Formal Methods in Software Engineering

An Introduction to Model-Based Analyis and Testing

Vesal Vojdani

Fall 2015

1 Introduction

Software quality and FM

- Goal: Increased confidence in software!
- We explain our *intentions* to The Machine.
- The Machine helps us check if they're satisfied.
- According to RTCA DO-333:

$$\text{formal method} = \text{formal model} + \text{formal analysis}$$

What is a formal model?

A model is formal if it has...

- Well-defined syntax.
- Unambiguous (mathematical) semantics.
- The Machine must truly grok it.

Formal Analysis

1. Deductive Methods
2. Model Checking
3. Abstract & Symbolic Execution

In General: Satisfiability

$$\mathcal{M} \models \varphi$$

- $\mathcal{M}$: a *model* of the system
- φ: a *specification* of what is expected of $\mathcal{M}$

1.1 Deductive Methods

Deductive Methods

- Describe the system as set of logical formulas Γ.
- The specification is another formula φ.
- Verify by finding a proof of
$$\Gamma \vdash \varphi$$
- Thus, any $\mathcal{M} \models \Gamma$, will also satisfy φ.
- Deductive methods typically require some guidance from the expert (you).

Example: Avoid division by Zero

$$\begin{aligned}
&\quad (\!| y \neq 1 |\!) \\
&\quad (\!| y - 1 \neq 0 |\!) \\
&x := y \\
&\quad (\!| x - 1 \neq 0 |\!) \\
&x := x - 1 \\
&\quad (\!| x \neq 0 |\!) \\
&z := y/x
\end{aligned}$$

Any model (initial state) where $y \neq 1$ is safe.

1.2 Model-Checking

Model Checking

- $\mathcal{M}$ is given in a dedicated modelling language, based on say transition systems.
- The specification is a formula φ in temporal logic.
- Verify by checking if we have
$$\mathcal{M} \models \varphi$$
by exhaustive exploration of the model.

Why create a model?

- The model can be fairly simple, but ...
- execution may be complex (*concurrency*!)

- Visualize and explore the model manually.
- Automatically check for typical safety and liveness properties.

- Models can also be used for test-generation.

Model-Based Testing

- Automatic test generation requires an *oracle*.
- The model can be used to automatically generate unit tests with the required assertions.
- We have model-based *coverage* criteria.

- Testing is the only way to test the entire system (including hardware, network, environment).
- Here, we are interested in validating the test suite.
- Versus a finite model, testing can be complete!

1.3 Abstraction

If the model is too complicated

- Some models cannot be analyzed directly.
- (Source code itself is a formal model!)
- How can we make the model smaller?

- We can explore the model up to a finite bound.
- We can explore the model symbolically.
- We can evaluate the model abstractly.

Bounds versus Abstraction

- What is the problem with bounded models?
- In contrasts, abstraction over-approximates:
- Instead of $\mathcal{M}$, we consider $\mathcal{M}'$, such that

$$\mathcal{M}' \models \varphi \implies \mathcal{M} \models \varphi$$

- Abstract interpretation is a lattice-based theory of approximation.

1.4 This Course

Course Outline

- Run-time checking of assertions/contracts.
  - We begin with simple assertions for debugging.
  - Looking at pre- and post-conditions (contracts).
  - We'll see how these improve run-time checking.

- Static contract verification.
  - Proving that the program satisfies the contract in different systems.
  - Kalmer will give a brief tutorial on completing proofs manually a proof-assistant.

- Automated verification techniques.
  - Automated theorem provers.
  - Abstraction-based analyzers.
  - Configuring these tools require some deeper knowledge of how they actually work.

- Creating models of systems.
  - I want to spend proper time on the SPIN model checker.
  - We should also look into Alloy.

- Test Generation
  - Model-based testing.
  - Symbolic execution & concolic testing.

Course Organization

- We want to learn to use these tools properly.
- I will do very much a reverse classroom approach.

- You will be given assignments.
- We solve and discuss them at class.
- You must attend roughly once per week to show your work!

- Grading will be based on coursework. (70p)
- You will present a project in an oral exam. (30p)

2 Hoare Logic

Hoare Triplets

$$(\!|\, \phi \,|\!)\ S\ (\!|\, \psi \,|\!)$$

- A Hoare triple is satisfied under partial correctness:
  - for each state satisfying ϕ,
  - if execution reaches the end of S,
  - the resulting state satisfies ψ.
- (Total correctness: partial + termination)

Simple Language: Expressions

$$\begin{array}{rll} e ::= & x & \text{variable } x \in V \\ | & c & \text{constant } c \in \mathbb{Z} \\ | & e_1 + e_2 \mid \ldots & \text{arithmetics} \\ b ::= & \mathit{true} \mid \mathit{false} & \text{booleans} \\ | & b_1 \wedge b_2 \mid \ldots & \text{logical operators} \\ | & e_1 \leq e_2 \mid \ldots & \text{comparisons} \end{array}$$

Simple Language: Statements

$$\begin{aligned} S ::= & \ S_1 \ ; S_2 \\ | & \ x := e \\ | & \ \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \\ | & \ \textsf{while } b \textsf{ do } S \\ | & \ \textsf{skip} \\ | & \ \{S\} \end{aligned}$$

FOL (quantification over variables)

- Our language for reasoning:

$$\begin{array}{rll} \phi ::= & b & \text{boolean expression} \\ | & \phi_1 \wedge \phi_2 & \text{conjunction} \\ | & \phi_1 \vee \phi_2 & \text{disjunction} \\ | & \phi_1 \to \phi_2 & \text{implication} \\ | & \exists y : \phi & \text{existential quantification} \\ | & \forall y : \phi & \text{universal quantification} \end{array}$$

- We do not actually evaluate ϕ at runtime.
- Can only evaluate b (no quantifiers).

Composition

$$\frac{(\!|\ \phi\ |\!)\ S_1\ (\!|\ \eta\ |\!) \qquad (\!|\ \eta\ |\!)\ S_2\ (\!|\ \psi\ |\!)}{(\!|\ \phi\ |\!)\ S_1 \ ; S_2\ (\!|\ \psi\ |\!)}$$

Assignment

$$\frac{}{(\!|\ \psi[e/x]\ |\!)\ x := e\ (\!|\ \psi\ |\!)}$$

- Is this backwards?
- Consider examples for $x := 2$ and $x := x + 1$.

Conditional Statements

$$\frac{(\!|\, \phi \wedge b \,|\!)\ S_1\ (\!|\, \psi \,|\!) \qquad (\!|\, \phi \wedge \neg b \,|\!)\ S_2\ (\!|\, \psi \,|\!)}{(\!|\, \phi \,|\!)\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ (\!|\, \psi \,|\!)}$$

While Statements

$$\frac{(\!|\, \phi \wedge b \,|\!)\ S\ (\!|\, \phi \,|\!)}{(\!|\, \phi \,|\!)\ \mathsf{while}\ b\ \mathsf{do}\ S\ (\!|\, \phi \wedge \neg b \,|\!)}$$

Implication

$$\frac{\phi' \Rightarrow \phi \qquad (\!|\, \phi \,|\!)\ S\ (\!|\, \psi \,|\!) \qquad \psi \Rightarrow \psi'}{(\!|\, \phi' \,|\!)\ S\ (\!|\, \psi' \,|\!)}$$

- These end up as *verification conditions*.
- Automated theorem provers have to dismiss them.

