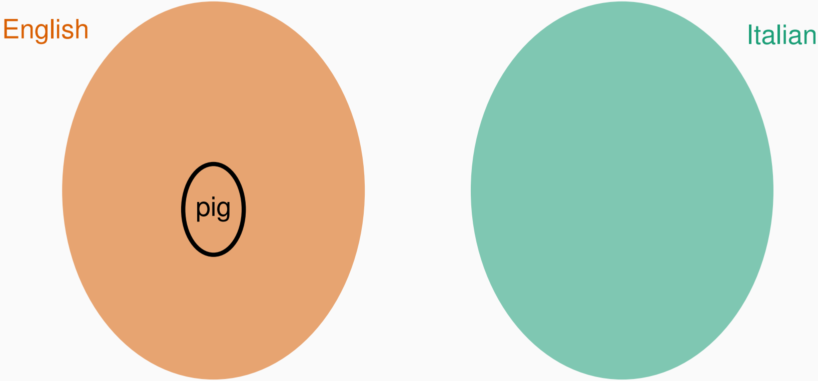


The analogy is not perfect though: *italian* gives in general more than one translation of each word. See also Bertrand Meyer's blog.

Dictionaries

Bilingual dictionaries behave somewhat like Galois connections:

$$e \in \textit{english}(\textit{italian}(\{e\})) \qquad \text{for all } e \in \text{English}$$

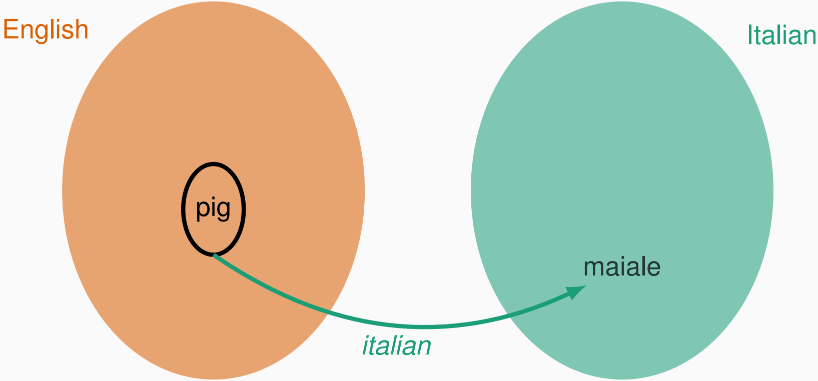


The analogy is not perfect though: *italian* gives in general more than one translation of each word. See also Bertrand Meyer's blog.

Dictionaries

Bilingual dictionaries behave somewhat like Galois connections:

$$e \in \mathit{english}(\mathit{italian}(\{e\})) \qquad \text{for all } e \in \text{English}$$

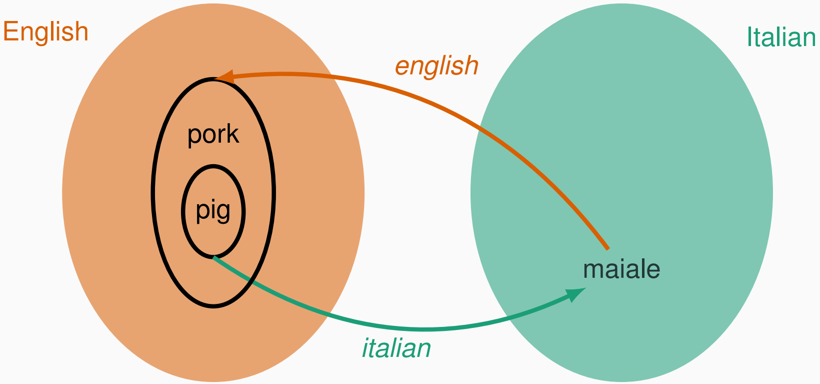


The analogy is not perfect though: *italian* gives in general more than one translation of each word. See also Bertrand Meyer's blog.

Dictionaries

Bilingual dictionaries behave somewhat like Galois connections:

$$e \in \mathit{english}(\mathit{italian}(\{e\})) \qquad \text{for all } e \in \text{English}$$



The analogy is not perfect though: *italian* gives in general more than one translation of each word. See also Bertrand Meyer's blog.

Galois insertions

The concretization and abstraction functions have the following properties by construction:

monotonicity α and γ are monotonic functions

Galois connection: α and γ form a Galois connection

$$C \subseteq \gamma(\alpha(C)) \qquad \text{for all } C \in \wp(C)$$
$$a \sqsupseteq \alpha(\gamma(a)) \qquad \text{for all } a \in A$$

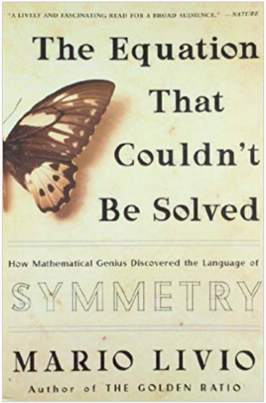
When the following stronger property holds:

$$a = \alpha(\gamma(a)) \qquad \text{for all } a \in A$$

we have a Galois insertion: the abstraction is defined in a way that there is no "redundancy" in A to describe C.

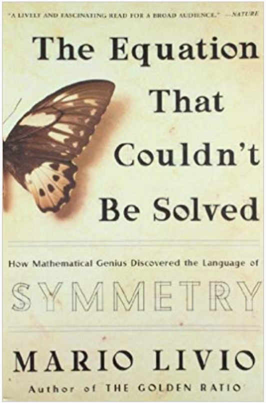
Évariste Galois

Évariste Galois



Évariste Galois

Évariste Galois
(1811–1832)



Induced operations

Once we have α and γ that form a Galois connection, we can induce abstract operations from concrete operations.

$$\begin{tikzcd}
A \arrow[rr, "\mathit{op} = \alpha \circ \mathbf{op} \circ \gamma"] & & A \arrow[d, "\gamma"] \\
C \arrow[u, "\alpha"] \arrow[rr, "\mathbf{op}"'] & & C
\end{tikzcd}$$

Induced operations

Once we have α and γ that form a Galois connection, we can induce abstract operations from concrete operations.

$$\begin{tikzcd}
\mathsf{A} \arrow[r, "\mathit{op} = \alpha \circ \mathbf{op} \circ \gamma"] & \mathsf{A} \arrow[d, "\gamma"] \\
\mathsf{C} \arrow[u, "\alpha"] \arrow[r, "\mathbf{op}"'] & \mathsf{C}
\end{tikzcd}$$

Induced abstract operation $\mathit{op} = \alpha \circ \mathbf{op} \circ \gamma$ is the most precise abstraction of **op** <u>by construction</u>.

Induced operations

Once we have α and γ that form a Galois connection, we can induce abstract operations from concrete operations.

$$\begin{tikzcd}
A \arrow[r, "\mathit{op} = \alpha \circ \mathbf{op} \circ \gamma"] & A \arrow[d, "\gamma"] \\
C \arrow[u, "\alpha"] \arrow[r, "\mathbf{op}"'] & C
\end{tikzcd}$$

function composition

Induced abstract operation $\mathit{op} = \alpha \circ \mathbf{op} \circ \gamma$ is
the most precise abstraction of **op** by construction.

$\mathit{op} : A \to Abs$

$\mathbf{op} : \wp(C) \to \wp(C)$

Induced operations

Once we have α and γ that form a Galois connection, we can induce abstract operations from concrete operations.

$$\begin{tikzcd}
A \arrow[r, "\mathit{op} = \alpha \circ \mathbf{op} \circ \gamma"] & A \arrow[d, "\gamma"] \\
C \arrow[u, "\alpha"] \arrow[r, "\mathbf{op}"'] & C
\end{tikzcd}$$

function composition

Induced abstract operation $\mathit{op} = \alpha \circ \mathbf{op} \circ \gamma$ is
the most precise abstraction of **op** by construction.

$\mathit{op} \colon A \to \mathit{Abs}$

$\mathbf{op} \colon \wp(C) \to \wp(C)$

If the induced op is not computable, we can use any approximation $\mathit{op}^\sharp$ such that $\mathit{op}(a) \sqsubseteq \mathit{op}^\sharp(a)$ for all $a \in A$.

Induced operations: example

In our running example of simple expressions, we can induce $\oplus$ from $+$ and γ, α – which in turn have been built from β.

