Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 6: Abstract Interpretation

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Previously

- Operational Semantics
  - Large step (Natural)
  - Small step (SOS)

How?

- Axiomatic Semantics
  - aka Hoare Logic
  - aka axiomatic (manual) verification

Why?

From verification to analysis

- Manual **program verification**
  - Verifier provides assertions
    - Loop invariants

- Automatic **program verification** (Program analysis)
  - Tool automatically synthesize assertions
    - Finds loop invariants

Manual proof for max

```
nums : array
N : int

x := 0

res := nums[0]


while x < N

    if nums[x] > res then

        res := nums[x]


    x := x + 1
```

Manual proof for max

```
nums : array
N : unsigned int
{ N≥0 }
x := 0
{ N≥0 ∧ x=0 }
res := nums[0]
{ x=0 }
Inv = { x≤N }
while x < N
    { x=k ∧ k<N }
    if nums[x] > res then
        { x=k ∧ k<N }
        res := nums[x]
        { x=k ∧ k<N }
    { x=k ∧ k<N }
    x := x + 1
    { x=k+1 ∧ k≤N }
{ x≤N ∧ x≥N }
{ x=N }
```

We only prove no buffer (array) overflow

Can we find this proof automatically?

```
nums : array
N : unsigned int
{ N≥0 }
x := 0
{ N≥0 ∧ x=0 }
res := nums[0]
{ x=0 }
Inv = { x≤N }
while x < N
    { x=k ∧ k<N }
    if nums[x] > res then
        { x=k ∧ k<N }
        res := nums[x]
        { x=k ∧ k<N }
    { x=k ∧ k<N }
    x := x + 1
    { x=k+1 ∧ k≤N }
{ x≤N ∧ x≥N }
{ x=N }
```

Observation: predicates in proof have the general form

$$\bigwedge \text{constraint}$$

where constraint has the form

$X - Y \leq c$

or

$\pm X \leq c$

Zone Abstract Domain (Analysis)

- Developed by Antoine Mine in his Ph.D. thesis
- Uses constraints of the form $X - Y \leq c$ and $\pm X \leq c$
- Built on top of Difference Bound Matrices (DBM) and shortest-path algorithms
  - $O(n^3)$ time
  - $O(n^2)$ space

Analysis with Zone abstract domain

```
nums : array
N : unsigned int
{ N≥0 }
x := 0
{ N≥0 ∧ x=0 }
res := nums[0]
{ N≥0 ∧ x=0 }
Inv = { N≥0 ∧ 0≤x≤N }
while x < N
    { N≥0 ∧ 0≤x<N }
    if nums[x] > res then
        { N≥0 ∧ 0≤x<N }
        res := nums[x]
        { N≥0 ∧ 0≤x<N }
    { N≥0 ∧ 0≤x<N }
    x := x + 1
    { N≥0 ∧ 0<x≤N }
{N≥0 ∧ 0≤x ∧ x=N }
```

Static Analysis with Zone Abstraction

```
nums : array
N : unsigned int
{ N≥0 }
x := 0
{ N≥0 ∧ x=0 }
res := nums[0]
{ x=0 }
Inv = { x≤N }
while x < N
    { x=k ∧ k≤N }
    if nums[x] > res then
        { x=k ∧ k<N }
        res := nums[x]
        { x=k ∧ k<N }
    { x=k ∧ k<N }
    x := x + 1
    { x=k+1 ∧ k≤N }
{ x≤N ∧ x≥N }
{ x=N }
```

Manual Proof

Abstract Interpretation [Cousot'77]

- Mathematical foundation of static analysis

Abstract Interpretation [Cousot'77]

- Mathematical foundation of static analysis

  - Abstract (semantic) domains ("abstract states")
  - Transformer functions ("abstract steps")
  - Chaotic iteration ("abstract computation")

Abstract Interpretation [CC77]

- A very general mathematical framework for approximating semantics
  - Generalizes Hoare Logic
  - Generalizes weakest precondition calculus
- Allows designing sound static analysis algorithms
  - Usually compute by iterating to a fixed-point
  - *Not specific to any programming language style*
- Results of an abstract interpretation are (loop) invariants
  - Can be interpreted as axiomatic verification assertions and used for verification

Abstract Interpretation in 5 Slides

- Disclaimer
  - Do not worry if you feel that you do not understand the next 5 slides
    - You are not expected to …
  - This is just to give you a view of the land …

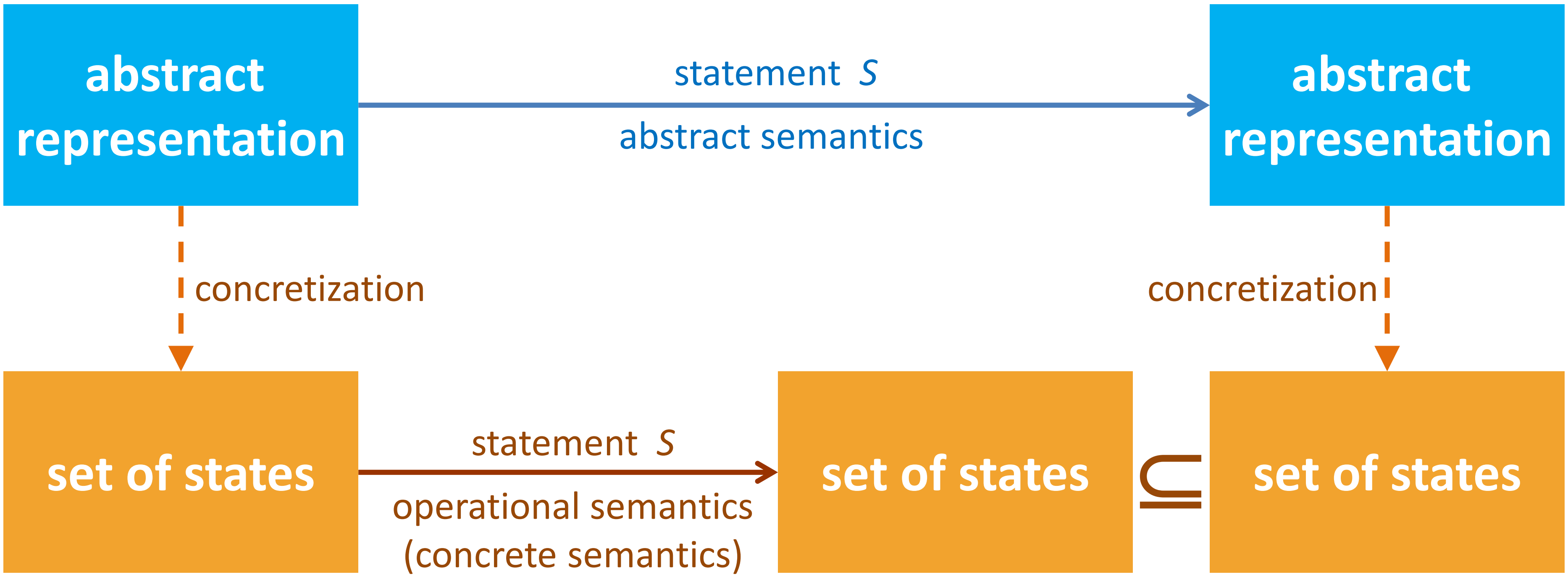
Collecting semantics

- For a set of program states **State**, we define the collecting lattice

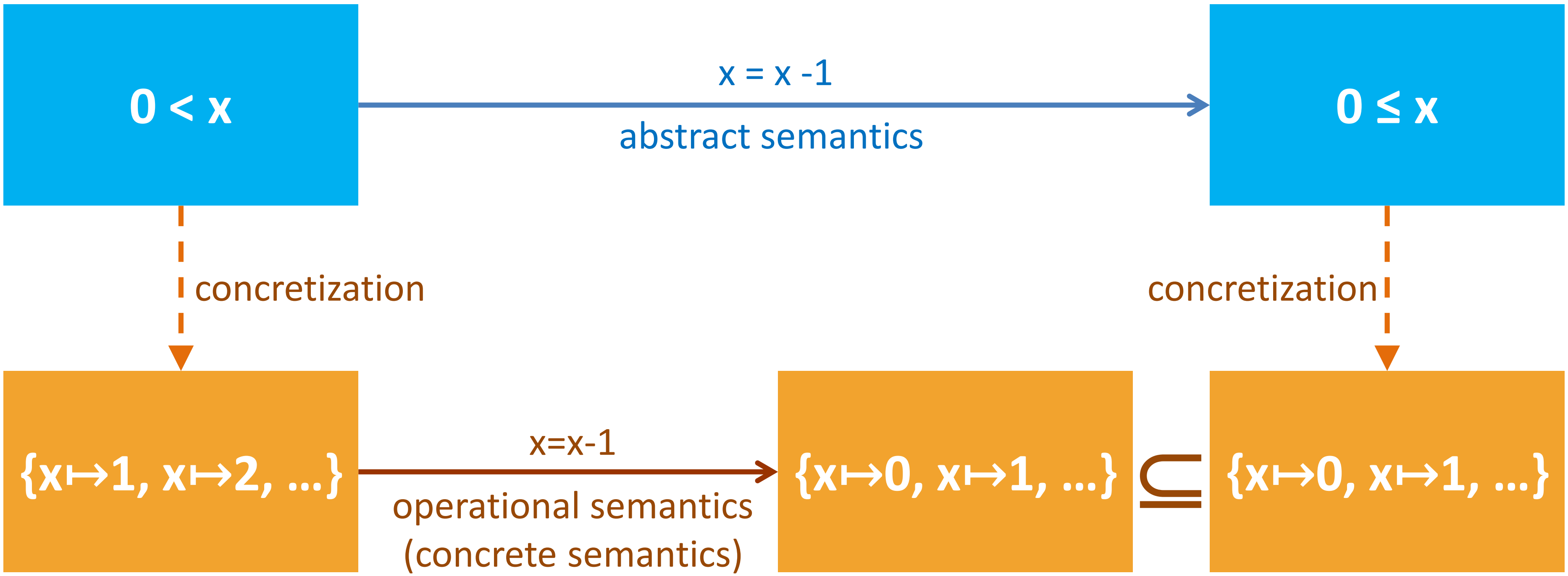
  $(2^{\mathbf{State}}, \subseteq, \cup, \cap, \emptyset, \mathbf{State})$