Hello World!

```
int abs(int i) {
    if (0 <= i)
        r := i;
    else
        r := -i;
}
```

- Prove: always returns a non-negative value.
- (Where exactly would an overflow invalidate this proof?)

Step by step

1. We first have the conditional:

$$\frac{(\!|\, 0 \leq i \,|\!)\ r := i\ (\!|\, 0 \leq r \,|\!) \qquad (\!|\, i < 0 \,|\!)\ r := -i\ (\!|\, 0 \leq r \,|\!)}{(\!|\, \mathit{true} \,|\!)\ \mathsf{if}\ 0 \leq i\ \mathsf{then}\ r := i\ \mathsf{else}\ r := -i\ (\!|\, 0 \leq r \,|\!)}$$

2. The true-branch follows from the assignment axiom.
3. The false-branch relies on a simple implication:

$$\frac{i < 0 \Rightarrow 0 \leq -i \qquad (\!|\, 0 \leq -i \,|\!)\ r := -i\ (\!|\, 0 \leq r \,|\!)}{(\!|\, i < 0 \,|\!)\ r := -i\ (\!|\, 0 \leq r \,|\!)}$$

Proof trees

$$\dfrac{(\!|\, 0 \le i \,|\!)\; r := i \;(\!|\, 0 \le r \,|\!) \qquad \dfrac{i < 0 \Rightarrow 0 \le -i \qquad (\!|\, 0 \le -i \,|\!)\; r := -i \;(\!|\, 0 \le r \,|\!)}{(\!|\, i < 0 \,|\!)\; r := -i \;(\!|\, 0 \le r \,|\!)}}{(\!|\, \mathit{true} \,|\!)\; \textsf{if } 0 \le i \textsf{ then } r := i \textsf{ else } r := -i \;(\!|\, 0 \le r \,|\!)}$$

- The sequential application of inference rules are often represented as *proof trees*.
- These trees can grow large. . .
- Instead: annotate the program code!Tree structure is implicit.

Tableaux Proofs

$$\begin{array}{ll}
 & (\!|\, \phi_0 \,|\!) \\
S_1 \;; & \\
 & (\!|\, \phi_1 \,|\!) \\
S_2 \;; & \\
 & (\!|\, \phi_2 \,|\!) \\
\vdots & \\
 & (\!|\, \phi_{n-1} \,|\!) \\
S_n & \\
 & (\!|\, \phi_n \,|\!)
\end{array}$$

Tableaux: Composition

$$\dfrac{(\!|\, \phi \,|\!)\; S_1 \;(\!|\, \eta \,|\!) \qquad (\!|\, \eta \,|\!)\; S_2 \;(\!|\, \psi \,|\!)}{(\!|\, \phi \,|\!)\; S_1 \;;\; S_2 \;(\!|\, \psi \,|\!)}$$

$$\begin{array}{ll}
 & (\!|\, \phi \,|\!) \\
S_1 \;; & \\
 & (\!|\, \eta \,|\!) \\
S_2 & \\
 & (\!|\, \psi \,|\!)
\end{array}$$

Tableaux: Conditional

$$\frac{(\!|\,\phi \wedge b\,|\!)\ S_1\ (\!|\,\psi\,|\!) \qquad (\!|\,\phi \wedge \neg b\,|\!)\ S_2\ (\!|\,\psi\,|\!)}{(\!|\,\phi\,|\!)\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ (\!|\,\psi\,|\!)}$$

$(\!|\,\phi\,|\!)$

$\mathsf{if}\ b\ \mathsf{then}\ \{$

$(\!|\,\phi \wedge b\,|\!)$

S_1

$\}\ \mathsf{else}\ \{$

$(\!|\,\phi \wedge \neg b\,|\!)$

S_2

$\}$

$(\!|\,\psi\,|\!)$

Tableaux: Implication

$$\frac{\phi' \Rightarrow \phi \qquad (\!|\,\phi\,|\!)\ S\ (\!|\,\psi\,|\!) \qquad \psi \Rightarrow \psi'}{(\!|\,\phi'\,|\!)\ S\ (\!|\,\psi'\,|\!)}$$

$(\!|\,\phi'\,|\!)$

$(\!|\,\phi\,|\!)$

S

$(\!|\,\psi\,|\!)$

$(\!|\,\psi'\,|\!)$

The example as tableaux proof

$$
\begin{array}{l}
\quad (\!|\, \mathit{true} \,|\!) \\
\mathsf{if}\ (0 \le i)\ \mathsf{then}\ \{ \\
\qquad\quad (\!|\, \mathit{true} \wedge 0 \le i \,|\!) \\
\quad r := i \\
\qquad\quad (\!|\, 0 \le r \,|\!) \\
\}\ \mathsf{else}\ \{ \\
\qquad\quad (\!|\, \mathit{true} \wedge i < 0 \,|\!) \\
\qquad\quad (\!|\, 0 \le -i \,|\!) \\
\quad r := -i \\
\qquad\quad (\!|\, 0 \le r \,|\!) \\
\} \\
\quad (\!|\, 0 \le r \,|\!)
\end{array}
$$

2.1 Weakest Pre-Conditions

Weakest Pre-Conditions

- We have so far only rules for *valid* Hoare triples.
- Not all triples are equally useful

$$(\!|\, \mathit{false} \,|\!)\ S\ (\!|\, \psi \,|\!)$$

- How do we infer these triples?
- We will now move towards a more *syntax-driven* method to infer *weakest* pre-conditions.

Definition

- We say ϕ is *weaker* than ϕ' if

$$\phi' \Rightarrow \phi$$

- For $\phi = \mathsf{WP}\,[\![S]\!]\ \psi$, we have

$$(\!|\, \phi \,|\!)\ S\ (\!|\, \psi \,|\!) \text{ is valid}$$
$$\text{if } (\!|\, \phi' \,|\!)\ S\ (\!|\, \psi \,|\!) \text{ then } \phi' \Rightarrow \phi$$

- ψ holds after S iff ϕ holds before. (when the logic can describe program states uniquely)

Assignment

- Consider sequential composition:

$$\begin{aligned} &z := x; \\ &z := z + y; \\ &u := z \end{aligned}$$

- It suffices with definitions:

$$\begin{aligned} \mathsf{WP}\ [\![x := e]\!]\ \psi\ &= \psi[e/x] \\ \mathsf{WP}\ [\![S_1\ ;\ S_2]\!]\ \psi &= \mathsf{WP}\ [\![S_1]\!]\ (\mathsf{WP}\ [\![S_2]\!]\ \psi) \end{aligned}$$