First we express the concrete operation $+$ as an operation on sets of integers:

$$+ : \wp(\mathbb{Z}) \to \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$$
$$+(N_1, N_2) = \{n_1 + n_2 \mid n_1 \in N_1, n_2 \in N_2\}$$

Then we induce the abstract operation:

$$\oplus : \mathit{Sign} \to \mathit{Sign} \to \mathit{Sign}$$
$$\oplus(s_1, s_2) = \alpha(+(\gamma(s_1), \gamma(s_2)))$$

Induced operations: example

In our running example of simple expressions, we can induce $\oplus$ from $+$ and γ, α – which in turn have been built from β.

First we express the concrete operation $+$ as an operation on sets of integers:

$$+ \colon \wp(\mathbb{Z}) \to \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$$
$$+(N_1, N_2) = \{n_1 + n_2 \mid n_1 \in N_1, n_2 \in N_2\}$$

Then we induce the abstract operation:

$$\oplus \colon \mathit{Sign} \to \mathit{Sign} \to \mathit{Sign}$$
$$\oplus(s_1, s_2) = \alpha(+(\gamma(s_1), \gamma(s_2)))$$

For example:

$$+ \oplus - = \alpha(\gamma(+) + \gamma(-)) = \alpha(\{n > 0\} + \{n < 0\}) = \alpha(\mathbb{Z}) = \top$$
$$- \oplus 0 = \alpha(\gamma(-) + \gamma(0)) = \alpha(\{n < 0\} + \{0\}) = \alpha(\{n < 0\}) = -$$

Abstract interpretation

Widening

Range analysis

Let us look at a more informative abstract domain for integer variables: the interval domain.

empty interval

$$\mathit{Interval} \quad = \quad \{[\,]\} \cup \{[m, n] \mid m, n \in \mathbb{Z} \cup \{+\infty, -\infty\} \text{ and } m \leq n\}$$

Every element of *Interval* identifies a subset of $\mathbb{Z}$.

Range analysis

Let us look at a more informative abstract domain for integer variables: the interval domain.

empty interval

$$\mathit{Interval} \quad = \quad \{[\,]\} \cup \{[m, n] \mid m, n \in \mathbb{Z} \cup \{+\infty, -\infty\} \text{ and } m \leq n\}$$

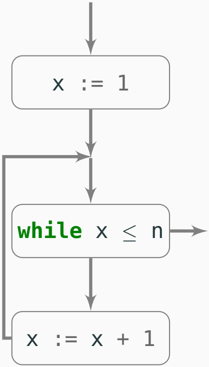
Every element of *Interval* identifies a subset of $\mathbb{Z}$.

We see that $\langle \mathit{Interval}, \sqsubseteq \rangle$ is a complete lattice:

- $\sqsubseteq$ is the subset relation $\subseteq$ between sets of integers
- $\top$ is $[-\infty, +\infty] = \mathbb{Z}$
- $\bot$ is $[\,] = \{\}$

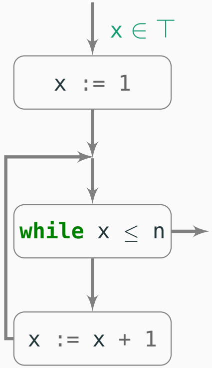
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



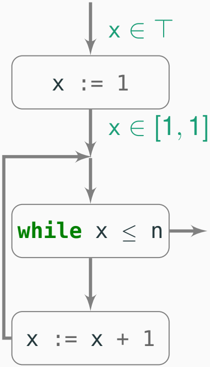
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



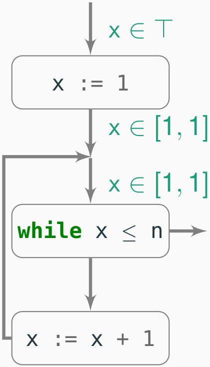
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



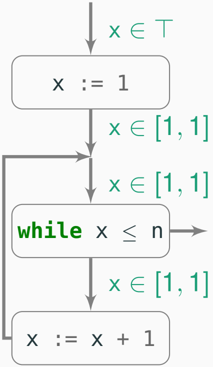
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



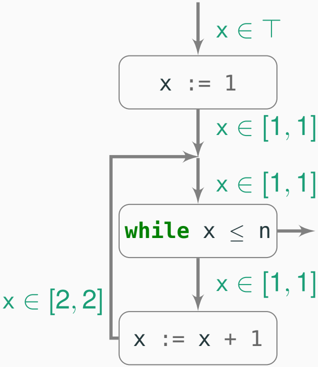
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.

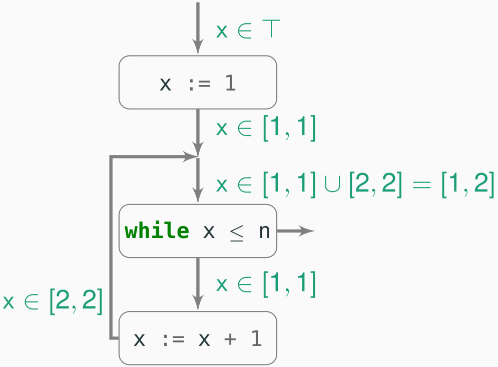


Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.

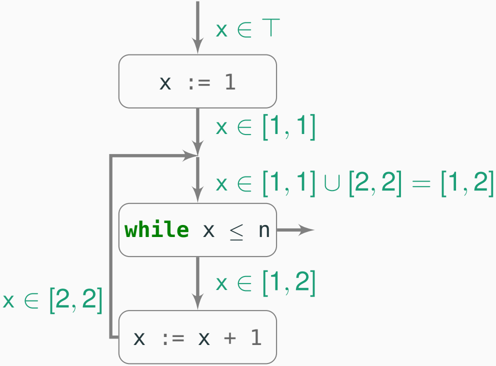


Let us try to do an abstract computation over *Interval* of a simple program.



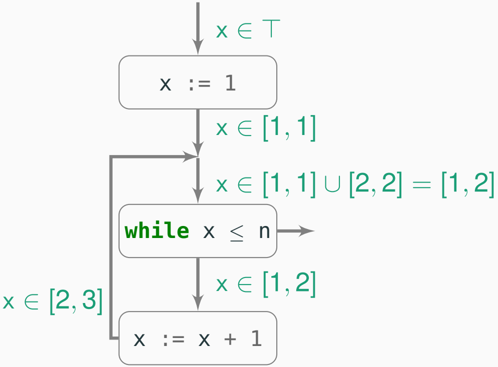
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



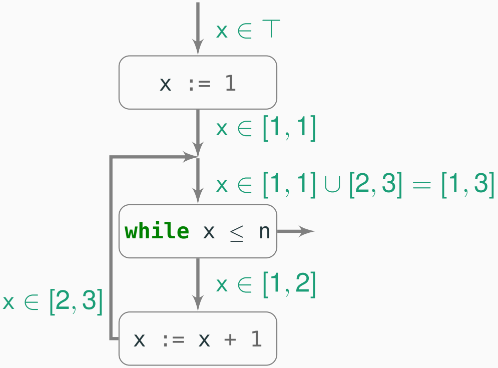
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



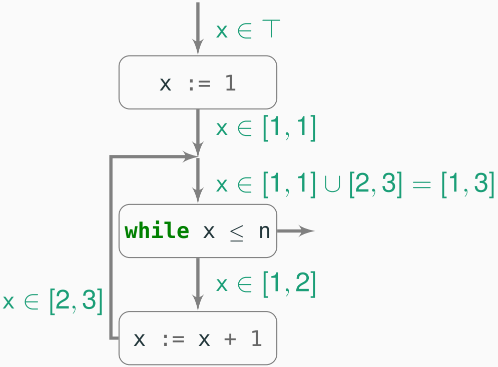
Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



Abstract computation over intervals

Let us try to do an abstract computation over *Interval* of a simple program.



Problem: the abstract state of x at loop entry does not converge:

$$[1, 1] \rightsquigarrow [1, 2] \rightsquigarrow [1, 3] \rightsquigarrow \dots$$

The analysis does not terminate – or, if it has access to static information about n, is not faster than executing the concrete computation.

Ascending chain conditions

The interval domain $\langle \mathit{Interval}, \subseteq \rangle$ is a complete lattice, and the data-flow equations are monotonic. Therefore, there exists a fixed point. The problem is that the fixed point is not computable by repeated evaluation from the least element!