- The collecting semantics accumulates the (possibly infinite) sets of states generated during the execution
  - Not computable in general

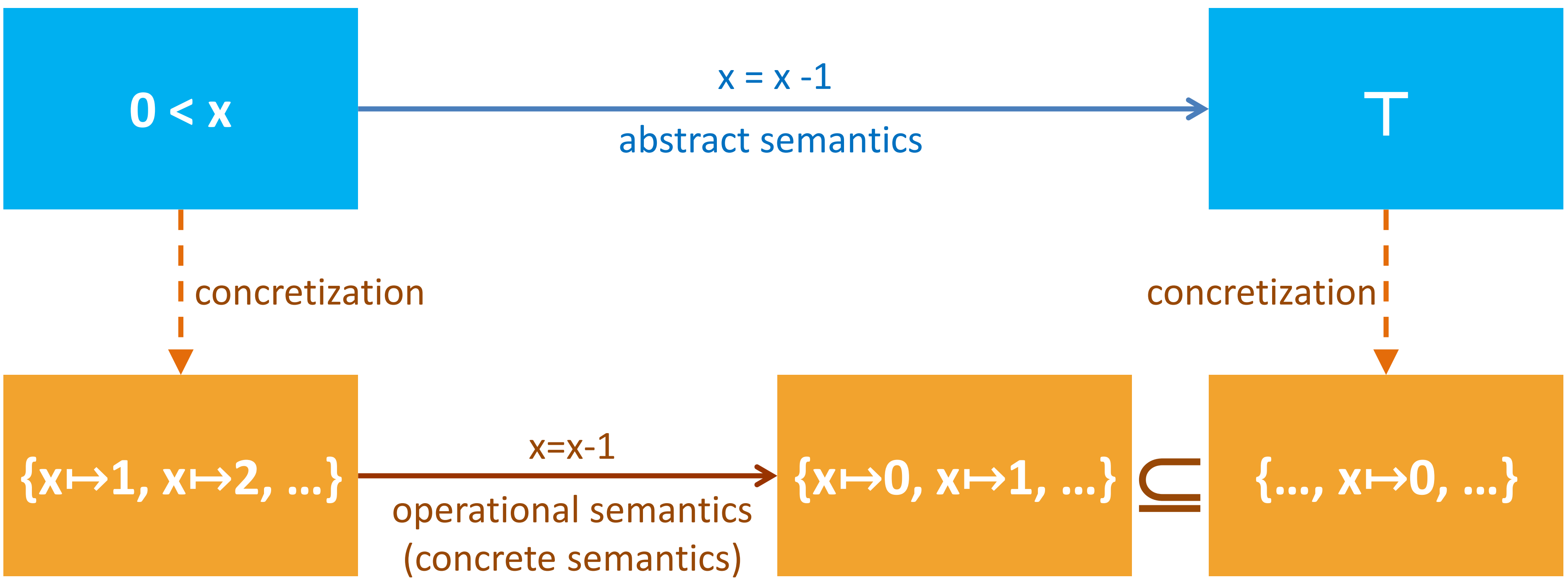
Abstract (conservative) interpretation



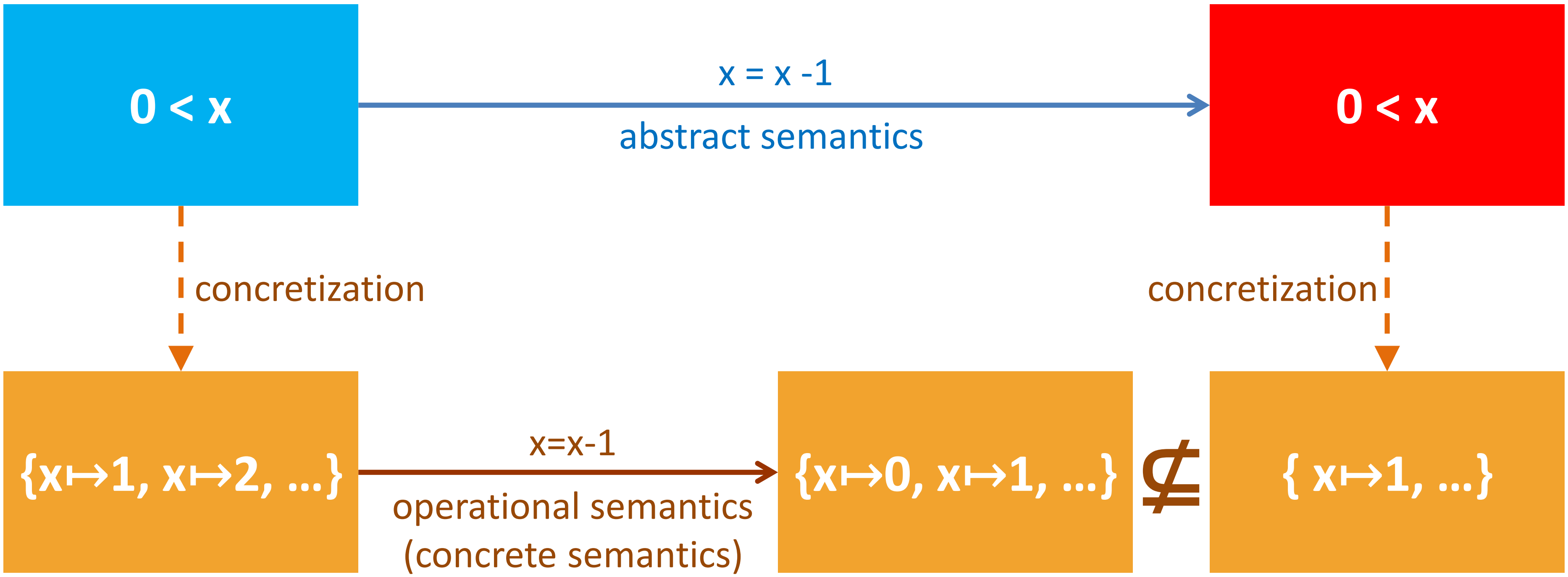
Abstract (conservative) interpretation



Abstract (conservative) interpretation



Abstract (non-conservative) interpretation



Abstract Interpretation by Example

Motivating Application: Optimization

- A compiler optimization is defined by a program transformation:
  $T : \text{Prog} \rightarrow \text{Prog}$
- The transformation is semantics-preserving:
  $\forall s \in \text{State}.\ S_{\text{sos}} [\![C]\!] s = S_{\text{sos}} [\![T(C)]\!] s$
- The transformation is applied to the program only if an *enabling condition* is met
- We use static analysis for inferring enabling conditions

Common Subexpression Elimination

- If we have two variable assignments
  x := a op b
  …
  y := a op b

  op ∈ {+, -, *, ==, <=}

  and the values of x, a, and b have not changed between the assignments, rewrite the code as
  x = a op b
  …
  y := x

- Eliminates useless recalculation
- Paves the way for more optimizations
  - e.g., dead code elimination

What do we need to prove?

{ true }
C_1
x := a op b
C_2
{ x = a op b }
y := a op b
C_3

CSE

{ true }
C_1
x := a op b
C_2
{ x = a op b }
y := x
C_3

Available Expressions Analysis

- A static analysis that infers for every program point a set of facts of the form
  $AV = \{ x = y \mid x, y \in \text{Var} \} \cup$
  $\{ x = -y \mid x, y \in \text{Var} \} \cup$
  $\{ x = y \ op \ z \mid y, z \in \text{Var}, op \in \{+, -, *, <=\} \}$

- For every program with n=|Var| variables number of possible facts is finite: $|AV|=O(n^3)$
  - Yields a trivial algorithm … but, is it efficient?