A tableaux proof from WPs

$$\begin{aligned} &\ (\!|\, x + y = 42 \,|\!) \\ &z := x\ ; \\ &\ (\!|\, z + y = 42 \,|\!) \\ &z := z + y\ ; \\ &\ (\!|\, z = 42 \,|\!) \\ &u := z \\ &\ (\!|\, u = 42 \,|\!) \end{aligned}$$

Conditional

- Hoare logic:

$$\frac{(\!|\, \phi \wedge b \,|\!)\ S_1\ (\!|\, \psi \,|\!) \qquad (\!|\, \phi \wedge \neg b \,|\!)\ S_2\ (\!|\, \psi \,|\!)}{(\!|\, \phi \,|\!)\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ (\!|\, \psi \,|\!)}$$

- A more syntax-driven rule:

$$\frac{(\!|\, \phi_1 \,|\!)\ S_1\ (\!|\, \psi \,|\!) \qquad (\!|\, \phi_2 \,|\!)\ S_2\ (\!|\, \psi \,|\!)}{(\!|\, \phi' \,|\!)\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ (\!|\, \psi \,|\!)}$$

  where $\phi' = (b \to \phi_1) \wedge (\neg b \to \phi_2)$

Proof Tableaux for Conditional 2.0

$$
\begin{array}{l}
\qquad (\!|\, (b \to \mathsf{WP}\,[\![S_1]\!]\,\psi) \land (\neg b \to \mathsf{WP}\,[\![S_2]\!]\,\psi) \,|\!) \\
\mathsf{if}\ b\ \mathsf{then}\ \{ \\
\qquad\qquad (\!|\, \mathsf{WP}\,[\![S_1]\!]\,\psi \,|\!) \\
\qquad S_1 \\
\}\ \mathsf{else}\ \{ \\
\qquad\qquad (\!|\, \mathsf{WP}\,[\![S_2]\!]\,\psi \,|\!) \\
\qquad S_2 \\
\} \\
\qquad (\!|\, \psi \,|\!)
\end{array}
$$

The Example Again

$$
\begin{array}{l}
\qquad (\!|\, \mathit{true} \,|\!) \\
\qquad (\!|\, (0 \le i \to 0 \le i) \land (i < 0 \to 0 \le -i) \,|\!) \\
\mathsf{if}\ (0 \le i)\ \mathsf{then}\ \{ \\
\qquad\qquad (\!|\, 0 \le i \,|\!) \\
\qquad r := i \\
\}\ \mathsf{else}\ \{ \\
\qquad\qquad (\!|\, 0 \le -i \,|\!) \\
\qquad r := -i \\
\} \\
\qquad (\!|\, 0 \le r \,|\!)
\end{array}
$$

2.2 Loop Invariants

While Loops

- Recall the proof rule

$$\frac{(\!|\, \phi \land b \,|\!)\ S\ (\!|\, \phi \,|\!)}{(\!|\, \phi \,|\!)\ \mathsf{while}\ b\ \mathsf{do}\ S\ (\!|\, \phi \land \neg b \,|\!)}$$

- Given a ψ as post-condition. . .
- How can we apply this rule?
- What is the WP of a while loop?

Termination?

- Weakest Liberal Preconditions

$$wp\,[\![S]\!]\,\psi \equiv wp\,[\![S]\!]\,true \wedge wlp\,[\![S]\!]\,\psi$$

- We did not care about this distinction
  - Termination is an outdated concept. ;)
  - Only loops have different definitions.

WLP for while loops

- $\mathsf{WP}\,[\![\mathsf{while}\ b\ \mathsf{do}\ S]\!]\ \psi$?
- Unrolling the loop:

$$\begin{aligned} F_0 &= \mathsf{while}\ b\ \mathsf{do\ skip} \\ F_i &= \mathsf{if}\ b\ \mathsf{then}\ S\ ;\ F_{i-1}\ \mathsf{else\ skip} \end{aligned}$$

- WLP for "exiting the loop after at most i iterations in a state satisfying ψ":

$$\begin{aligned} L_0 &\equiv \psi \wedge \neg b \\ L_i &\equiv (\neg b \rightarrow \psi) \wedge (b \rightarrow \mathsf{WP}\,[\![S]\!]\ L_{i-1}) \end{aligned}$$

- We then define

$$\mathsf{WLP}\,[\![\mathsf{while}\ b\ \mathsf{do}\ S]\!]\ \psi = \exists i \in \mathbb{N} : L_i$$

- Not very practical. . .

Unrolling Example

$$\begin{aligned} &\mathsf{WLP}\,[\![\mathsf{while}\ x < 3\ \mathsf{do}\ x := x + 1]\!]\ (x = 3) \\ F_0 &= \mathsf{while}\ x < 3\ \mathsf{do\ skip} \\ F_1 &= \mathsf{if}\ x < 3\ \mathsf{then}\ x := x + 1\ ; \\ &\qquad (\mathsf{while}\ x < 3\ \mathsf{do\ skip})\ \mathsf{else\ skip} \\ F_2 &= \mathsf{if}\ x < 3\ \mathsf{then}\ x := x + 1\ ; \\ &\qquad (\mathsf{if}\ x < 3\ \mathsf{then}\ x := x + 1\ ; \\ &\qquad (\mathsf{while}\ x < 3\ \mathsf{do\ skip})\ \mathsf{else\ skip})\ \mathsf{else\ skip} \\ F_3 &= \ldots \end{aligned}$$

$$
\begin{aligned}
L_0 &\equiv x = 3 \\
L_1 &\equiv ((3 \leq x) \rightarrow (x = 3) \wedge \\
&\qquad ((x < 3) \rightarrow \mathsf{WLP}\, [\![x := x + 1]\!]\, (x = 3)) \\
&\equiv (x \leq 3) \wedge ((x < 3) \rightarrow (x = 2)) \\
&\equiv 2 \leq x \leq 3 \\
L_2 &\equiv (x \leq 3) \wedge ((x < 3) \rightarrow \mathsf{WLP}\, [\![x := x + 1]\!]\, L_1) \\
&\equiv (x \leq 3) \wedge ((x < 3) \rightarrow (1 \leq x \leq 2)) \\
&\equiv 1 \leq x \leq 3
\end{aligned}
$$

$$
\begin{aligned}
L_0 &\equiv 3 \leq x \leq 3 \\
L_1 &\equiv 2 \leq x \leq 3 \\
L_2 &\equiv 1 \leq x \leq 3 \\
L_3 &\equiv 0 \leq x \leq 3 \\
&\dots \\
L_i &\equiv 3 - i \leq x \leq 3 \exists i \in \mathbb{N} : L_i && \equiv x \leq 3
\end{aligned}
$$

Precondition of a While Loop

To push ψ up through $\mathsf{while}\ b\ \mathsf{do}\ S$:

1. Guess a potential invariant ϕ.
2. Make sure $\phi \wedge \neg b \implies \psi$.
3. Compute $\phi' = \mathsf{WLP}\, [\![S]\!]\, \phi$.
4. Check that $\phi \wedge b \implies \phi'$.
5. Then, ϕ is a pre-condition for ψ.