Ascending chain conditions

The interval domain $\langle \mathit{Interval}, \subseteq \rangle$ is a complete lattice, and the data-flow equations are monotonic. Therefore, there exists a fixed point. The problem is that the fixed point is not computable by repeated evaluation from the least element!

A stronger condition on the abstract domain that guarantees that the fixed point is always computable is the ascending chain condition.

A complete lattice $\langle D, \sqsubseteq \rangle$ satisfies the ascending chain condition if, for every ascending sequence (chain) $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \cdots$, there exists n such that $a_n = a_{n+1} = \cdots$.

In other words, the ascending chain condition requires that every sequence of abstract values eventually stabilizes.

Ascending chain conditions

The interval domain $\langle \mathit{Interval}, \subseteq \rangle$ is a complete lattice, and the data-flow equations are monotonic. Therefore, there exists a fixed point. The problem is that the fixed point is not computable by repeated evaluation from the least element!

A stronger condition on the abstract domain that guarantees that the fixed point is always computable is the ascending chain condition.

A complete lattice $\langle D, \sqsubseteq \rangle$ satisfies the ascending chain condition if, for every ascending sequence (chain) $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \cdots$, there exists n such that $a_n = a_{n+1} = \cdots$.

In other words, the ascending chain condition requires that every sequence of abstract values eventually stabilizes.

Finite domains obviously satisfy the ascending chain condition.

Forcing termination

The interval domain does not satisfy the ascending chain condition.
To terminate, we must avoid getting stuck in the infinite chain:

$$[1,1] \rightsquigarrow [1,2] \rightsquigarrow [1,3] \rightsquigarrow \ldots$$

Forcing termination

The interval domain does not satisfy the ascending chain condition. To terminate, we must avoid getting stuck in the infinite chain:

$$[1, 1] \leadsto [1, 2] \leadsto [1, 3] \leadsto \dots$$

One trick is to forcefully terminate the chain by jumping to a larger value at some point:

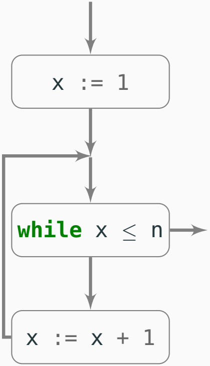
$$[1, 1] \leadsto [1, 2] \leadsto \cdots \leadsto [1, \infty]$$

To forcefully terminate the abstract computation we can replace the join operator with a widening operator ∇:

EXACT COMPUTATION	FORCED TERMINATION
$[1, 1] \sqcup [2, 2] = [1, 2]$	$[1, 1] \nabla [2, 2] = [1, +\infty]$

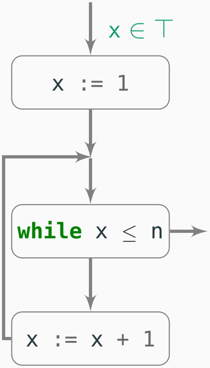
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



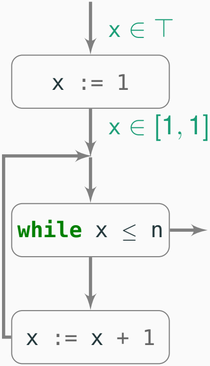
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



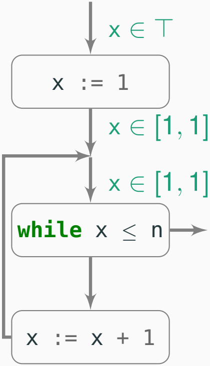
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



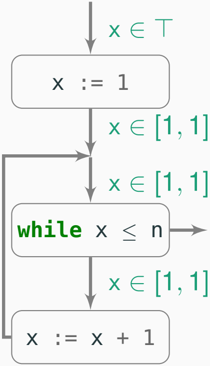
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



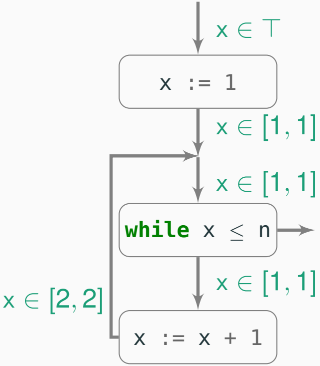
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



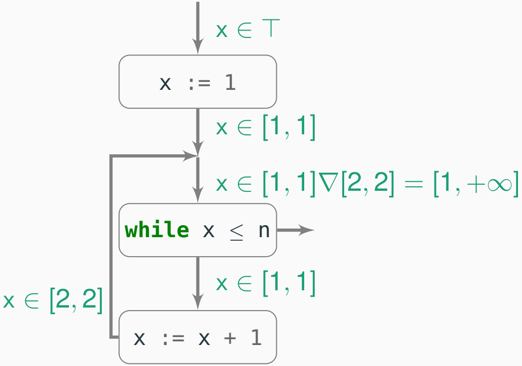
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



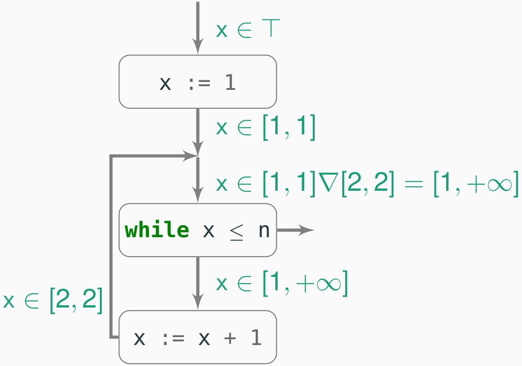
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



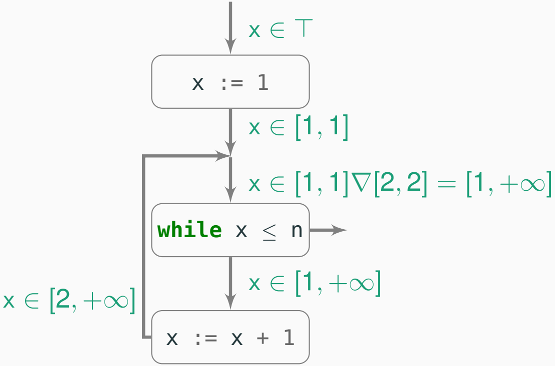
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



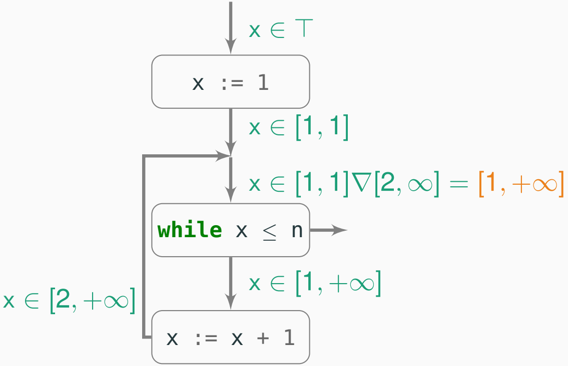
Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



Abstract computation over intervals

Using widening the abstract computation over *Interval* converges quickly but is less precise.



Widening

A widening $\nabla \colon D \times D \to D$ on a poset $\langle D, \sqsubseteq \rangle$
is a function with the properties:

upper bound: for $d_1, d_2 \in D$, $d_1 \sqsubseteq d_1 \nabla d_2$ and $d_2 \sqsubseteq d_1 \nabla d_2$

ascending chain: for all ascending chains $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \cdots$, the derived ascending chain $w_1 \sqsubseteq w_2 \sqsubseteq w_3 \sqsubseteq \cdots$

$$w_k = \begin{cases} d_1 & k = 1 \\ w_{k-1} \nabla d_k & k > 1 \end{cases}$$

eventually stabilizes

Widening

A widening $\nabla : D \times D \to D$ on a poset $\langle D, \sqsubseteq \rangle$
is a function with the properties:

upper bound: for $d_1, d_2 \in D$, $d_1 \sqsubseteq d_1 \nabla d_2$ and $d_2 \sqsubseteq d_1 \nabla d_2$

ascending chain: for all ascending chains $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \cdots$, the derived ascending chain $w_1 \sqsubseteq w_2 \sqsubseteq w_3 \sqsubseteq \cdots$

$$w_k = \begin{cases} d_1 & k = 1 \\ w_{k-1} \nabla d_k & k > 1 \end{cases}$$

eventually stabilizes

Using widening, we ensure that the abstract computation terminates – or we speed up a terminating but slow computation.