Which proof is more desirable?

```
{ true }
x := a + b
{ x=a+b }
z := a + c
{ x=a+b }
y := a + b
...
```

```
{ true }
x := a + b
{ x=a+b }
z := a + c
{ z=a+c }
y := a + b
...
```

```
{ true }
x := a + b
{ x=a+b }
z := a + c
{ x=a+b ∧ z=a+c }
y := a + b
...
```

Which proof is more desirable?

```
{ true }
x := a + b
{ x=a+b }
z := a + c
{ x=a+b }
y := a + b
...
```

```
{ true }
x := a + b
{ x=a+b }
z := a + c
{ z=a+c }
y := a + b
...
```

More detailed predicate = more optimization opportunities

```
{ true }
x := a + b
{ x=a+b }
z := a + c
{ x=a+b ∧ z=a+c }
y := a + b
...
```

$x=a+b \wedge z=a+c \Rightarrow x=a+b$

$x=a+b \wedge z=a+c \Rightarrow z=a+c$

Implication formalizes “more detailed” relation between predicates

Developing a theory of approximation

- Formulae are suitable for many analysis-based proofs but we may want to represent predicates in other ways:
  - Sets of “facts”
  - Automata
  - Linear (in)equalities
  - … ad-hoc representation

- Wanted: a uniform theory to represent semantic values and approximations

Preorder

- We say that a binary order relation $\sqsubseteq$ over a set D is a preorder if the following conditions hold for every d, d', d'' $\in$ D
  - Reflexive: $d \sqsubseteq d$
  - Transitive: $d \sqsubseteq d'$ and $d' \sqsubseteq d''$ implies $d \sqsubseteq d''$

- There may exist d, d' such that

  $d \sqsubseteq d'$ and $d' \sqsubseteq d$ yet $d \neq d'$

Preorder example

- **S**imple **A**vailable **E**xpressions
- Define SAV = $\{ x = y \mid x, y \in \text{Var} \} \cup$
  $\{ x = y + z \mid y, z \in \text{Var} \}$
- For $D=2^{SAV}$ (sets of available expressions) define (for two subsets $A_1, A_2 \in D$)
  $A_1 \sqsubseteq^{imp} A_2$ if and only if $\bigwedge A_1 \Rightarrow \bigwedge A_2$
- A_1 is "more detailed" if it implies all facts of A_2

Can we decide $A_1 \sqsubseteq^{imp} A_2$?

- Compare {x=y $\wedge$ x=a+b} with {x=y $\wedge$ y=a+b}
  - Which one should we choose?

The meaning of implication

- A predicate P represents the set of states
  $models(P) = \{ s \mid s \models P \}$
- $P \Rightarrow Q$ means
  $models(P) \subseteq models(Q)$

Partially ordered sets

- A partially ordered set (poset) is a pair (D , $\sqsubseteq$)
  - D is a set of elements – a (semantic) domain
  - $\sqsubseteq$ is a partial order between pairs of elements from D. That is $\sqsubseteq$: $D \times D$ with the following properties, for all d, d’, d’’ in D
    - Reflexive: $d \sqsubseteq d$
    - Transitive: $d \sqsubseteq d'$ and $d' \sqsubseteq d''$ implies $d \sqsubseteq d''$
    - **Anti-symmetric: $d \sqsubseteq d'$ and $d' \sqsubseteq d$ implies $d = d'$**

Unique “most detailed” element

- Notation: if $d \sqsubseteq d'$ and $d \neq d'$ we write $d \sqsubset d'$

From preorders to partial orders

- We can transform a preorder into a poset by
  1. Coarsening the ordering

  2. Switching to a canonical form by choosing a representative for the set of equivalent elements d^* for { d' | d $\sqsubseteq$ d' and d' $\sqsubseteq$ d }

Coarsening for SAV

- For $D=2^{SAV}$ (sets of available expressions) define (for two subsets $A_1, A_2 \in D$)
  $A_1 \sqsubseteq^{coarse} A_2$ if and only if $A_1 \supseteq A_2$
- Notice that if $A_1 \supseteq A_2$ then $\bigwedge A_1 \Rightarrow \bigwedge A_2$
- Compare {x=y ∧ x=a+b} with {x=y ∧ y=a+b}
- How about {x=y ∧ x=a+b ∧ y=a+b} ?

Canonical form for SAV

- For an available expressions element A define *Explicate*(A) = minimal set B such that:
  1. $A \subseteq B$
  2. x=y $\in B$ implies y=x $\in B$
  3. x=y $\in B$ and y=z $\in B$ implies x=z $\in B$
  4. x=y+z $\in B$ implies x=z+y $\in B$
  5. x=y $\in B$ and x=z+w $\in B$ implies y=z+w $\in B$
  6. x=y $\in B$ and z=x+w $\in B$ implies z=y+w $\in B$
  7. x=z+w $\in B$ and y=z+w $\in B$ implies x=y $\in B$
- Makes all implicit facts explicit

- Define $A^* = Explicate(A)$
- Define (for two subsets $A_1, A_2 \in$ D) $A_1 \sqsubseteq^{exp} A_2$ if and only if $A_1^* \supseteq A_2^*$
- **Lemma:** $A_1 \sqsubseteq^{exp} A_2$ if and only $A_1 \sqsubseteq^{imp} A_2$

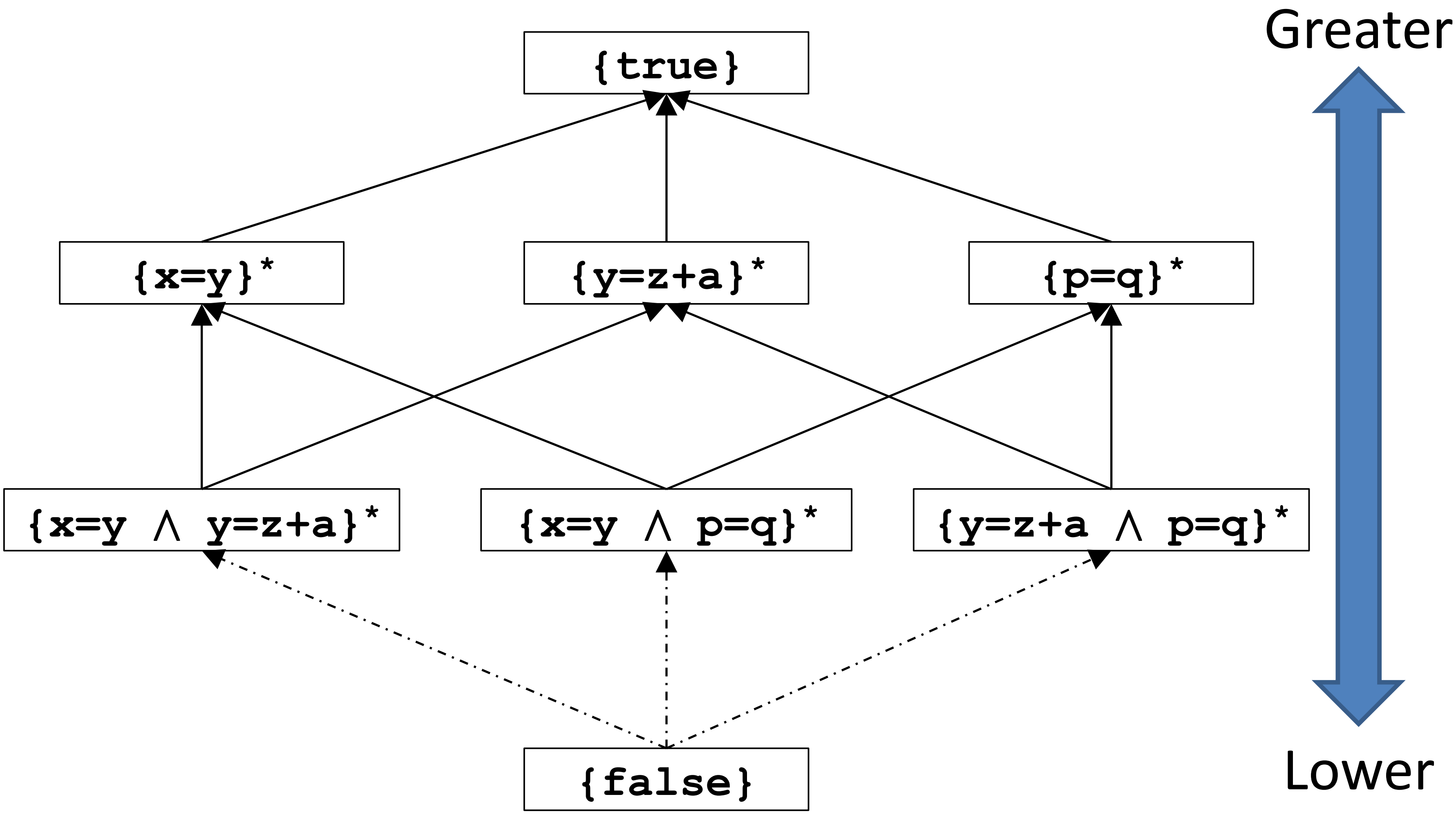
Therefore $A_1 \sqsubseteq^{imp} A_2$ is decidable