$$
\frac{(\!|\, \phi \wedge b \,|\!)\ S\ (\!|\, \phi \,|\!)}{(\!|\, \phi \,|\!)\ \mathsf{while}\ b\ \mathsf{do}\ S\ (\!|\, \phi \wedge \neg b \,|\!)}
$$

Proof Tableaux for Loops

$$
\begin{array}{l}
\quad\;\; (\!|\, \phi \,|\!) \\
\mathsf{while}\ b\ \mathsf{do}\ \{ \\
\qquad\quad (\!|\, \phi \wedge b \,|\!) \\
\qquad\quad (\!|\, \mathsf{WLP}\ [\![S]\!]\ \phi \,|\!) \\
\quad\;\; S \\
\qquad\quad (\!|\, \phi \,|\!) \\
\} \\
\quad\;\; (\!|\, \phi \wedge \neg b \,|\!) \\
\quad\;\; (\!|\, \psi \,|\!)
\end{array}
$$

For the Example with $\phi \equiv x \leq 3$

$$
\begin{array}{l}
\quad\;\; (\!|\, x \leq 3 \,|\!) \\
\mathsf{while}\ x < 3\ \mathsf{do}\ \{ \\
\qquad\quad (\!|\, (x \leq 3) \wedge (x < 3) \,|\!) \\
\qquad\quad (\!|\, x + 1 \leq 3 \,|\!) \\
\quad\;\; x := x + 1 \\
\qquad\quad (\!|\, x \leq 3 \,|\!) \\
\} \\
\quad\;\; (\!|\, (x \leq 3) \wedge (3 \leq x) \,|\!) \\
\quad\;\; (\!|\, x = 3 \,|\!)
\end{array}
$$

2.2.1 Exercise: Factorial

Exercise!

```
int fact(int x) {
    y = 1;
    z = 0;
    while (z != x) {
        z = z + 1;
        y = y * z;
    }
    return y;
}
```

Guessing the invariant

- Doing a trace:

iteration	x	y	z	B
0	6	1	0	*true*
1	6	1	1	*true*
2	6	2	2	*true*
3	6	6	3	*true*
4	6	24	4	*true*
5	6	120	5	*true*
6	6	720	6	*false*
i		$i!$	i	

- Formulate hypothesis: $y = z!$

Proof obligations

Want to establish $\psi \equiv y = x!$.

1. Our invariant $\phi \equiv y = z!$
2. Check that $\phi \wedge \neg(z \neq x) \implies \psi$.
3. Compute WLP of loop body:

$$\phi' \equiv y \cdot (z + 1) = (z + 1)!$$

4. Check if $\phi \wedge z \neq x \implies \phi'$.
5. Continue WLP computation with ϕ.

Writing the Tableaux Proof

$$
\begin{aligned}
&\quad (\!| \textit{true} |\!) \\
&\quad (\!| 1 = 0! |\!) \\
&y := 1 \,; \\
&\quad (\!| x = 0! |\!) \\
&z := 0 \,; \\
&\quad (\!| y = z! |\!) \\
&\textsf{while } z \neq x \textsf{ do } \{ \\
&\qquad\quad (\!| (y = z!) \wedge (z \neq x) |\!) \\
&\qquad\quad (\!| y \cdot (z+1) = (z+1)! |\!) \\
&\qquad z := z + 1 \,; \\
&\qquad\quad (\!| y \cdot z = z! |\!) \\
&\qquad y := y \cdot z \\
&\qquad\quad (\!| y = z! |\!) \\
&\} \\
&\quad (\!| (y = z!) \wedge (z = x) |\!) \\
&\quad (\!| x = y! |\!)
\end{aligned}
$$

Another Example

Consider now the following program.

$$
\begin{aligned}
&\quad (\!| \textit{true} |\!) \\
&x := 2 \cdot y \,; \\
&z := 0 \,; \\
&\textsf{while } z \neq x \textsf{ do } \{ \\
&\qquad z := z + 1 \,; \\
&\qquad x := x - 1 \\
&\} \\
&\quad (\!| z = y |\!)
\end{aligned}
$$

How do we infer an invariant for this program? Well, what is invariant here? Since the loop essentially "takes one from z and gives it to x", their sum $z + x$ must remain invariant. Initially, $z + x = y + y$, so we use $z + x = 2 \cdot y$ as our invariant. We then complete the proof:

$$
\begin{array}{l}
\quad (\!| \, \mathit{true} \, |\!) \\
\quad (\!| \, (2 \cdot y = 2 \cdot y) \, |\!) \\
x := 2 \cdot y \, ; \\
\quad (\!| \, (x = 2 \cdot y) \, |\!) \\
z := 0 \, ; \\
\quad (\!| \, (z + x = 2 \cdot y) \, |\!) \\
\textsf{while } z \neq x \textsf{ do } \{ \\
\qquad\quad (\!| \, (z + x = 2 \cdot y) \wedge (z \neq x) \, |\!) \\
\qquad\quad (\!| \, (z + 1) + (x - 1) = 2 \cdot y \, |\!) \\
\qquad z := z + 1 \, ; \\
\qquad\quad (\!| \, z + (x - 1) = 2 \cdot y \, |\!) \\
\qquad x := x - 1 \\
\qquad\quad (\!| \, z + x = 2 \cdot y \, |\!) \\
\} \\
\quad (\!| \, (z + x = 2 \cdot y) \wedge (z = x) \, |\!) \\
\quad (\!| \, z = y \, |\!)
\end{array}
$$

Proof Rule for Total Correctness

$$
\frac{(\!| \, \phi \wedge b \wedge (0 \leq V = V_0) \, |\!) \; S \; (\!| \, \phi \wedge (0 \leq V < V_0) \, |\!)}{(\!| \, \phi \wedge (0 \leq V) \, |\!) \; \textsf{while } b \textsf{ do } S \; (\!| \, \phi \wedge \neg b \, |\!)}
$$

- Requires a variant V.
- Must be bounded $0 \leq V$.
- Body must decrease it: $V < \mathit{old}(V)$.

Simple loop variant: $z - x$

$$
\begin{array}{l}
x := 0 \, ; \\
\quad (\!| \, (x \leq z) \wedge (0 \leq z - x) \, |\!) \\
\textsf{while } x < z \textsf{ do } \{ \\
\qquad\quad (\!| \, (x \leq z) \wedge (x < z) \wedge (0 \leq z - x = V_0) \, |\!) \\
\qquad\quad (\!| \, (x + 1 \leq z) \wedge (0 \leq z - (x + 1) < V_0) \, |\!) \\
\qquad x := x + 1 \\
\qquad\quad (\!| \, (x \leq z) \wedge (0 \leq z - x < V_0) \, |\!) \\
\} \\
\quad (\!| \, (x \leq z) \wedge (z \leq x) \, |\!)
\end{array}
$$

2.3 Challenge: MinSum

Challenge: Minimal-Sum Section

- Given an integer array $a[0], a[1], \ldots, a[n-1]$.
- A section of a is a continuous slice

$$a[i \ldots j] = a[i], a[i+1], \ldots, a[j-1]$$

  where $0 \leq i < j \leq n$.
- Section sum: $S_{i,j} = a[i] + \cdots + a[j-1]$.
- A minimal-sum section is a section $a[i \ldots j]$ s.t. for any other $a[i' \ldots j']$, we have $S_{i,j} \leq S_{i',j'}$.