Widening

A widening $\nabla : D \times D \to D$ on a poset $\langle D, \sqsubseteq \rangle$
is a function with the properties:

upper bound: for $d_1, d_2 \in D$, $d_1 \sqsubseteq d_1 \nabla d_2$ and $d_2 \sqsubseteq d_1 \nabla d_2$

ascending chain: for all ascending chains $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \cdots$, the derived ascending chain $w_1 \sqsubseteq w_2 \sqsubseteq w_3 \sqsubseteq \cdots$

$$w_k = \begin{cases} d_1 & k = 1 \\ w_{k-1} \nabla d_k & k > 1 \end{cases}$$

eventually stabilizes

Using widening, we ensure that the abstract computation terminates – or we speed up a terminating but slow computation.

Speed is traded-off against precision: using widening we get to a fixed point but it may not be the least fixed point but only an <u>upper bound</u> on the least fixed point.

Abstract interpretation in practice

This was just a brief overview of abstract interpretation.

The abstract interpretation framework includes a vast body of research, and various techniques to support the construction of correct static analyses.

Defining a new analysis is still far from trivial, but the tools of abstract interpretation help us ensuring its correctness a priori – as opposed to defining an analysis first, and then checking its correctness as an afterthought.

Abstract interpretation in practice

This was just a brief overview of abstract interpretation.

The abstract interpretation framework includes a vast body of research, and various techniques to support the construction of correct static analyses.

Defining a new analysis is still far from trivial, but the tools of abstract interpretation help us ensuring its correctness a priori – as opposed to defining an analysis first, and then checking its correctness as an afterthought.

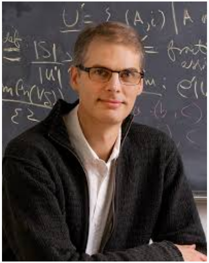
See chapter 12 of Bradley and Manna's "The calculus of computation" for an original presentation of the concepts of abstract interpretation using the notation and terminology of Hoare logic.

Type systems

What are type systems?

A type system is a <u>tractable syntactic method</u> for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

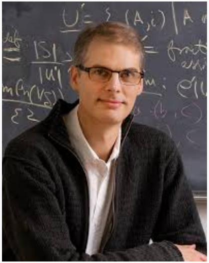
Benjamin Pierce



What are type systems?

a static analysis

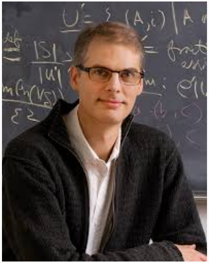
> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
>
> *Benjamin Pierce*



What are type systems?

a static analysis

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
>
> *Benjamin Pierce*



Type systems are a form of static analysis for proving the absence of certain errors based on classifying program terms according to the kinds of values they may take.

For example: which expressions are Boolean and which are integer.

Well typed programs

A type system consists of rules to check an arbitrary program term.

A program that can be checked successfully using a type system's rules is called well typed (typable).

Well typed programs

A type system consists of rules to check an arbitrary program term.

A program that can be checked successfully using a type system's rules is called well typed (typable).

Well-typed programs cannot "go wrong"

Robin Milner, 1978

In other words, a type system's rules soundly check that each type is used according to the operations and values that it permits.

Type systems

Well typedness

Expression language E

To illustrate type systems we initially focus on a very simple language E of conditional, relational, and integer arithmetic expressions:

constants

conditional expression

$$E ::= C \mid E + E \mid E \leq E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E$$

$$C ::= \texttt{true} \mid \texttt{false} \mid n \qquad \text{for } n \in \mathbb{Z}$$

Expression language E

To illustrate type systems we initially focus on a very simple language E of conditional, relational, and integer arithmetic expressions:

constants conditional expression

$$E ::= C \mid E + E \mid E \leq E \mid \texttt{if } E \texttt{ then } E \texttt{ else } E$$

$$C ::= \texttt{true} \mid \texttt{false} \mid n \qquad \text{for } n \in \mathbb{Z}$$

Even though the language is very simple, note that it can express all Boolean combinations of integer comparison expressions:

$$\neg A \triangleq \texttt{if } A \texttt{ then false else true}$$

$$A \wedge B \triangleq \texttt{if } A \texttt{ then } (\texttt{if } B \texttt{ then true else false}) \texttt{ else false}$$

$$A = B \triangleq A \leq B \wedge B \leq A$$

$$A < B \triangleq \neg(B \leq A)$$

Expression language E: semantics

The semantics $[\![\ ]\!]\colon E \to \mathbb{Z} \cup \mathbb{B}$ of language E is a set of rules to evaluate expressions.

$$\frac{n \in \mathbb{Z}}{[\![n]\!] = n} \qquad \frac{}{[\![\texttt{true}]\!] = \top} \qquad \frac{}{[\![\texttt{false}]\!] = \bot}$$

$$\frac{[\![E_1]\!] = e_1 \quad [\![E_2]\!] = e_2}{[\![E_1 + E_2]\!] = e_1 + e_2} \qquad \frac{[\![E_1]\!] = e_1 \quad [\![E_2]\!] = e_2}{[\![E_1 \leq E_2]\!] = e_1 \leq e_2}$$

$$\frac{[\![E_1]\!] = \top \quad [\![E_2]\!] = e_2 \quad [\![E_3]\!] = e_3}{[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!] = e_2} \qquad \frac{[\![E_1]\!] = \bot \quad [\![E_2]\!] = e_2 \quad [\![E_3]\!] = e_3}{[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!] = e_3}$$

Expression language E: semantics

The semantics $[\![\ ]\!]\colon E \to \mathbb{Z} \cup \mathbb{B}$ of language E is a set of rules to evaluate expressions.

$$\frac{n \in \mathbb{Z}}{[\![n]\!] = n} \qquad \frac{}{[\![\texttt{true}]\!] = \top} \qquad \frac{}{[\![\texttt{false}]\!] = \bot}$$

$$\frac{[\![E_1]\!] = e_1 \quad [\![E_2]\!] = e_2}{[\![E_1 + E_2]\!] = e_1 + e_2} \qquad \frac{[\![E_1]\!] = e_1 \quad [\![E_2]\!] = e_2}{[\![E_1 \leq E_2]\!] = e_1 \leq e_2}$$

$$\frac{[\![E_1]\!] = \top \quad [\![E_2]\!] = e_2 \quad [\![E_3]\!] = e_3}{[\![\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3]\!] = e_2} \qquad \frac{[\![E_1]\!] = \bot \quad [\![E_2]\!] = e_2 \quad [\![E_3]\!] = e_3}{[\![\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3]\!] = e_3}$$

These rules are partial because they are not applicable to every expression E. For example $[\![\texttt{true} + 4]\!]$ is undefined: if we apply the rules we get stuck at some point.

Expression language E: type system

Types provide a way to check whether an expression can be successfully evaluated without actually evaluating it. To this end, we need to distinguish between two kinds of values – two types integer and Boolean.

$$T ::= \texttt{Integer} \mid \texttt{Boolean}$$

For an expression E and a type T, $E : T$ denotes that E has type T: E certainly evaluates to a value of type T.

Expression language E: type system

Types provide a way to check whether an expression can be successfully evaluated without actually evaluating it. To this end, we need to distinguish between two kinds of values – two types integer and Boolean.

$$T ::= \texttt{Integer} \mid \texttt{Boolean}$$

For an expression E and a type T, $E : T$ denotes that E has type T: E certainly evaluates to a value of type T.

A type system is a collection of typing rules to determine the type of an arbitrary expression:

$$\frac{}{n : \texttt{Integer}} \qquad \frac{}{\texttt{true} : \texttt{Boolean}} \qquad \frac{}{\texttt{false} : \texttt{Boolean}}$$

$$\frac{E_1 : \texttt{Integer} \quad E_2 : \texttt{Integer}}{E_1 + E_2 : \texttt{Integer}} \qquad \frac{E_1 : \texttt{Integer} \quad E_2 : \texttt{Integer}}{E_1 \leq E_2 : \texttt{Boolean}}$$

$$\frac{E_1 : \texttt{Boolean} \quad E_2 : T \quad E_3 : T}{\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 : T}$$

Well typedness

An expression E is well typed (typable) if we can infer that $E : T$, for some type T, using the type system's rules.

well typed: $3 + 4 + 7 + 0$ not well typed: `true` $+ 4$

Well typedness

An expression E is well typed (typable) if we can infer that $E : T$, for some type T, using the type system's rules.

well typed: $3 + 4 + 7 + 0$ not well typed: `true` $+ 4$

Well typedness is an over-approximation (sound and imprecise) of correct program behavior.