Some posets-related terminology

- If $x \sqsubseteq y$ we can say
  - x is *lower* than y
  - x is *more precise* than y
  - x is *more concrete* than y
  - x *under-approximates* y

  - y is *greater* than x
  - y is *less precise* than x
  - y is *more abstract* than x
  - y *over-approximates* x

Visualizing ordering for SAV



D={x=y, y=x, p=q, q=p, y=z+a, y=a+z, z=y+z, x=z+a}

Pointed poset

- A poset (D, $\sqsubseteq$) with a least element $\perp$ is called a pointed poset
  - For all d$\in$D we have that $\perp \sqsubseteq$ d
- The pointed poset is denoted by (D , $\sqsubseteq$, $\perp$)
- We can always transform a poset (D, $\sqsubseteq$) into a pointed poset by adding a special bottom element

  $$(D \cup \{\perp\}, \sqsubseteq \cup \{\perp \sqsubseteq d \mid d \in D\}, \perp)$$
- Greatest element for SAV = {**true** = ?}
- Least element for SAV = {**false**= ?}

Annotating conditions

$$[\text{if}_p]\quad \frac{\{ b \wedge P \}\, S_1\, \{ Q \},\quad \{ \neg b \wedge P \}\, S_2\, \{ Q \}}{\{ P \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{ Q \}}$$

$\{ P \}$
`if` b `then`
    $\{ b \wedge P \}$
    S_1
    $\{ Q_1 \}$
`else`
    $\{ b \wedge P \}$
    S_2
    $\{ Q_2 \}$
$\{ Q \}$

Q approximates Q_1 and Q_2

We need a general way to approximate a set of semantic elements by a single semantic element

Join operator

- Assume a poset $(D, \sqsubseteq)$
- Let $X \subseteq D$ be a subset of D (finite/infinite)
- The join of X is defined as
  - $\sqcup X$ = the least upper bound (LUB) of all elements in X *if it exists*
  - $\sqcup X = \min_{\sqsubseteq}\{ b \mid \text{forall } x \in X \text{ we have that } x \sqsubseteq b\}$
  - The supremum of the elements in X
  - A kind of abstract union (disjunction) operator
- Properties of a join operator
  - Commutative: $x \sqcup y = y \sqcup x$
  - Associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  - Idempotent: $x \sqcup x = x$

Meet operator

- Assume a poset $(D, \sqsubseteq)$
- Let $X \subseteq D$ be a subset of D (finite/infinite)
- The meet of X is defined as
  - $\sqcap X$ = the greatest lower bound (GLB) of all elements in X *if it exists*
  - $\sqcap X = \max_{\sqsubseteq}\{ b \mid \text{forall } x \in X \text{ we have that } b \sqsubseteq x\}$
  - The infimum of the elements in X
  - A kind of abstract intersection (conjunction) operator
- Properties of a join operator
  - Commutative: $x \sqcap y = y \sqcap x$
  - Associative: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
  - Idempotent: $x \sqcap x = x$

Complete lattices

- A complete lattice $(D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is
- A set of elements D
- A partial order $x \sqsubseteq y$
- A join operator $\sqcup$
- A meet operator $\sqcap$
- A bottom element
  $\bot = ?$
- A top element
  $\top = ?$

Complete lattices

- A complete lattice $(D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is
- A set of elements D
- A partial order $x \sqsubseteq y$
- A join operator $\sqcup$
- A meet operator $\sqcap$
- A bottom element
  $\bot = \sqcup \varnothing$
- A top element
  $\top = \sqcup D$

Transfer Functions

- Mathematical foundations

Towards an automatic proof

- **Goal:** automatically compute an annotated program proving as many facts of the form $x = y + z$ as possible
- **Decision 1:** develop a forward-going proof
- **Decision 2:** draw predicates from a finite set $\mathbb{D}$
  - "looking under the light of the lamp"
  - A compromise that simplifies problem by focusing attention – possibly miss some facts that hold
- **Challenge 1:** handle straight-line code
- **Challenge 2:** handle conditions
- **Challenge 3:** handle loops

Domain for SAV

- Define *atomic facts* (for SAV) as
  $\theta = \{ x = y \mid x, y \in \text{Var} \} \cup \{ x = y + z \mid x, y, z \in \text{Var} \}$
  - For n=|Var| number of atomic facts is $O(n^3)$
- Define *sav-predicates* as $\Pi = 2^\theta$
- For $D \subseteq \theta$, Conj(D) = $\bigwedge D$
  - Conj($\{a=b, c=b+d, b=c\}$) = (a=b) $\wedge$ (c=b+d) $\wedge$ (b=c)
- Note:
  - Conj($D_1 \cup D_2$) = Conj(D_1) $\wedge$ Conj(D_1)
  - Conj({}) $\Leftrightarrow$ true

Challenge 2: handling straight-line code

handling straight-line code: Goal

- Given a program of the form

$$x_1 := a_1; \ldots x_n := a_n$$

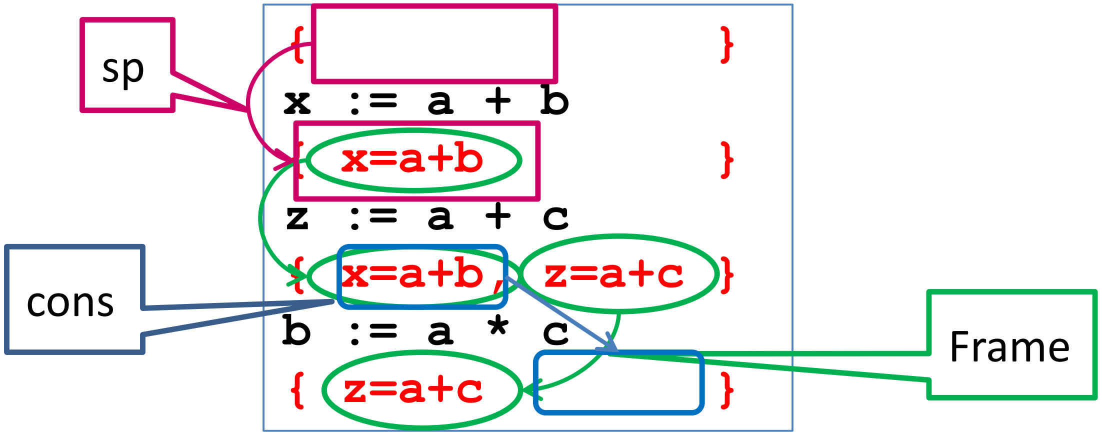
- Find predicates $P_0, \ldots, P_n$ such that
  1. $\{P_0\}\ x_1 := a_1\ \{P_1\} \ldots \{P_{n-1}\}\ x_n := a_n\ \{P_n\}$ is a proof
     - **sp**$(x_i := a_i, P_{i-1}) \Rightarrow P_i$
  2. $P_i = \text{Conj}(D_i)$
     - D_i is a set of simple (SAV) facts

Example

```
{                  }
x := a + b
{                  }
z := a + c
{                  }
b := a * c
{                  }
```

- Find a proof that satisfies both conditions

Example



- Can we make this into an algorithm?