What to do?

- Compute the sum of the minimal-sum sections in linear time.
- Prove that the code is correct!
- For example...
  - $[-1, 3, 15, -6, 4, -5]$ is -7 for $[-6, 4, -5]$.
  - $[-2, -1, 3, -3]$ is -3 for $[-2, -1]$ or $[-3]$.

The Program (in Java)

```java
int minsum(int a[]) {
    k = 1;
    t = a[0];
    s = a[0];
    while (k < n) {
        t = min(t + a[k], a[k]);
        s = min(s,t);
        k = k + 1;
    }
    return s;
}
```

Post-conditions

- The value s is smaller than the sum of any section.

$$\psi_1 = \forall i, j : 0 \leq i < j \leq n \rightarrow s \leq S_{i,j}$$

- There is a section whose sum is s

$$\psi_2 = \exists i, j : 0 \leq i < j \leq n \wedge s = S_{i,j}$$

Trying to prove ψ_1

- Suitable Invariant:

$$\psi_1 = \forall i, j : 0 \leq i < j \leq n \to s \leq S_{i,j}$$
$$I_1(s, k) = \forall i, j : 0 \leq i < j \leq k \to s \leq S_{i,j}$$

- Additional Invariant

$$I_2(t, k) = \forall i : 0 \leq i < k \to t \leq S_{i,k}$$

For the second invariant

- The assignment in the loop:

$$\begin{aligned} t &:= \min(t + a[k], a[k]); \\ k &:= k + 1; \end{aligned}$$

- Show that the invariant for t is maintained:

$$I_2(t, k) \wedge k < n \implies I_2(\min(t + a[k], a[k]), k + 1)$$

What do we have to prove?
We are given this:

$$\forall i : 0 \leq i < k \to t \leq S_{i,k}$$
$$k < n$$

We have to show this:

$$\forall i : 0 \leq i \leq k \to \min(t + a[k], a[k]) \leq S_{i,k+1}$$

You can split into two cases:

1. $i = k$ is trivial ($S_{k,k+1} = a[k]$).
2. $0 \leq i < k$.

The key step
We have to show this then:

$$\forall i : 0 \leq i < k \to \min(t + a[k], a[k]) \leq S_{i,k+1}$$

For any such i, we can compute:

$$\begin{aligned} \min(t + a[k], a[k]) &\leq t + a[k] \\ &\leq S_{i,k} + a[k] \\ &= S_{i,k+1} \end{aligned}$$

(Dafny needs help only with the final equality, see the distributive lemma from the Dafny tutorial.)

The Complete Lemma

- In the end, we have to prove that

$$I_1(s,k) \wedge I_2(t,k) \wedge k < n$$
$$\Longrightarrow$$
$$I_1(\min(s, (\min(t + a[k], a[k])), k+1) \wedge$$
$$I_2(\min(t + a[k], a[k]), k+1)$$

- Then we had the other post-condition (ψ_2)
- Do as much of the exercise as we can...
- In terms of Dafny code, very little remains! Disclaimer: This may not correlate with man hours.

3 VC generation

Purpose of this lecture

- Get an idea of how verification condition generation works.
- We consider the simplest possible implementation.
- This is based on early work on ESC/Java.
- We see some important concepts:
  - collecting semantics
  - constraint systems
  - abstraction

Quick: What is the Loop Invariant?

$$\begin{array}{l}
y := 5 \;;\; x := 0 \;; \\
\quad (\!|\, \phi \,|\!) \\
\textsf{while } x \neq 5 \textsf{ do } \{ \\
\qquad\quad (\!|\, \phi \wedge x \neq 5 \,|\!) \\
\qquad\quad (\!|\, \phi[x+1/x] \,|\!) \\
\qquad x := x + 1 \\
\qquad\quad (\!|\, \phi \,|\!) \\
\} \\
\quad (\!|\, \phi \wedge x = 5 \,|\!) \\
\quad (\!|\, x = y \,|\!)
\end{array}$$

Generating VCs

- *Non-trivial* loop-invariants must be supplied, but everything else automatic.
- Assume program is annotated with
  - Pre- & Post-conditions.
  - For every while-loop, a supposed loop-invariant.
- How do we check *automatically* that the implementation satisfies the contract?

Verification Conditions

- Consider the triplets:

$$\llparenthesis\, \phi \,\rrparenthesis\ C\ \llparenthesis\, \psi \,\rrparenthesis$$
$$\llparenthesis\, x = x' \,\rrparenthesis\ x := x - y\ \llparenthesis\, x + y = x' \,\rrparenthesis$$

- The verification conditions would be

$$\phi \to \mathsf{WP}\,[\![C]\!]\ \psi$$
$$(x = x') \to ((x - y) + y = x')$$

Asking an SMT Solver

- We then ask an SMT solver if the VC is true.

$$(x = x') \to ((x - y) + y = x')$$

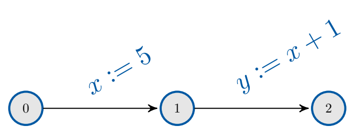
- We want the VC to hold for all parameters.
- Check if the negated formula is satisfiable!
- Think: searching for a falsifying assignment(failing test case).

3.1 Control Flow Graphs

Translation into Flow Graphs

Control Flow Graph $G = (N, E, s, r)$

- N are program points, and $s, r \in N$ are start/return nodes.
- $E = N \times C \times N$ are transition, where C is the set of basic statements (commands).



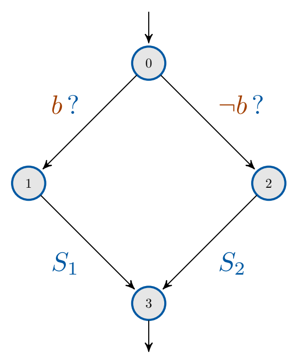
Basic Edges

$$\begin{array}{rlll} C & ::= & \mathsf{skip} & \text{skip} \\ & | & x := e & \text{assign} \\ & | & \phi\,? & \text{assume} \\ & | & \phi\,! & \text{assert} \end{array}$$

- Recall that ϕ may contain quantifiers.
- This translation is for VCG, not execution!

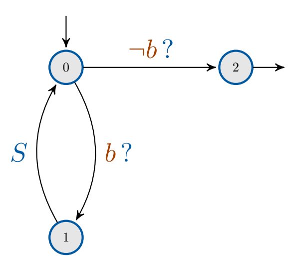
Translating If-Statements

$\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$



Translating While-Statements

$\mathsf{while}\ b\ \mathsf{do}\ S$



3.2 State and Satisfiability

Semantics

- We want to generate VC.
- Why not just show the algorithm?
- After all, I haven't bothered with semantics so far.
- Why stop now when we're having so much fun?
- Well, the constructs are now much less intuitive!
- It is time to assign *meanings* to our programs.