Well typedness

An expression E is well typed (typable) if we can infer that $E : T$, for some type T, using the type system's rules.

well typed: $3 + 4 + 7 + 0$ not well typed: `true` $+ 4$

Well typedness is an over-approximation (sound and imprecise) of correct program behavior.

SOUNDNESS	INCOMPLETENESS
if E is well typed, then the evaluation of E cannot go wrong	if E is not well typed, the evaluation of E may still be successful
example: $3+4+7+0$	false positive: `if true then 3 else (true + 4)`

Well-typed programs don't get stuck

An expression E is well typed (typable) if we can infer that $E : T$, for some type T, using the type system's rules.

To ensure that a well-typed program does not get stuck, a type system has to satisfy two fundamental properties that, together, ensure safety:

progress: if E is well-typed, then either E is a value or we can take one step of evaluation

preservation: if E is well-typed, then taking one step of evaluation leads to some E' that is also well typed

Well-typed programs don't get stuck

An expression E is well typed (typable) if we can infer that $E : T$, for some type T, using the type system's rules.

To ensure that a well-typed program does not get stuck, a type system has to satisfy two fundamental properties that, together, ensure safety:

progress: if E is well-typed, then either E is a value or we can take one step of evaluation

preservation: if E is well-typed, then taking one step of evaluation leads to some E' that is also well typed

$$\text{SAFETY} = \text{PROGRESS} + \text{PRESERVATION}$$

Well-typed programs don't get stuck

An expression E is well typed (typable) if we can infer that $E : T$, for some type T, using the type system's rules.

To ensure that a well-typed program does not get stuck, a type system has to satisfy two fundamental properties that, together, ensure safety:

progress: if E is well-typed, then either E is a value or we can take one step of evaluation

preservation: if E is well-typed, then taking one step of evaluation leads to some E' that is also well typed

$$\text{SAFETY} = \text{PROGRESS} + \text{PRESERVATION}$$

We can prove by structural induction that the simple type system for E is safe.

Type systems

Type checking

Lambda language F

Let us extend E with a syntax for lambda expressions:

variable

function application

$$F ::= E \mid \mathrm{x} \mid \lambda\, \mathrm{x} : T \,.\, F \mid F F$$

function abstraction

type annotation

Lambda language F

Let us extend E with a syntax for lambda expressions:

$$F ::= E \mid \overset{\text{variable}}{\mathtt{x}} \mid \underset{\text{function abstraction}}{\boldsymbol{\lambda}}\, \mathtt{x} : \underset{\text{type annotation}}{T} \,.\, F \mid \overset{\text{function application}}{F\,F}$$

Function abstraction $\boldsymbol{\lambda}\ \mathtt{x}:\ T.F$ defines an expression F as an anonymous function of its argument $\mathtt{x}$, which has to be annotated with its type T. In addition to the integer and Boolean types, now we also have a function type $T \to T$ from any type to any type:

$$T ::= \mathtt{Integer} \mid \mathtt{Boolean} \mid T \to T$$

Lambda language F

Let us extend E with a syntax for lambda expressions:

function application

variable

$$F ::= \; E \mid \mathtt{x} \mid \boldsymbol{\lambda}\ \mathtt{x}\colon\ T\,.\,F \mid F\,F$$

function abstraction   type annotation

Function abstraction $\boldsymbol{\lambda}\ \mathtt{x}\colon\ T.F$ defines an expression F as an anonymous function of its argument x, which has to be annotated with its type T. In addition to the integer and Boolean types, now we also have a function type $T \to T$ from any type to any type:

$$T ::= \mathtt{\textbf{Integer}} \mid \mathtt{\textbf{Boolean}} \mid T \to T$$

Examples:

`(λ x: Integer . (x + x)) 4`   evaluates to

`((λ f: Integer → Integer . λ x: Integer . f x) (λ y: Integer . (y + 1))) 3`

evaluates to

Lambda language F

Let us extend E with a syntax for lambda expressions:

function application

variable

$$F ::= E \mid \texttt{x} \mid \boldsymbol{\lambda}\ \texttt{x}\!:\ T\ .\ F \mid F\,F$$

function abstraction　type annotation

Function abstraction $\boldsymbol{\lambda}$ $\texttt{x}$: $T.F$ defines an expression F as an anonymous function of its argument $\texttt{x}$, which has to be annotated with its type T. In addition to the integer and Boolean types, now we also have a function type $T \to T$ from any type to any type:

$$T ::= \textbf{Integer} \mid \textbf{Boolean} \mid T \to T$$

Examples:

```
(λ x: Integer . (x + x)) 4
```
evaluates to 8

```
((λ f: Integer → Integer . λ x: Integer . f x) (λ y: Integer . (y + 1))) 3
```
evaluates to

Lambda language F

Let us extend E with a syntax for lambda expressions:

variable

function application

$$F ::= E \mid \mathtt{x} \mid \boldsymbol{\lambda}\, \mathtt{x}\!:\, T \,.\, F \mid F\, F$$

function abstraction

type annotation

Function abstraction $\boldsymbol{\lambda}\ \mathtt{x}\!:\ T.F$ defines an expression F as an anonymous function of its argument $\mathtt{x}$, which has to be annotated with its type T. In addition to the integer and Boolean types, now we also have a function type $T \to T$ from any type to any type:

$$T ::= \textbf{Integer} \mid \textbf{Boolean} \mid T \to T$$

Examples:

`(λ x: Integer . (x + x)) 4` evaluates to 8

`((λ f: Integer → Integer . λ x: Integer . f x) (λ y: Integer . (y + 1))) 3`

evaluates to 4

Lambda language F: semantics

The semantics of language F extends E's with a rule to handle lambda expressions:

$$\frac{}{[\![(\boldsymbol{\lambda}\ \mathsf{x}\colon\ T \cdot E_1)\ E_2]\!] = [\![E_1[\mathsf{x} \mapsto E_2]]\!]}$$

Lambda language F: semantics

The semantics of language F extends E's with a rule to handle lambda expressions:

$$\frac{}{[\![(\boldsymbol{\lambda}\ \mathtt{x}\colon T \cdot E_1)\ E_2]\!] = [\![E_1[\mathtt{x} \mapsto E_2]]\!]}$$

For example:

$$\begin{aligned}
&[\![((\boldsymbol{\lambda}\ \mathtt{f}\colon \mathtt{Integer} \to \mathtt{Integer}\ .\ \boldsymbol{\lambda}\ \mathtt{x}\colon \mathtt{Integer}\ .\ \mathtt{f\ x})\ (\boldsymbol{\lambda}\ \mathtt{y}\colon \mathtt{Integer}\ .\ (\mathtt{y} + \mathtt{1})))\ 3]\!] \\
&\qquad = [\![(\boldsymbol{\lambda}\ \mathtt{x}\colon \mathtt{Integer}\ .\ (\boldsymbol{\lambda}\ \mathtt{y}\colon \mathtt{Integer}\ .\ (\mathtt{y} + \mathtt{1}))\ \mathtt{x})\ 3]\!] \\
&\qquad = [\![(\boldsymbol{\lambda}\ \mathtt{y}\colon \mathtt{Integer}\ .\ (\mathtt{y} + \mathtt{1}))\ 3]\!] \\
&\qquad = [\![3 + 1]\!] = 4
\end{aligned}$$

Lambda language F: semantics

The semantics of language F extends E's with a rule to handle lambda expressions:

$$\frac{}{[\![(\boldsymbol{\lambda}\ \mathtt{x}\colon\ T \cdot E_1)\ E_2]\!] = [\![E_1[\mathtt{x} \mapsto E_2]]\!]}$$

For example:

$$\begin{aligned}
&[\![((\boldsymbol{\lambda}\ \mathtt{f}\colon\ \mathtt{Integer} \to \mathtt{Integer}\ .\ \boldsymbol{\lambda}\ \mathtt{x}\colon\ \mathtt{Integer}\ .\ \mathtt{f}\ \mathtt{x})\ (\boldsymbol{\lambda}\ \mathtt{y}\colon\ \mathtt{Integer}\ .\ (\mathtt{y} + \mathtt{1})))\ 3]\!]\\
&\qquad = [\![(\boldsymbol{\lambda}\ \mathtt{x}\colon\ \mathtt{Integer}\ .\ (\boldsymbol{\lambda}\ \mathtt{y}\colon\ \mathtt{Integer}\ .\ (\mathtt{y} + \mathtt{1}))\ \mathtt{x})\ 3]\!]\\
&\qquad = [\![(\boldsymbol{\lambda}\ \mathtt{y}\colon\ \mathtt{Integer}\ .\ (\mathtt{y} + \mathtt{1}))\ 3]\!]\\
&\qquad = [\![3 + 1]\!] = 4
\end{aligned}$$

The semantics of F is partial, which implies that:

- we can evaluate abstractions only when they are applied
- we can evaluate variables only when they appear inside lambda expressions

A type system can enforce these rules by means of additional typing rules.