Algorithm for straight-line code

- **Goal:** find predicates P_0, …, P_n such that
  1. $\{P_0\}\ x_1 := a_1\ \{P_1\} \ldots \{P_{n-1}\}\ x_n := a_n\ \{P_n\}$ is a proof
     That is: $\mathbf{sp}(x_i := a_i, P_{i-1}) \Rightarrow P_i$
  2. Each P_i has the form Conj(D_i) where D_i is a set of simple (SAV) facts
- **Idea:** define a function $F^{SAV}[x:=a] : \Pi \to \Pi$ s.t.
  if $F^{SAV}[x:=a](D) = D'$
  then $\mathbf{sp}(x := a, \mathrm{Conj}(D)) \Rightarrow \mathrm{Conj}(D')$
  – We call F the abstract transformer for $x:=a$
- Initialize $D_0=\{\}$
- For each i: compute $D_{i+1} = \mathrm{Conj}(F^{SAV}[x_i := a_i]\ D_i)$
- Finally $P_i = \mathrm{Conj}(D_i)$

Defining an SAV abstract transformer

- **Goal:** define a function $F^{SAV}[x:=a] : \Pi \rightarrow \Pi$ s.t.
  if $F^{SAV}[x:=a](D) = D'$
  then **sp**(x := a, Conj(D)) $\Rightarrow$ Conj(D')

Defining an SAV abstract transformer

- **Goal:** define a function $F^{SAV}[x{:=}a] : \Pi \rightarrow \Pi$ s.t.
  if $F^{SAV}[x{:=}a](D) = D'$
  then **sp**($x := a$, Conj(D)) $\Rightarrow$ Conj(D')
- **Idea:** define rules for individual facts and generalize to sets of facts by the conjunction rule

Defining an SAV abstract transformer

- **Goal:** define a function $F^{SAV}[x{:=}a] : \Pi \to \Pi$ s.t.
  if $F^{SAV}[x{:=}a](D) = D'$
  then **sp**($x := a$, Conj(D)) $\Rightarrow$ Conj(D')
- **Idea:** define rules for individual facts and generalize to sets of facts by the conjunction rule

[kill-lhs] $\{ x=\omega \}\ x{:=}a\ \{ \}$

[kill-rhs-1] $\{ y=x+w \}\ x{:=}a\ \{ \}$

[kill-rhs-2] $\{ y=w+x \}\ x{:=}a\ \{ \}$

[gen] $\{ \}\ x{:=}\ \omega\ \{ x=\omega \}$

[preserve] $\{ y=z+w \}\ x{:=}a\ \{ y=z+w \}$

ω Is either a variable v or an addition expression $v+w$

SAV abstract transformer example

```
{                }
x := a + b
{ x=a+b          }
z := a + c
{ x=a+b, z=a+c }
b := a * c
{ z=a+c          }
```

[kill-lhs] { $x=\omega$ } $x:=a$ { }

[kill-rhs-1] { $y=x+w$ } $x:=a$ { }

[kill-rhs-2] { $y=w+x$ } $x:=a$ { }

[gen] { } $x:=\omega$ { $x=\omega$ }

[preserve] { $y=z+w$ } $x:=a$ { $y=z+w$ }

ω Is either a variable v or an addition expression $v+w$

Problem 1: large expressions

```
{              }
x := a + b + c
{              }
y := a + b + c
{              }
```

Missed CSE opportunity

- Large expressions on the right hand sides of assignments are problematic
  - Can miss optimization opportunities
  - Require complex transformers
- Solution: transform code to normal form where right-hand sides have bounded size

Solution: Simplify Prog. Lang.

```
{                }
x := a + b + c
{                }
y := a + b + c
{                }
```

```
{                }
i1 := a + b
{                }
x := i1 + c
{                  }
i2 := a + b
{                          }
y := i2 + c
{                                  }
```

- Main idea: simplify expressions by storing intermediate results in new temporary variables
  - Three-address code
- Number of variables in simplified statements ≤ 3

Solution: Simplify Prog. Lang.

```
{                }
x := a + b + c
{                }
y := a + b + c
{                }
```

```
{                }
i1 := a + b
{ i1=a+b         }
x := i1 + c
{ i1=a+b, x=i1+c }
i2 := a + b
{ i1=a+b, x=i1+c, i2=a+b }
y := i2 + c
{ i1=a+b, x=i1+c, i2=a+b, y=i2+c }
```

Need to infer
i1=i2

- Main idea: simplify expressions by storing intermediate results in new temporary variables
  - Three-address code
- Number of variables in simplified statements ≤ 3

Problem 2: Transformer Precision

```
{                   }
i1 := a + b
{ i1=a+b            }
x := i1 + c
{ i1=a+b, x=i1+c }
i2 := a + b
{ i1=a+b, x=i1+c, i2=a+b }
y := i2 + c
{ i1=a+b, x=i1+c, i2=a+b, y=i2+c }
```

Need to infer `i1=i2`

- Our transformer only infers syntactically available expressions – ones that appear in the code explicitly
- We want a transformer that looks deeper into the semantics of the predicates
  - Takes equalities into account

Solution: Use Canonical Form

- **Idea:** make as many implicit facts explicit by
  - Using symmetry and transitivity of equality
  - Commutativity of addition
  - Meaning of equality – can substitute equal variables
- For P=Conj(D) let *Explicate*(D) = minimal set D^* such that:
  1. $D \subseteq D^*$
  2. x=y $\in D^*$ implies y=x $\in D^*$
  3. x=y $\in D^*$ y=z $\in D^*$ implies x=z $\in D^*$
  4. x=y+z $\in D^*$ implies x=z+y $\in D^*$
  5. x=y $\in D^*$ and x=z+w $\in D^*$ implies y=z+w $\in D^*$
  6. x=y $\in D^*$ and z=x+w $\in D^*$ implies z=y+w $\in D^*$
  7. x=z+w $\in D^*$ and y=z+w $\in D^*$ implies x=y $\in D^*$
- Notice that *Explicate*(D) $\Leftrightarrow$ D
  - *Explicate* is a special case of a reduction operator

Sharpening the transformer

- **Define:** $F^{*}[x:=a]$ = Explicate ∘ $F^{SAV}[x:=a]$

```
{                 }
i1 := a + b
{ i1=a+b, i1=b+a        }
x := i1 + c
{ i1=a+b, i1=b+a, x=i1+c, x=c+i1 }
i2 := a + b
{ i1=a+b, i1=b+a, x=i1+c, x=c+i1, i2=a+b,
  i2=b+a, i1=i2, i2=i1, x=i2+c, x=c+i2, }
y := i2 + c
{ ... }
```

Since sets of facts and their conjunction are isomorphic we will use them interchangeably

An algorithm for annotating SLP

- Annotate(P, x:=a) = {P} x:=a $F^*[x:=a](P)$

- Annotate(P, S_1; S_2) = {P} S_1; {Q_1} S_2 {Q_2
  - Annotate(P, S_1) = {P} S_1 {Q_1}
  - Annotate(Q_1, S_2) = {Q_1} S_2 {Q_2}

Challenge 2: handling conditions

handling conditions: Goal

$$[\mathrm{if}_p]\ \frac{\{\, b \wedge P \,\}\, S_1 \,\{\, Q \,\},\quad \{\, \neg b \wedge P \,\}\, S_2 \,\{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\, Q \,\}}$$

- Annotate a program
  `if` b `then` S_1 `else` S_2
  with predicates from Π

$\{ P \}$
`if` b `then`
    $\{ b \wedge P \}$
    S_1
    $\{ Q_1 \}$
`else`
    $\{ \neg b \wedge P \}$
    S_2
    $\{ Q_2 \}$
$\{ Q \}$

handling conditions: Goal

$$[\text{if}_p]\ \frac{\{\, b \wedge P \,\}\, S_1 \,\{\, Q \,\},\quad \{\, \neg b \wedge P \,\}\, S_2 \,\{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ \ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\, Q \,\}}$$

- Annotate a program
  $\texttt{if}\ \ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$
  with predicates from Π
- **Assumption 1:** *P* is given (otherwise use true)
- **Assumption 2:** *b* is a simple binary expression
  e.g., x=y, x≠y, x<y (why?)

```
{ P }
if  b then
        { b ∧ P }
        S1
        { Q1 }
else
        { ¬b ∧ P }
        S2
        { Q2 }
{ Q }
```

Annotating conditions

$$[\text{if}_p]\ \frac{\{\, b \wedge P \,\}\ S_1\ \{\, Q \,\},\quad \{\, \neg b \wedge P \,\}\ S_2\ \{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\, Q \,\}}$$

1. Start with P or $\{b \wedge P\}$ and annotate S_1 (yielding Q_1)
2. Start with P or $\{\neg b \wedge P\}$ and annotate S_2 (yielding Q_2)
3. How do we infer a Q such that $Q_1 \Rightarrow Q$ and $Q_2 \Rightarrow Q$?

$\{ P \}$
`if` b `then`
    $\{ b \wedge P \}$
    S_1
    $\{ Q_1 \}$
`else`
    $\{ \neg b \wedge P \}$
    S_2
    $\{ Q_2 \}$
$\{ Q \}$

Possibly an SAV-fact

Possibly an SAV-fact

Joining predicates

$$[\text{if}_p]\ \frac{\{\, b \wedge P \,\}\, S_1\, \{\, Q \,\},\quad \{\, \neg b \wedge P \,\}\, S_2\, \{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\, Q \,\}}$$

1. Start with P or $\{b \wedge P\}$ and annotate S_1 (yielding Q_1)
2. Start with P or $\{\neg b \wedge P\}$ and annotate S_2 (yielding Q_2)
3. How do we infer a Q such that $Q_1 \Rightarrow Q$ and $Q_2 \Rightarrow Q$?