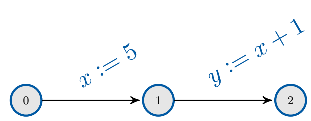
Program State

- A state σ assigns values to variables:

$$\sigma : V \to \mathbb{Z}$$

- Example:

$$\begin{aligned} \sigma_0 &= \{x \mapsto 0, y \mapsto 0\} \\ \sigma_1 &= \{x \mapsto 5, y \mapsto 0\} \\ \sigma_2 &= \{x \mapsto 5, y \mapsto 6\} \end{aligned}$$



Giving meaning to expressions

- For a state σ, we may evaluate expressions:

$$[\![e]\!]\sigma \in \mathbb{Z}$$

- And assign a truth-value to a formula:

$$[\![\phi]\!]\sigma \in \mathbb{B}$$

- For $\sigma = \{x \mapsto 5, y \mapsto 6\}$,

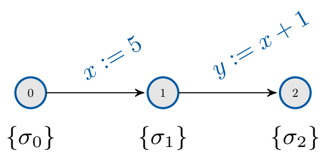
$$\begin{aligned} [\![x + y]\!]\sigma &= 11 \\ [\![x \le y]\!]\sigma &= \mathit{true} \end{aligned}$$

3.3 Collecting Semantics

Collecting Semantics

- For every point $p \in N$, we want to know
- The set of states reaching p: Σ_p.
- If we assume that $\Sigma_s = \Sigma_0 = \{\sigma_0\}$.

$$\sigma_0\ v = 0 \quad (\forall v \in V)$$



Starting State

- We need this semantics to validate our WP computation.
- Therefore, the best choice is $\Sigma_s = V \to \mathbb{Z}$, so that only tautologies hold at s.
- We include all logical variables from assume statements in V.

Quiz: The Error State

- For any Σ, what are the results of the edges?

$$\begin{array}{cc} \mathit{false}\,? & \mathit{false}\,! \\ \emptyset & \{\bot\} \end{array}$$

- However, $\bot$ should pass through other edges (like *exceptions* / maybe monad)

$$[\![\phi]\!]\,\bot = \mathit{false} \qquad\qquad \bot\,[x \mapsto e] = \bot$$

- We amend the assume rule...

Transfer functions

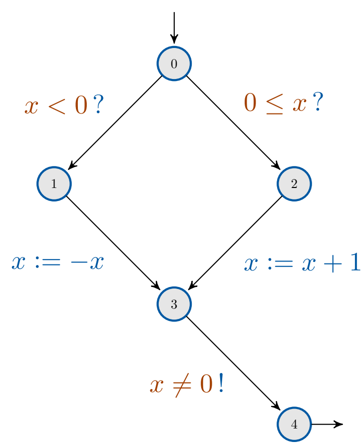
$$[\![\mathsf{skip}]\!]\,\Sigma = \Sigma$$

$$[\![x := e]\!]\,\Sigma = \{\sigma\,[x \mapsto [\![e]\!]\,\sigma] \mid \sigma \in \Sigma\}$$

$$\begin{aligned}[\![\phi\,?]\!]\,\Sigma = {} & \{\sigma \mid \sigma \in \Sigma,\ [\![\phi]\!]\,\sigma = \mathit{true}\} \\ & \cup \{\bot \mid \bot \in \Sigma\}\end{aligned}$$

$$\begin{aligned}[\![\phi\,!]\!]\,\Sigma = {} & \{\sigma \mid \sigma \in \Sigma,\ [\![\phi]\!]\,\sigma = \mathit{true}\} \\ & \cup \{\bot \mid \sigma \in \Sigma,\ [\![\phi]\!]\,\sigma = \mathit{false}\}\end{aligned}$$

Example



Equation & Constraint Systems

- Recall $G = (N, E, s, r)$.
- First we set the starting state:

  $$\Sigma_s = \{\sigma_s\} \qquad (\text{or } \Sigma_s = V \to \mathbb{Z})$$

  And for each point $q \in N$:

  $$\Sigma_q = \bigcup \{[\![C]\!]\,\Sigma_p \mid (p, C, q) \in E\}$$

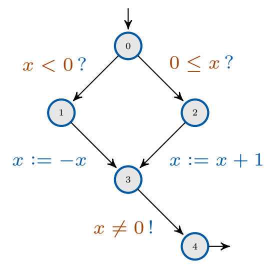
- As a constraint system:

  $$\begin{aligned} &\Sigma_s \supseteq \{\sigma_s\} \\ &\Sigma_q \supseteq [\![C]\!]\,\Sigma_p && \text{for } (p, C, q) \in E \end{aligned}$$

Constraint System Example

- Let $x_p = \{\sigma\, x \mid \sigma \in \Sigma_p\}$ (and $\perp$ if $\sigma = \perp$).
- We start with $x_0 = x_s = \mathbb{Z}$.

$$\begin{aligned}
x_0 &\supseteq \mathbb{Z} \\
x_1 &\supseteq \{z \mid z \in x_0,\, z < 0\} \\
x_2 &\supseteq \{z \mid z \in x_0,\, 0 \leq z\} \\
x_3 &\supseteq \{-z \mid z \in x_1\} \\
x_3 &\supseteq \{z + 1 \mid z \in x_2\} \\
x_4 &\supseteq \{z \mid z \in x_3,\, z \neq 0\} \\
&\quad \cup \{\perp \mid z \in x_3,\, z = 0\}
\end{aligned}$$



3.4 VC Generation

And Now WP...

$$\begin{aligned}
\mathsf{WP}\,[\![\mathsf{skip}]\!]\,\psi &= \psi \\
\mathsf{WP}\,[\![x := e]\!]\,\psi &= \psi[e/x] \\
\mathsf{WP}\,[\![\phi\,?]\!]\,\psi &= \phi \rightarrow \psi \\
\mathsf{WP}\,[\![\phi\,!]\!]\,\psi &= \phi \wedge \psi
\end{aligned}$$

Quiz: Error State Again

- Recall our false assume/assert edges:

$$\begin{array}{cc}
\mathit{false}\,? & \mathit{false}\,! \\
\emptyset & \{\perp\}
\end{array}$$

- Now what is the WP for these?

$$\begin{array}{cc}
\mathsf{WP}\,[\![\mathit{false}\,?]\!]\,\psi & \mathsf{WP}\,[\![\mathit{false}\,!]\!]\,\psi \\
\mathit{true} & \mathit{false}
\end{array}$$