Lambda language F: type system

Function abstractions (which can be nested) introduce assumptions about the type of their arguments using type annotations. To keep track of these assumptions, the type system's rules now use an environment Γ, which is a mapping from variables to their types.

$\Gamma \vdash F : T$ "F has type T under environment Γ"

Lambda language F: type system

Function abstractions (which can be nested) introduce assumptions about the type of their arguments using type annotations. To keep track of these assumptions, the type system's rules now use an environment Γ, which is a mapping from variables to their types.

$\Gamma \vdash F : T$ "F has type T under environment Γ"

With this new notation in place, the type system for F adds the rules:

$$\frac{\Gamma(\mathrm{x}) = T}{\Gamma \vdash \mathrm{x} : T} \qquad \frac{\Gamma \cup [\mathrm{x} \mapsto T_1] \vdash F : T_2}{\Gamma \vdash \boldsymbol{\lambda}\ \mathrm{x} : T_1\ .\ F : T_1 \to T_2} \qquad \frac{\Gamma \vdash F_1 : T_1 \to T_2 \quad \Gamma \vdash F_2 : T_1}{\Gamma \vdash F_1 F_2 : T_2}$$

For simplicity, we assume that variables are <u>uniquely named</u> throughout a whole expression.

Inverted rules

We can invert every rule of the type system, since they each refer to syntactically distinct terms. Some examples of inverted rules:

RULE	INVERSION
$\dfrac{}{\Gamma \vdash \texttt{true} : \texttt{Boolean}}$	if $\Gamma \vdash \texttt{true} : T$ then $T = \texttt{Boolean}$
$\dfrac{\Gamma \vdash F_1 : \texttt{Boolean} \quad \Gamma \vdash F_2, F_3 : T}{\Gamma \vdash \texttt{if}\ F_1\ \texttt{then}\ F_2\ \texttt{else}\ F_3 : T}$	if $\Gamma \vdash \texttt{if}\ F_1\ \texttt{then}\ F_2\ \texttt{else}\ F_3 : T$ then $\Gamma \vdash F_1 : \texttt{Boolean}$ and $\Gamma \vdash F_2, F_3 : T$
$\dfrac{\Gamma \cup [\mathrm{x} \mapsto T_1] \vdash F : T_2}{\Gamma \vdash \lambda\ \mathrm{x} : T_1\ .\ F : T_1 \to T_2}$	if $\Gamma \vdash \lambda\ \mathrm{x} : T_1\ .\ F : T$ then $T = T_1 \to T_2$ for some T_2 such that $\Gamma \cup [\mathrm{x} \mapsto T_1] \vdash F : T_2$
$\dfrac{\Gamma \vdash F_1 : T_1 \to T_2 \quad \Gamma \vdash F_2 : T_1}{\Gamma \vdash F_1\ F_2 : T_2}$	if $\Gamma \vdash F_1\ F_2 : T$ then there is some type T_1 such that $\Gamma \vdash F_1 : T_1 \to T$ and $\Gamma \vdash F_2 : T_1$

Type checking

Inverted rules lead to a recursive type checking algorithm.

```
typeOf :: Environment -> F -> T
typeOf g fexp = case fexp of
  "true"  -> Boolean
  "if" f1 "then" f2 "else" f3 -> if (typeOf g f1) == Boolean
                                    && (typeOf g f2) == (typeOf g f3)
                                 then (typeOf g f2)
                                 else error
  "lambda" x ":" t1 "." f     -> let t2 = typeOf (g + (x, t1)) f in
                                   t1 -> t2
  f1 f2                       -> let t1         = typeOf g f2
                                     (t1 -> t2) = typeOf g f1 in
                                   t2
  -- more rules...
```

An expression `f` is well typed iff `typeOf [] f` returns without errors.

Type checking

Inverted rules lead to a recursive type checking algorithm.

More idiomatically using Haskell's **`Maybe`** monad:

```
typeOf :: Environment ->F -> Maybe T
typeOf g fexp = case fexp of
  "true" -> return Boolean
  "if" f1 "then" f2 "else" f3 -> do
           t1 <- typeOf g f1
           t2 <- typeOf g f2
           t3 <- typeOf g f3
           if t1 == Boolean && t2 == t3 then return t2 else Nothing
  "lambda" x ":" t1 "." f     -> do
           t2 <- typeOf (g + (x, t1)) f
           return (t1 -> t2)
  f1 f2                       -> do
           t1 <- typeOf g f2
           t  <- typeOf g f1
           case t of
              (t1 -> t2) -> return t2
              otherwise  -> Nothing
  -- more rules...
wellTyped :: F -> Bool
wellTyped f = case typeOf [] f of
  Just _   -> True
  Nothing -> False
```

Type annotations

The type annotations used in lambda abstractions are only used for typing:

- the semantics ignores them
- inconsistent annotations will make typechecking fail (spuriously, if the system is correct)

Type annotations

The type annotations used in lambda abstractions are only used for typing:

- the semantics ignores them
- inconsistent annotations will make typechecking fail (spuriously, if the system is correct)
  - fails type checking even though it is well typed:
    ```
    (λ x: Integer . if x then 0 else 1) true
    ```
  - fails type checking and it is not well typed:
    ```
    (λ x: Integer . if x then 0 else 1) 0
    ```

Type annotations

The type annotations used in lambda abstractions are only used for typing:

- the semantics ignores them
- inconsistent annotations will make typechecking fail (spuriously, if the system is correct)
  - fails type checking even though it is well typed:
    `(λ x: Integer . if x then 0 else 1) true`
  - fails type checking and it is not well typed:
    `(λ x: Integer . if x then 0 else 1) 0`

Annotations are a trade-off between automation and expressiveness (flexibility):

- Users of the type system provide these annotations to support more expressive type checking rules
- The type checker is completely automatic given the annotations

Besides, explicit annotations are also a useful form of documentation.

Type reconstruction

An alternative to typing annotations is type reconstruction: the type checker tries to guess suitable types that make type checking pass.

Type reconstruction

An alternative to typing annotations is type reconstruction: the type checker tries to guess suitable types that make type checking pass.

An approach to type reconstruction is constraint-based typing:

- typing constraints are equations between type expressions involving type variables
- typing rules generate constraints instead of directly checking them
- an expression is well typed iff the corresponding typing constraints have a solution – providing an instantiation of type variables

Type reconstruction

An alternative to typing annotations is type reconstruction: the type checker tries to guess suitable types that make type checking pass.

An approach to type reconstruction is constraint-based typing:

- typing constraints are equations between type expressions involving type variables
- typing rules generate constraints instead of directly checking them
- an expression is well typed iff the corresponding typing constraints have a solution – providing an instantiation of type variables

$\Gamma \vdash F : T \mid C$ $\quad$ F has type T under Γ whenever constraints C are satisfied

Constraint-based type rules

Here are some examples of constraint-based type rules for F.

We use lowercase letters to denote (fresh) type variables, which we implicitly assume are always fresh to lighten the notation.

Values do not introduce any constraints:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Boolean} \mid \{\}} \qquad \frac{}{\Gamma \vdash n : \texttt{Integer} \mid \{\}}$$

Constraint-based type rules

Here are some examples of constraint-based type rules for F.