   Q_1=Conj(D_1), Q_2=Conj(D_2)
   Define: $Q = Q_1 \sqcup Q_2$
   $= \text{Conj}(D_1 \cap D_2)$

The join operator for SAV

```
{ P }
if b then
        { b ∧ P }
        S1
        { Q1 }
else
        { ¬b ∧ P }
        S2
        { Q2 }
{ Q }
```

Joining predicates

- Q_1=Conj(D_1), Q_2=Conj(D_2)
- We want to soundly approximate $Q_1 \vee Q_2$ in Π
- Define: $Q = Q_1 \sqcup Q_2$
  $= \text{Conj}(D_1 \cap D_2)$
- Notice that $Q_1 \Rightarrow Q$ and $Q_2 \Rightarrow Q$
  meaning $Q_1 \vee Q_2 \Rightarrow Q$

Handling conditional expressions

- Let D be a set of facts and b be an expression
- Goal: Elements in Π that soundly approximate
  - $D \wedge bexpr$
  - $D \wedge \neg bexpr$
- Technique: Add statement `assume` ***bexpr***

$$\langle \texttt{assume}\ \boldsymbol{bexpr}, \boldsymbol{s}\rangle \Rightarrow^{sos} \boldsymbol{s} \text{ if } \mathcal{B}[\![bexpr]\!]\, s = \mathbf{tt}$$

- Find a function F[`assume` ***bexpr***] : $\Pi \rightarrow \Pi$

$$\text{Conj}(D) \wedge \boldsymbol{bexpr} \Rightarrow \text{Conj}(\text{F}[\texttt{assume}\ \boldsymbol{bexpr}])$$

Handling conditional expressions

- F[`assume` *bexpr*] : $\Pi \rightarrow \Pi$ such that
  Conj(*D*) $\wedge$ *bexpr* $\Rightarrow$ Conj(F[`assume` *bexpr*])

- β(*bexpr*) = if *bexpr* is an SAV-fact then {*bexpr*} else {}
  - Notice *bexpr* $\Rightarrow$ β(*bexpr*)
  - Examples
    - β(y=z) = {y=z}
    - β(y<z) = {}

- F[`assume` *bexpr*](*D*) = *D* $\cup$ β(*bexpr*)

Example

```
{                 }
if (x = y)
     {            }
     a := b + c
     {                               }
     d := b - c
     {                               }
else
     {                 }
     a := b + c
     {                 }
     d := b + c
     {                                                 }
{                 }
```

Example

```
{                  }
if (x = y)
     { x=y, y=x }
     a := b + c
     { x=y, y=x, a=b+c, a=c+b }
     d := b - c
     { x=y, y=x, a=b+c, a=c+b }
else
     {                  }
     a := b + c
     { a=b+c, a=c+b }
     d := b + c
     { a=b+c, a=c+b, d=b+c, d=c+b, a=d, d=a }
{ a=b+c, a=c+b }
```

Recap

- We now have an **algorithm** for soundly annotating loop-free code
- Generates forward-going proofs
- Algorithm operates on abstract syntax tree of code
  - Handles straight-line code by applying F^*
  - Handles conditions by recursively annotating true and false branches and then intersecting their postconditions

An algorithm for conditions

- Annotate(P, `if` *bexpr* `then` S_1 `else` S_2) =

  {P}
  `if` *bexpr* `then` S_1 `else` S_2
  $Q_1 \sqcup Q_2$

  - Annotate($P \cup \beta(bexpr), S_1$) =
    F[`assume` *bexpr*](P) S_1 {Q_1}

  - Annotate($P \cup \beta(\neg bexpr), S_2$)=
    F[`assume` $\neg$*bexpr*](P) S_2 {Q_2}

Challenge 2: handling loops

handling loops: Goal

[while$_p$] $$\frac{\{ bexpr \wedge P \}\ S\ \{ P \}}{\{ P \}\ \texttt{while}\ \ b\ \texttt{do}\ S\ \{\neg bexpr \wedge P \}}$$

$\{ P \}$

Inv = $\{ N \}$

$\texttt{while}$ $bexpr$ $\texttt{do}$

$\{ bexpr \wedge N \}$

S

$\{ Q \}$

$\{\neg bexpr \wedge N \}$

handling loops: Goal

$$[\text{while}_p]\ \frac{\{bexpr \land P\}\ S\ \{P\}}{\{P\}\ \texttt{while}\ \ b\ \texttt{do}\ S\ \{\neg bexpr \land P\}}$$

- Annotate a program $\texttt{while}\ bexpr\ \texttt{do}\ S$ with predicates from Π
  - s.t. $P \Rightarrow N$
- **Main challenge:** find N
- **Assumption 1:** P is given (otherwise use true)
- **Assumption 2:** $bexpr$ is a simple binary expression

$\{ P \}$

Inv = $\{ N \}$

$\texttt{while}\ bexpr\ \texttt{do}$

$\quad \{ bexpr \land N \}$

$\quad S$

$\quad \{ Q \}$

$\{\neg bexpr \land N \}$

Example: annotate this program

```
{ y=x+a, y=a+x, w=d, d=w }
Inv = {                                              }
while (x ≠ z) do
    {                                                }
    x := x + 1
    {                                                }
    y := x + a
    {                                                }
    d := x + a
    {                                                }
{                                                    }
```

Example: annotate this program

```
{ y=x+a, y=a+x, w=d, d=w }
Inv = { y=x+a, y=a+x }
while (x ≠ z) do
    { y=x+a, y=a+x }
    x := x + 1
    { }
    y := x + a
    { y=x+a, y=a+x }
    d := x + a
    { y=x+a, y=a+x, d=x+a, d=a+x, y=d, d=y }
{ y=x+a, y=a+x, x=z, z=x }
```

handling loops: Idea

$$[\text{while}_p]\quad \frac{\{bexpr \land P\}\ S\ \{P\}}{\{P\}\ \texttt{while}\ \ b\ \texttt{do}\ S\ \{\neg bexpr \land P\}}$$

- **Idea:** try to guess a loop invariant from a small number of loop unrollings
  - We know how to annotate *S* (by induction)

$\{ P \}$

Inv = $\{ N \}$

`while` *bexpr* `do`

$\quad\{ bexpr \land N \}$

$\quad S$

$\quad\{ Q \}$

$\{\neg bexpr \land N \}$

k-loop unrolling

```
{ P }
Inv = { N }
while (x ≠ z) do
    x := x + 1
    y := x + a
    d := x + a
```

```
{ y=x+a, y=a+x, w=d, d=w }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = {          }
```

```
{ P }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = {          }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q2 = {          }
```

...

k-loop unrolling

```
{ P }
Inv = { N }
while (x ≠ z) do
    x := x + 1
    y := x + a
    d := x + a
```

⟺

```
{ y=x+a, y=a+x, w=d, d=w }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
```

```
{ P }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q2 = { y=x+a, y=a+x }
```

…

k-loop unrolling

```
{ P }
Inv = { N }
while (x ≠ z) do
    x := x + 1
    y := x + a
    d := x + a
```

The following must hold:

$P \Rightarrow N$

$Q_1 \Rightarrow N$

$Q_2 \Rightarrow N$

…

$Q_k \Rightarrow N$

```
{ y=x+a, y=a+x, w=d, d=w }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
```

```
{ P }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q2 = { y=x+a, y=a+x }
```

…

k-loop unrolling

```
{ P }
Inv = { N }
while (x ≠ z) do
    x := x + 1
    y := x + a
    d := x + a
```

```
{ y=x+a, y=a+x, w=d, d=w }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
```

```
{ P }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q2 = { y=x+a, y=a+x }
```

…

The following must hold:

$P \Rightarrow N$

$Q_1 \Rightarrow N$

$Q_2 \Rightarrow N$

…

$Q_k \Rightarrow N$

…

Observation 1: No need to explicitly unroll loop – we can reuse postcondition from unrolling k-1 for k

We can compute the following sequence:

$N_0 = P$

$N_1 = N_1 \sqcup Q_1$

$N_2 = N_1 \sqcup Q_2$

…

$N_k = N_{k\text{-}1} \sqcup Q_k$

k-loop unrolling

```
{ P }
Inv = { N }
while (x ≠ z) do
    x := x + 1
    y := x + a
    d := x + a
```

```
{ y=x+a, y=a+x, w=d, d=w }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
```

```
{ P }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q1 = { y=x+a, y=a+x }
if (x ≠ z)
    x := x + 1
    y := x + a
    d := x + a
Q2 = { y=x+a, y=a+x }
```

…

The following must hold:

$P \Rightarrow N$

$Q_1 \Rightarrow N$

$Q_2 \Rightarrow N$

…

$Q_k \Rightarrow N$

…

Observation 2: N_k monotonically decreases set of facts.
Question: does it stabilizes for some k?