Equation system for WP

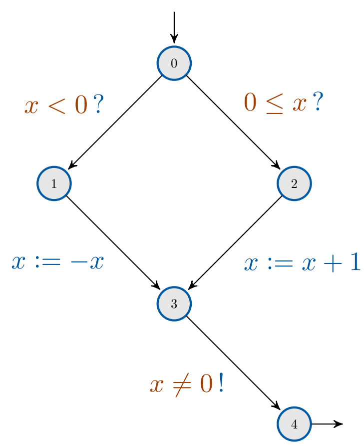
- We now start from the *end node* $r \in N$.
- Post-conditions are explicitly asserted, so...
- We start with $\psi_r = \mathit{true}$ and for $p \in N$:

$$\psi_p = \bigwedge \{\mathsf{WP}\, [\![c]\!]\, \psi_q \mid (p, c, q) \in E\}$$

- Alternatively, as a constraint system:

$$\begin{array}{lll} \psi_r & \Longrightarrow & \mathit{true} \\ \psi_p & \Longrightarrow & \mathsf{WP}\, [\![c]\!]\, \psi_q \qquad\qquad \text{for } (p, c, q) \in E \end{array}$$

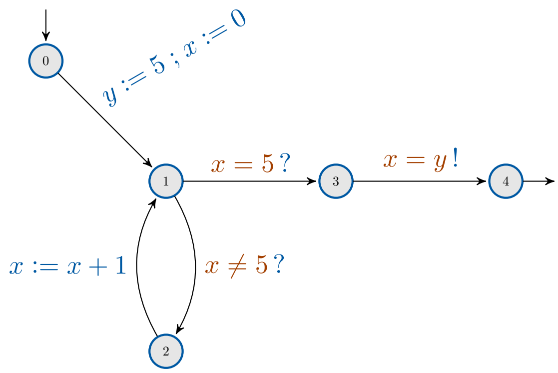
Again this example:



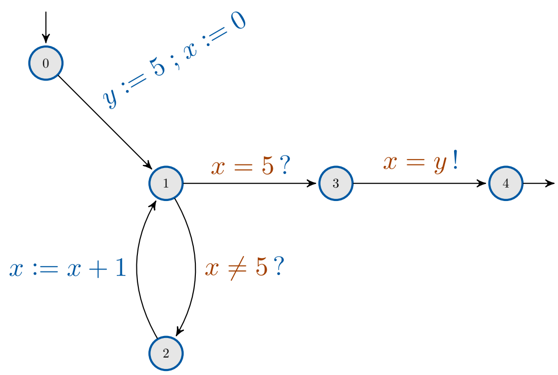
Now recall this example...

$y := 5\ ;$
$x := 0\ ;$
$\mathsf{while}\ x \neq 5\ \mathsf{do}$
$\quad x := x + 1\ ;$
$x = y\,!$

We could compute this...



WP computation was stuck in this loop



Havoc!

- Concrete semantics:

$$[\![\mathsf{havoc}\, x]\!]\, \Sigma = \{\sigma\, [x \mapsto z] \mid \sigma \in \Sigma,\, z \in \mathbb{Z}\}$$

- WP for havoc:

$$\mathsf{WP}\ [\![\mathsf{havoc}\, x]\!]\ \psi = \psi[x'/x] \qquad x' \text{ is fresh!}$$

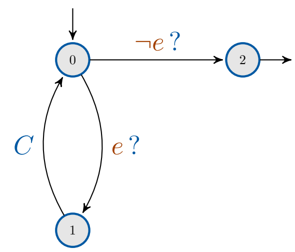
- We need ψ to hold for all values of x. Usually, we have assumes after havoc, so a typical example is

$$\mathsf{WP}\ [\![\mathsf{havoc}\, x]\!]\ ((y = x) \to (x = z)) \implies (y = z)$$

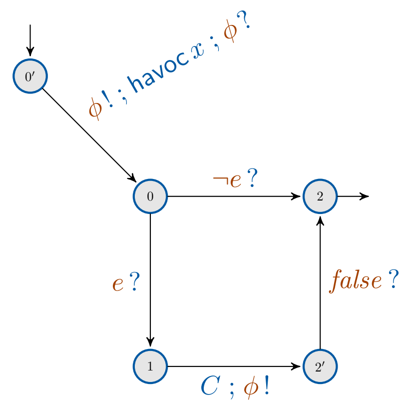
A simple assumption

- We should havoc all variables that are assigned to in the loop body.
- For simplicity, we assume this is only x.
- (You may think of x as a vector.)

Normal While Loop



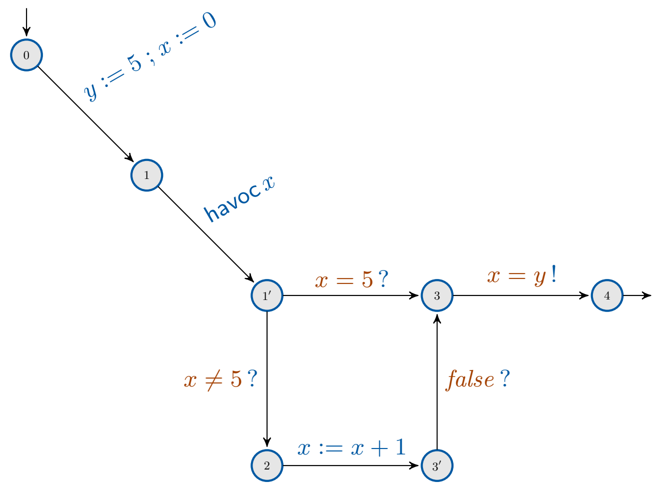
Abstraction using invariant ϕ



Why can we do this?

- The construction guarantees that if
$$\perp \notin \Sigma_2$$
we have
$$\Sigma_2' \subseteq \Sigma_2$$
where Σ_i' are the sets computed for the original while loop.
- Note: it follows very closely the proof rules of Hoare logic.

Now we really can compute a VC



What happened?

- Well, there was no invariant to check.
- That's good because the invariant was trivial.
- The homework requires making this construction with an invariant.
- There's just one more thing. . .

Procedure Calls

- Given a function P with parameter p and result r and contract

$$(\!|\, \phi \,|\!)\ P\ (\!|\, \psi \,|\!)$$

- We produce the following translation for a call $x = P(e)$.

$$\begin{aligned} & p := e \\ & \phi\,! \\ & \psi\,? \\ & x := r \end{aligned}$$

4 Data Flow Analysis

Data Flow Analysis

- We now consider how to check assertions using data flow analysis.
- Before we do that, we *must* to understand the basics of classical data flow analysis frameworks.
- We need to reason about soundness.
- Statements about programs are ordered...

Partial Orders

Definition
A set $\mathbb{D}$ together with a relation $\sqsubseteq$ is a *partial order* if for all $a, b, c \in \mathbb{D}$,

$$
\begin{array}{ll}
a \sqsubseteq a & \text{reflexivity} \\
a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b & \text{anti-symmetry} \\
a \sqsubseteq b \wedge b \sqsubseteq c \implies a \sqsubseteq c & \text{transitivity}
\end{array}
$$

Examples

1. $\mathbb{D} = 2^{\{a,b,c\}}$ with the relation "$\subseteq$"
2. $\mathbb{Z}$ with the relation "$=$"
3. $\mathbb{Z}$ with the relation "$\leq$"
4. $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ with the ordering:

$$x \sqsubseteq y \iff (x = \perp) \vee (x = y)$$

Facts about the program

- Our domain elements represent propositions about the program.
- Let $p \models x$ denote "x holds whenever execution reaches program point p".
- We order these propositions such that

$$x \sqsubseteq y \text{ whenever } (p \models x) \implies (p \models y)$$

- Consider examples:
  - The set of possibly live variables.
  - The set of definitely initialized variables.