We use lowercase letters to denote (fresh) type variables, which we implicitly assume are always fresh to lighten the notation.

Conditional expressions introduce constraints about the type t_1 of the condition, and about the two branches' types t_2, t_3 which have to be equal:

$$\frac{\Gamma \vdash F_1 : t_1 \mid C_1 \quad \Gamma \vdash F_2 : t_2 \mid C_2 \quad \Gamma \vdash F_3 : t_3 \mid C_3 \qquad C = C_1 \cup C_2 \cup C_3 \cup \{t_1 = \texttt{Boolean}, t_2 = t_3\}}{\Gamma \vdash \texttt{if } F_1 \texttt{ then } F_2 \texttt{ else } F_3 : t_2 \mid C}$$

Constraint-based type rules

Here are some examples of constraint-based type rules for F.

We use lowercase letters to denote (fresh) type variables, which we implicitly assume are always fresh to lighten the notation.

Function applications introduce constraints about how the types of the applied abstraction, argument, and result are related:

$$\frac{\Gamma \vdash F_1 : t_1 \mid C_1 \quad \Gamma \vdash F_2 : t_2 \mid C_2 \qquad C = C_1 \cup C_2 \cup \{t_1 = t_2 \to t\}}{\Gamma \vdash F_1\ F_2 : t \mid C}$$

Constraint-based type rules

Here are some examples of constraint-based type rules for F.

We use lowercase letters to denote (fresh) type variables, which we implicitly assume are always fresh to lighten the notation.

Function abstractions, now without type annotations, introduce a fresh type variable t_1, which will be constrained by function applications:

$$\frac{\Gamma \cup [\mathrm{x} \mapsto t_1] \vdash F : t_2 \mid C}{\Gamma \vdash \boldsymbol{\lambda}\ \mathrm{x}\ .\ F : t_1 \rightarrow t_2 \mid C}$$

Constraint-based type rules: example

Type checking the expression `λ x . if x then 0 else 1` results in:

Constraint-based type rules: example

Type checking the expression `λ x . if x then 0 else 1` results in:

$$
\begin{aligned}
\texttt{x}:&\ t_x \\
\texttt{0}:&\ t_0 \\
\texttt{1}:&\ t_1 \\
\texttt{if x then 0 else 1}:&\ t_0 \\
\lambda\ \texttt{x . if x then 0 else 1}:&\ t_x \to t_0
\end{aligned}
$$

with constraints:

Constraint-based type rules: example

Type checking the expression `λ x . if x then 0 else 1` results in:

$$
\begin{aligned}
\texttt{x}:&\ t_x\\
\texttt{0}:&\ t_0\\
\texttt{1}:&\ t_1\\
\texttt{if x then 0 else 1}:&\ t_0\\
\lambda\ \texttt{x . if x then 0 else 1}:&\ t_x \rightarrow t_0
\end{aligned}
$$

with constraints:

$$C = \{t_x = \texttt{Boolean}, t_0 = \texttt{Integer}, t_1 = \texttt{Integer}, t_0 = t_1\}$$

which is clearly satisfiable.

Constraint-based type rules: example

Type checking the expression `(λ x . if x then 0 else 1) 3` adds the judgments:

Constraint-based type rules: example

Type checking the expression `(λ x . if x then 0 else 1) 3` adds the judgments:

`3`: t_3

`(λ x . if x then 0 else 1) 3`: t_0

with overall constraints:

Constraint-based type rules: example

Type checking the expression `(λ x . if x then 0 else 1) 3` adds the judgments:

$$\begin{aligned} \texttt{3}:&\ t_3 \\ \texttt{(λ x . if x then 0 else 1) 3}:&\ t_0 \end{aligned}$$

with overall constraints:

$$C = \left\{ \begin{array}{l} t_x = \texttt{Boolean}, t_0 = \texttt{Integer}, t_1 = \texttt{Integer}, t_0 = t_1, \\ t_3 = \texttt{Integer}, t_x \to t_0 = t_3 \to t_0 \end{array} \right\}$$

which is unsatisfiable.

Unification

The constraints generated by type reconstruction are equations with uninterpreted symbols (the <u>type variables</u>).

Such equations can be solved using the unification algorithm, which is very efficient (runs in linear time).

Type systems

More expressive type systems

Effect systems

The framework of type systems can be extended to accommodate checking more complex properties by extending the language of type annotations.

Effect systems

The framework of type systems can be extended to accommodate checking more complex properties by extending the language of type annotations.

An effect system piggybacks a standard type system to keep track of additional information about a program's behavior.

Suppose evaluating an expression may the side effects of modifying the value stored in some global variable. To analyze the side effects of evaluating a given expression:

- add the annotated function type $T_1 \xrightarrow{\mu} T_2$: the type of a function from T_1 to T_2 which, when evaluated, may modify variables in μ
- use type judgments

$\Gamma \vdash E : T \wr \mu$ — E evaluates to a value of type T under Γ; during evaluation, side effects μ may take place

Effect systems

The framework of type systems can be extended to accommodate checking more complex properties by extending the language of type annotations.

An effect system piggybacks a standard type system to keep track of additional information about a program's behavior.

Suppose evaluating an expression may the side effects of modifying the value stored in some global variable. To analyze the side effects of evaluating a given expression:

- add the annotated function type $T_1 \xrightarrow{\mu} T_2$: the type of a function from T_1 to T_2 which, when evaluated, may modify variables in μ
- use type judgments

$\Gamma \vdash E : T \wr \mu$	E evaluates to a value of type T under Γ; during evaluation, side effects μ may take place

After building an effect system with suitable rules, type reconstruction algorithms can be modified so that they also compute the side effects of each expression in a program.

Dependent types

Another extension of type systems are dependent types – types whose definition depends on some values.

(regular) type: lists of integers

dependent type: sorted lists of integers

Dependent types

Another extension of type systems are dependent types – types whose definition depends on some values.

(regular) type: lists of integers

dependent type: sorted lists of integers

Type checking programs using expressive dependent type annotations (with dependency constraints using an expressive logic) may be very complex or even undecidable – since it is essentially equivalent to deciding the validity of complex logic formulas.

For example, Coq is an interactive theorem prover whose logic is a functional language with very expressive dependent types.

Curry-Howard correspondence

The connection between constructive logic and types is deep as summarized by the Curry-Howard correspondence (also called isomorphism). The intuition is that a constructive proof of a proposition is isomorphic to the typechecking of a term.

LOGIC	TYPES
propositions	types
implication $P \Longrightarrow Q$	function type $P \to Q$
conjunction $P \wedge Q$	product type $P \times Q$
proof of proposition P	term T has type P
proposition P is provable	type P is not empty

Curry-Howard correspondence

The connection between constructive logic and types is deep as summarized by the Curry-Howard correspondence (also called isomorphism). The intuition is that a constructive proof of a proposition is isomorphic to the typechecking of a term.

LOGIC	TYPES
propositions	types
implication $P \Longrightarrow Q$	function type $P \rightarrow Q$
conjunction $P \wedge Q$	product type $P \times Q$
proof of proposition P	term T has type P
proposition P is provable	type P is not empty

Haskell Brooks Curry, after whom programming languages Haskell, Brook, and Curry are named

Static analysis in practice

Sound vs. soundy

We characterized static analysis as sound and imprecise. In practice, the trade-off between soundness and completeness is more subtle and sometimes partially compromises on soundness.

Sound vs. soundy

We characterized static analysis as sound and imprecise. In practice, the trade-off between soundness and completeness is more subtle and sometimes partially compromises on soundness.

VIEWPOINT

In Defense of Soundiness: A Manifesto

By Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, Dimitrios Vardoulakis
Communications of the ACM, February 2015, Vol. 58 No. 2, Pages 44-46
10.1145/2644805

We are not aware of a single realistic whole-program analysis tool that does not purposely make unsound choices. The reasons for such choices are engineering compromises [soundness vs. efficiency or precision trade off].

Most common language features are over-approximated. Some specific language features are under-approximated.

We introduce the term soundy for such analyses.

Whole-program analyses

Most static analyses are whole program: they model executions that traverse the overall program state crossing module boundaries.

The main challenge for whole-program analyses is scalability.

Whole-program analyses

Most static analyses are whole program: they model executions that traverse the overall program state crossing module boundaries.