We can compute the following sequence:

$N_0 = P$

$N_1 = N_1 \sqcup Q_1$

$N_2 = N_1 \sqcup Q_2$

…

$N_k = N_{k-1} \sqcup Q_k$

Algorithm for annotating a loop

Annotate(P, `while` *bexpr* `do S`) =

Initialize $N' := N_c := P$

repeat

let Annotate(P, **`if b then S else skip`**) be

$\{N_c\}$ **`if`** *bexpr* **`then`** *S* **`else skip`** $\{N'\}$

$N_c := N_c \sqcup N'$

until $N' = N_c$

return {P}

INV= N'

`while` *bexpr* `do`

F[`assume` *bexpr*](*N*)

Annotate(F[`assume` *bexpr*](*N*), S)

F[`assume` ¬*bexpr*](*N*)

A technical issue

- Unrolling loops is quite inconvenient and inefficient (but we can avoid it as we just saw)
- How do we handle more complex control-flow constructs, e.g., `goto` , `break`, exceptions…?
- **Solution:** model control-flow by labels and `goto` statements
- Would like a dedicated data structure to explicitly encode control flow in support of the analysis
- **Solution:** control-flow graphs (CFGs)

Intermediate language example

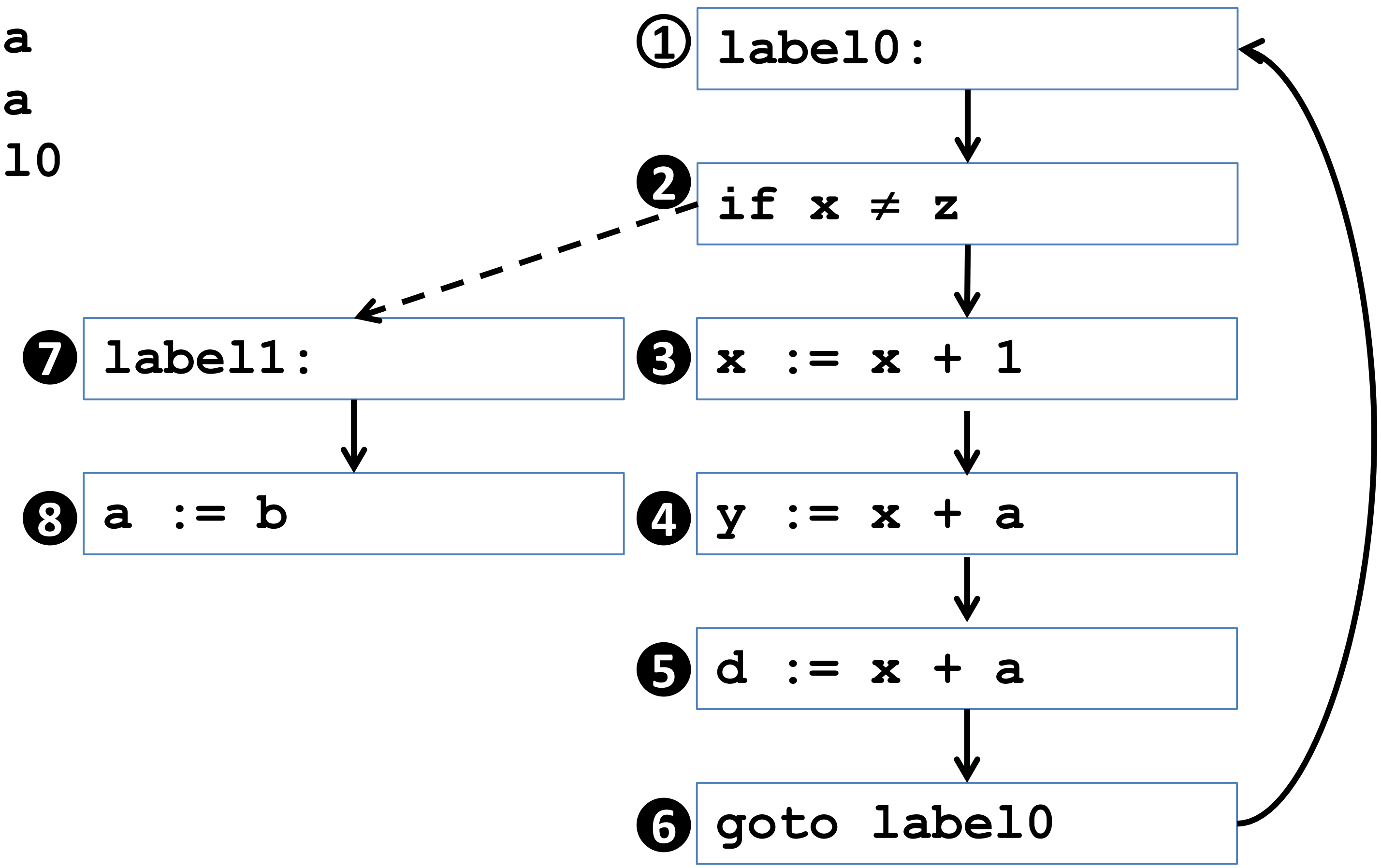
```
while (x ≠ z) do
    x := x + 1
    y := x + a
    d := x + a
a := b
```

```
label0:
   if x ≠ z goto label1
   x := x + 1
   y := x + a
   d := x + a
   goto label0

label1:
   a := b
```

Control-flow graph example

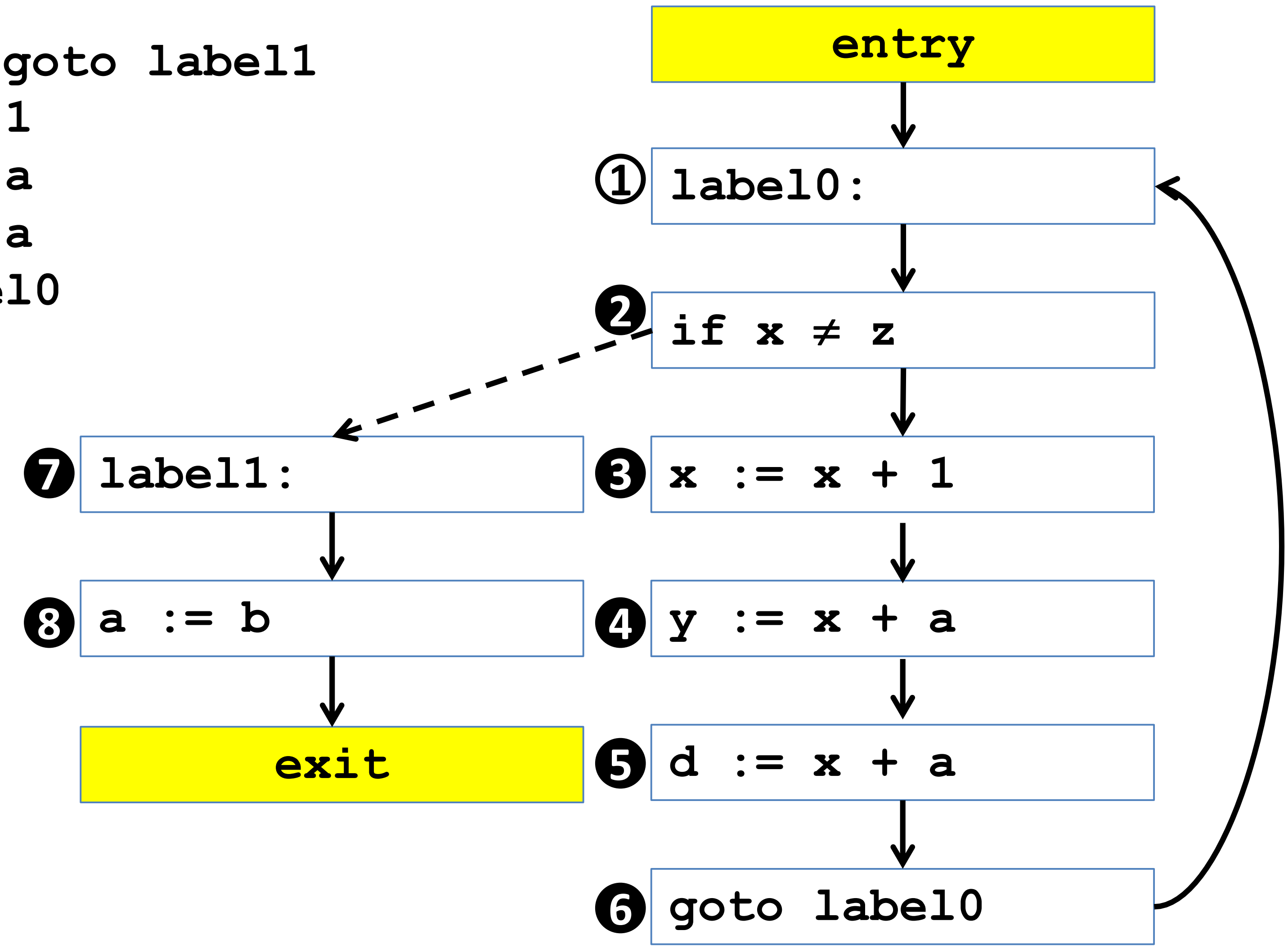
line number

```
① label0:
❷   if x ≠ z goto label1
❸   x := x + 1
❹   y := x + a
❺   d := x + a
❻   goto label0

❼ label1:
❽   a := b
```