The main challenge for whole-program analyses is scalability.

```
procedure set_zero(x: ref Integer): { [x] := 0 }
```

Which variables are modified by the call `set_zero(y)`?

We need to know all variables `y` may be aliased to. For this, we need to know all program executions that may reach the call – knowing the callee and the caller in isolation is not enough.

Whole-program analyses

Most static analyses are whole program: they model executions that traverse the overall program state crossing module boundaries.

The main challenge for whole-program analyses is scalability.

```
procedure set_zero(x: ref Integer): { [x] := 0 }
```

Which variables are modified by the call `set_zero(y)`?

We need to know all variables `y` may be aliased to. For this, we need to know all program executions that may reach the call – knowing the callee and the caller in isolation is not enough.

Alternatives approaches:

annotations: users provide frame specifications of each procedure – losing automation

coarse approximation: assume that every global variable may be modified by the call – losing precision

Modular analyses

Some static analysis are naturally modular: they model each procedure or module separately, and have a way of combining each module's analysis results with the others'.

The main challenge for modular analyses is precision of modular summaries.

Modular analyses

Some static analysis are naturally modular: they model each procedure or module separately, and have a way of combining each module's analysis results with the others'.

The main challenge for modular analyses is precision of modular summaries.

```
procedure unknown(in: Integer): (out: Boolean)
```

Is the call `b := unknown(c)` well typed?

We only need to know the type of `b` and `c`, and the signature of `unknown`. Knowing the callee and the caller in isolation is enough.

Modular analyses

Some static analysis are naturally modular: they model each procedure or module separately, and have a way of combining each module's analysis results with the others'.

The main challenge for modular analyses is precision of modular summaries.

```
procedure unknown(in: Integer): (out: Boolean)
```

Is the call `b := unknown(c)` well typed?

We only need to know the type of `b` and `c`, and the signature of `unknown`. Knowing the callee and the caller in isolation is enough.

Type systems are naturally modular because they summarize each program feature through its type.

Challenge: external code

Static whole-program analysis requires access to all source code that is executed. This may not be available:

- if we call a pre-compiled library
- if we use features that wrap native code calls

```java
public class HelloJNI {
   // Load native library hello.dll (Windows) or libhello.so (*nix)
   static { System.loadLibrary("hello"); }
   private native void sayHello();   // declare native method
   public static void main(String[] args)
   {  new HelloJNI().sayHello();  }  // call native method
}
```

Challenge: external code

Static whole-program analysis requires access to all source code that is executed. This may not be available:

- if we call a pre-compiled library
- if we use features that wrap native code calls

```
public class HelloJNI {
   // Load native library hello.dll (Windows) or libhello.so (*nix)
   static { System.loadLibrary("hello"); }
   private native void sayHello();   // declare native method
   public static void main(String[] args)
   {  new HelloJNI().sayHello();  }  // call native method
}
```

How to handle external code:

- unsound approximation: `sayHello()` does nothing
- imprecise approximation: `sayHello()` may modify anything
- annotations: users annotate `sayHello()` with relevant information

Practical solutions typically combine these three approaches.

Challenge: reflection

Reflection provides capabilities to modify a program at runtime. Reflection is particularly powerful in dynamic languages such as Python and Javascript.

```
eval(s)
```

This call executes the source code provided in (string) variable `s` as if it was declared at the call site.

Challenge: reflection

Reflection provides capabilities to modify a program at runtime. Reflection is particularly powerful in dynamic languages such as Python and Javascript.

```
eval(s)
```

This call executes the source code provided in (string) variable `s` as if it was declared at the call site.

How to handle reflection:

- unsound approximation: `eval` does nothing
- imprecise approximation: `eval` may modify anything
- string abstraction: the static analysis tries to keep track of the content of string variables, and uses this information to analyze `eval`'s effects with some precision

Challenge: reflection

Reflection provides capabilities to modify a program at runtime. Reflection is particularly powerful in dynamic languages such as Python and Javascript.

```
eval(s)
```

This call executes the source code provided in (string) variable `s` as if it was declared at the call site.

How to handle reflection:

- unsound approximation: `eval` does nothing
- imprecise approximation: `eval` may modify anything
- string abstraction: the static analysis tries to keep track of the content of string variables, and uses this information to analyze `eval`'s effects with some precision

The Eval that Men Do

A Large-scale Study of the Use of Eval in JavaScript Applications

Richards, Hammer, Burg, and Vitek: ECOOP 2011

Static analysis and deductive verification

Some of the challenges of static analysis (modularity and tricky language features) are challenges of deductive verification too!

Static analysis and deductive verification tend to target different trade-offs:

- static analyses target scalability and automation (that is, not user annotations)
- deductive verification uses modularity and assumes expert users who can supply complex annotations

Static analysis and deductive verification

Some of the challenges of static analysis (modularity and tricky language features) are challenges of deductive verification too!

Static analysis and deductive verification tend to target different trade-offs:

- static analyses target scalability and automation (that is, not user annotations)
- deductive verification uses modularity and assumes expert users who can supply complex annotations

Deductive verification may use some static analysis to lessen the annotation burden (for example, simple loop invariants) or to simplify what is to be proved (for example, assuming programs are well-typed).

Notable static analysis tools

Astrée checks embedded C programs (no dynamic memory allocation or recursion) for absence of runtime errors such as undefined behavior

CCC (Code Contracts Static Checker), formerly known as Clousot, is an abstract interpreter for .NET programs checking the absence of common runtime errors – relying on pre-/postconditions to achieve modularity

Frama-C is an extensible analyzer for C programs, which supports a variety of common static analyses (reaching definitions, slicing, ...) as well as deductive verification and dynamic analyses through dedicated plug-ins

Infer is static analyzer for C/C++/Objective C and Java code based on separation-logic abstractions of memory usage

Scan by Coverity is a multi-language analyzer that can detect memory errors, concurrency issues, and incorrect API usage – one of the first static analyzers that was widely applicable with good precision

Type checking tools

the **Checker Framework** extends Java's type system with custom type annotations that can be used to establish absence of bugs such as null-pointer dereferencing or string safety vulnerabilities

Type checking tools

the **Checker Framework** extends Java's type system with custom type annotations that can be used to establish absence of bugs such as null-pointer dereferencing or string safety vulnerabilities

In addition, every compiler (framework) includes type checkers and modules that perform static analyses enabling compiler optimizations.

Frameworks such as LLVM export APIs to perform and use static analyses in derived applications.

Summary

Static analysis: techniques

Static analysis is a large family of techniques for automatically establishing that a program is free from certain pre-defined erroneous behavior.

Static analysis techniques are normally based on over-approximating behavior at every program point.

soundness/completeness: sound and imprecise (finding a reasonable trade-off between number of spurious warnings and soundness)

complexity: efficient algorithms that scale up to large programs

automation: fully automated ("push button")

expressiveness: limited to fixed properties like "absence of common errors"

Static analysis: tools and practice

Static analysis tools range from the components in a compiler framework that support optimizations, to type checkers, to analyzers that detect possible runtime errors (such as undefined behavior and memory problems). Overall, static analysis is used extensively in software technology.

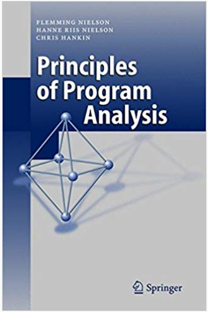
Besides its usage for compiler construction, case studies of static analysis include the safety verification of large embedded programs – such as Astrée's verification of the absence of runtime errors in Airbus control software ($>$ 130'000 lines of C code).

Main outstanding challenges:

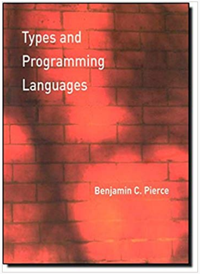
- supporting program features such as reflection and native calls without losing too much soundness or precision
- increasing flexibility and extensibility of frameworks to support checking new properties
- integrating additional information (such as specific assumptions or input constraints) when useful

Credits and further reading

This class's presentations of data-flow analysis and abstract interpretation was based on material by Sebastian Nanz (for the Software Verification course given at ETH Zurich in 2009–2015), which in turn was based on chapters in Principles of Program Analysis.



This class's presentations of type systems was adapted from Types and Programming Languages.



These slides' license