Control-flow graph example

```
① label0:
❷   if x ≠ z goto label1
❸   x := x + 1
❹   y := x + a
❺   d := x + a
❻   goto label0

❼ label1:
❽   a := b
```

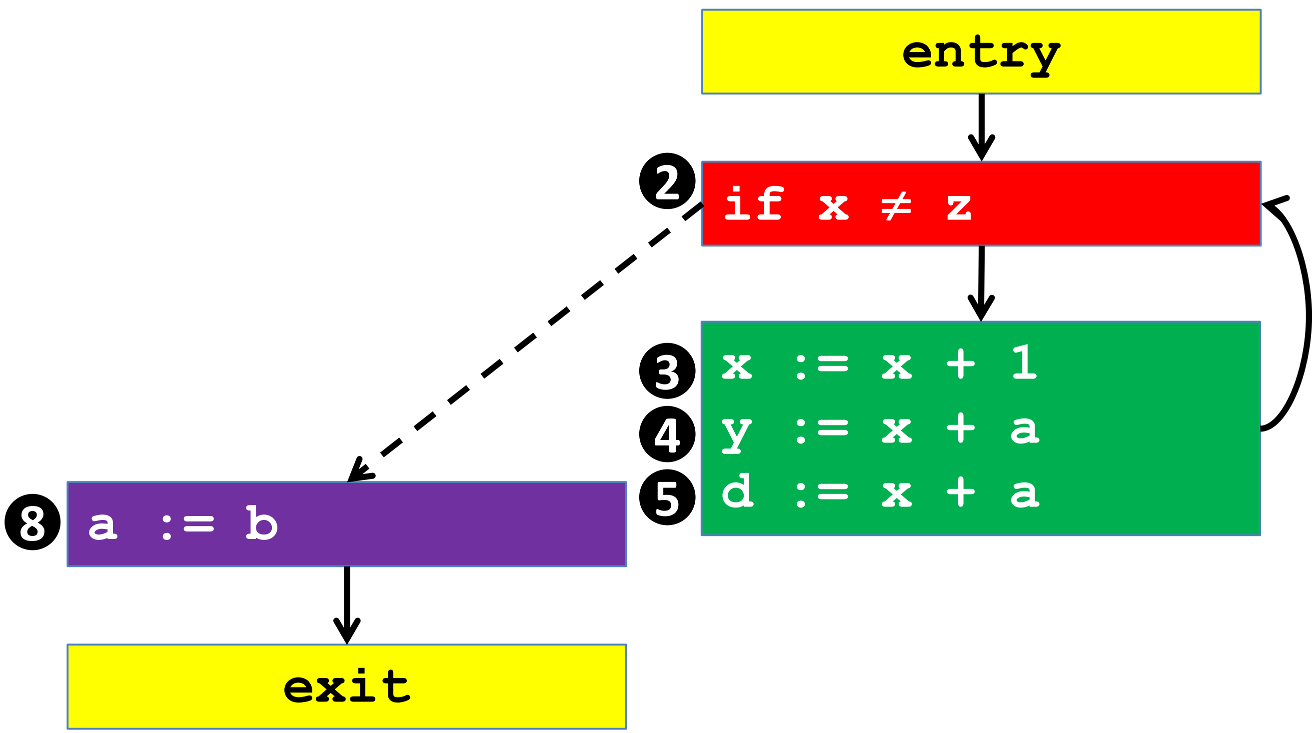


Control-flow graph

- Node are statements or labels
- Special nodes for entry/exit
- A edge from node *v* to node *w* means that after executing the statement of *v* control passes to *w*
  - Conditions represented by splits and join node
  - Loops create cycles
- Can be generated from abstract syntax tree in linear time
  - Automatically taken care of by the front-end
- Usage: store analysis results in CFG nodes

CFG with Basic Blocks

- Stores basic blocks in a single node
- Extended blocks – maximal connected loop-free subgraphs



Exercise: apply algorithm

```
{              }
y := a+b
{              }
x := y
{                     }
while (x≠z) do
    {                                      }
    w := a+b
    {                                      }
    x := a+b
    {                                      }
    a := z
    {                                      }
```

Step 1/18

```
{}
y := a+b
{ y=a+b }*
x := y
while (x≠z) do
    w := a+b
    x := a+b
    a := z
```

Not all factoids are shown – apply *Explicate* to get all factoids

Step 2/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
while (x≠z) do
    w := a+b
    x := a+b
    a := z
```

Step 3/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv' = { y=a+b, x=y, x=a+b }*
while (x≠z) do
    w := a+b
    x := a+b
    a := z
```

Step 4/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv' = { y=a+b, x=y, x=a+b }*
while (x≠z) do
    { y=a+b, x=y, x=a+b }*
    w := a+b
    x := a+b
    a := z
```

Step 5/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv’ = { y=a+b, x=y, x=a+b }*
while (x≠z) do
    { y=a+b, x=y, x=a+b }*
    w := a+b
    { y=a+b, x=y, x=a+b, w=a+b, w=x, w=y }*
    x := a+b
    a := z
```

Step 6/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv' = { y=a+b, x=y, x=a+b }*
while (x≠z) do
    { y=a+b, x=y, x=a+b }*
    w := a+b
    { y=a+b, x=y, x=a+b, w=a+b, w=x, w=y }*
    x := a+b
    { y=a+b, w=a+b, w=y, x=a+b, w=x, x=y }*
    a := z
```

Step 7/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv' = { y=a+b, x=y, x=a+b }*
while (x≠z) do
    { y=a+b, x=y, x=a+b }*
    w := a+b
    { y=a+b, x=y, x=a+b, w=a+b, w=x, w=y }*
    x := a+b
    { y=a+b, w=a+b, w=y, x=a+b, w=x, x=y }*
    a := z
    { w=y, w=x, x=y, a=z }*
```

Step 8/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv'' = { x=y }*
while (x≠z) do
    { y=a+b, x=y, x=a+b }*
    w := a+b
    { y=a+b, x=y, x=a+b, w=a+b, w=x, w=y }*
    x := a+b
    { y=a+b, w=a+b, w=y, x=a+b, w=x, x=y }*
    a := z
    { w=y, w=x, x=y, a=z }*
```

Step 9/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv'' = { x=y }*
while (x≠z) do
    { x=y }*
    w := a+b
    { y=a+b, x=y, x=a+b, w=a+b, w=x, w=y }*
    x := a+b
    { y=a+b, w=a+b, w=y, x=a+b, w=x, x=y }*
    a := z
    { w=y, w=x, x=y, a=z }*
```

Step 10/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv'' = { x=y }*
while (x≠z) do
    { x=y }*
    w := a+b
    { x=y, w=a+b }*
    x := a+b
    { y=a+b, w=a+b, w=y, x=a+b, w=x, x=y }*
    a := z
    { w=y, w=x, x=y, a=z }*
```

Step 11/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv'' = { x=y }*
while (x≠z) do
    { x=y }*
    w := a+b
    { x=y, w=a+b }*
    x := a+b
    { x=a+b, w=a+b, w=x }*
    a := z
    { w=y, w=x, x=y, a=z }*
```

Step 12/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv'' = { x=y }*
while (x≠z) do
   { x=y }*
   w := a+b
   { x=y, w=a+b }*
   x := a+b
   { x=a+b, w=a+b, w=x }*
   a := z
   { w=x, a=z }*
```

Step 13/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv''' = { }
while (x≠z) do
    { x=y }*
    w := a+b
    { x=y, w=a+b }*
    x := a+b
    { x=a+b, w=a+b, w=x }*
    a := z
    { w=x, a=z }*
```

Step 14/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv''' = { }
while (x≠z) do
    { }
    w := a+b
    { x=y, w=a+b }*
    x := a+b
    { x=a+b, w=a+b, w=x }*
    a := z
    { w=x, a=z }*
```

Step 15/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv''' = { }
while (x≠z) do
    { }
    w := a+b
    { w=a+b }*
    x := a+b
    { x=a+b, w=a+b, w=x }*
    a := z
    { w=x, a=z }*
```

Step 16/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv''' = { }
while (x≠z) do
    { }
    w := a+b
    { w=a+b }*
    x := a+b
    { x=a+b, w=a+b, w=x }*
    a := z
    { w=x, a=z }*
```

Step 17/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv''' = { }
while (x≠z) do
    { }
    w := a+b
    { w=a+b }*
    x := a+b
    { x=a+b, w=a+b, w=x }*
    a := z
    { w=x, a=z }*
```

Step 18/18

```
{}
y := a+b
{ y=a+b }*
x := y
{ y=a+b, x=y, x=a+b }*
Inv = { }
while (x≠z) do
    { }
    w := a+b
    { w=a+b }*
    x := a+b
    { x=a+b, w=a+b, w=x }*
    a := z
    { w=x, a=z }*
{ x=z }
```