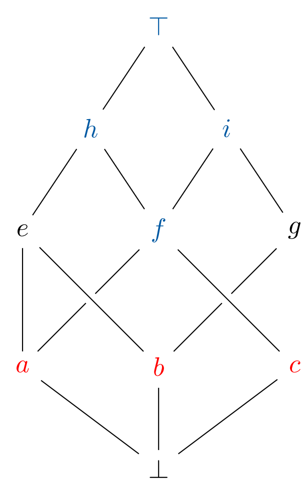
Combining information

- Assume there are two paths to reach p (true-branch and false-branch).
- If we have x along one path and y along the other, how can we combine this information?
$$x \sqcup y$$
- We want something that is true of both paths, and
- as precise as possible.

Least Upper Bounds

- $d \in \mathbb{D}$ is called an *upper bound* for $X \subseteq \mathbb{D}$ if
$$x \sqsubseteq d \qquad \text{for all } x \in X$$
- d is called a *least upper bound* if
  1. d is an upper bound and
  2. $d \sqsubseteq y$ for every upper bound y of X.

Do least upper bounds always exist?



Complete Lattice

Definition 1. A *complete lattice* $\mathbb{D}$ is a partial ordering where every subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$.

Every complete lattice has

- a *least* element $\bot = \bigsqcup \emptyset \in \mathbb{D}$;
- a *greatest* element $\top = \bigsqcup \mathbb{D} \in \mathbb{D}$.

Which are complete lattices?

1. $\mathbb{D} = 2^{\{a,b,c\}}$
2. $\mathbb{D} = \mathbb{Z}$ with "$=$".
3. $\mathbb{D} = \mathbb{Z}$ with "$\leq$".
4. $\mathbb{D} = \mathbb{Z}_\bot$.
5. $\mathbb{Z}_\bot^\top = \mathbb{Z} \cup \{\bot, \top\}$.

Solving constraint systems

- Recall the concrete semantics:

$$S_q \supseteq [\![c]\!]\, S_p \qquad\qquad \text{for } (p, c, q) \in E$$

- In general:

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

- We rewrite multiple constraints:

$$x \sqsupseteq d_1 \wedge \cdots \wedge x \sqsupseteq d_k \iff x \sqsupseteq \bigsqcup\{d_1, \ldots, d_k\}$$

So how to do it?

- In order to solve:

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

- We need f_i to be monotonic.
- A mapping f is *monotonic* if

$$a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$$

Monotonicity

- A mapping f is *monotonic* if

$$a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$$

- Which of the following is *not* monotonic?

$$\begin{array}{ll} \mathsf{inc}\ x\ = x + 1 & \mathsf{dec}\ x = x - 1 \\ \mathsf{top}\ x = \top & \mathsf{bot}\ x = \bot \\ \mathsf{id}\ x\ \ = x & \mathsf{inv}\ x\ = -x \end{array}$$

Vector function

- We want to solve:
$$x_i \sqsupseteq f_i(x_1, \ldots, x_n)$$
- Construct vector function $F\colon D^n \to D^n$
$$F(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$$
where $y_i = f_i(x_1, \ldots, x_n)$
- If f_i are monotonic, so is F.

Kleene iteration

- Successively iterate from $\perp$:
$$\perp, \quad F(\perp), \quad F^2(\perp), \quad \ldots$$
- Stop if we reach some $X = F^n(\perp)$ with
$$F(X) = X$$
- Will this terminate?
- Is this the *least* solution?

Simple Example

- For $\mathbb{D} = 2^{\{a,b,c\}}$
$$\begin{aligned} x_1 &\sqsupseteq \{a\} \cup x_3 \\ x_2 &\sqsupseteq x_3 \cap \{a, b\} \\ x_3 &\sqsupseteq x_1 \cup \{c\} \end{aligned}$$
- The Iteration

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	✓
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	✓
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	✓

Why Kleene iteration works

1. $\perp, F(\perp), F^2(\perp), \ldots$ is an *ascending chain*
$$\perp \sqsubseteq F(\perp) \sqsubseteq F^2(\perp) \sqsubseteq \cdots$$
2. If $F^k(\perp) = F^{k+1}(\perp)$, it is the *least* solution.
3. If all ascending chains in $\mathbb{D}$ are finite, Kleene iteration terminates.

Discussion

- What if $\mathbb{D}$ does contain infinite ascending chains?
- In particular, our concrete semantics was defined as the set of states with $\sigma \in V \to \mathbb{N}$.
- How do we know there aren't better solutions to the constraint system?

$$x = f(x) \qquad\qquad x \sqsupseteq f(x)$$

Answer to the first question

Theorem (Knaster-Tarski)
Assume $\mathbb{D}$ is a complete lattice. Then every monotonic function $f\colon \mathbb{D} \to \mathbb{D}$ has a least fixpoint $d_0 \in \mathbb{D}$ where

$$d_0 = \bigsqcap P \qquad\qquad P = \{d \in \mathbb{D} \mid d \sqsupseteq f(d)\}$$

1. Show that $d_0 \in P$.
2. Show that d_0 is a fixpoint.
3. Show that d_0 is the least fixpoint.

Answer to the second question

- Could there be better solutions to the constraint system than the least fixpoint?
- According to the theorem:

$$d_0 = \bigsqcap \{d \in \mathbb{D} \mid d \sqsupseteq f(d)\}$$

- Thus, d_0 is a lower bound for all solutions to the constraint system $d \sqsupseteq f(d)$.

Chaotic iteration

1. Set all x_i to $\bot$ and $W = \{1, \ldots, n\}$.
2. Take some $i \in W$ out of W. (if $W = \emptyset$, exit).
3. Compute $n := f_i(x_1, \ldots, x_n)$.
4. If $x_i \sqsupseteq n$, goto 2.
5. Set $x_i := x_i \sqcup n$ and reset $W := \{1, \ldots, n\}$.
6. Goto 2.

Data flow versus paths

- We want to verify that "whenever execution reaches program point p, a certain assertion holds."
- We need to check every *path* leading to p.
- Then: Why are we solving data flow constraint systems??

Path Semantics

- We define a path π inductively:

$$\begin{aligned} \pi &= \epsilon && \text{empty path} \\ \pi &= \pi' e && \text{where } e \in E \end{aligned}$$

- If π is a path from p to q, we write $\pi\colon p \to q$.
- We define the *path semantics*:

$$\begin{aligned} [\![\epsilon]\!]\, S &= S \\ [\![\pi(p, c, q)]\!]\, S &= [\![c]\!]\, ([\![\pi]\!]\, S) \end{aligned}$$

Merge Over All Paths

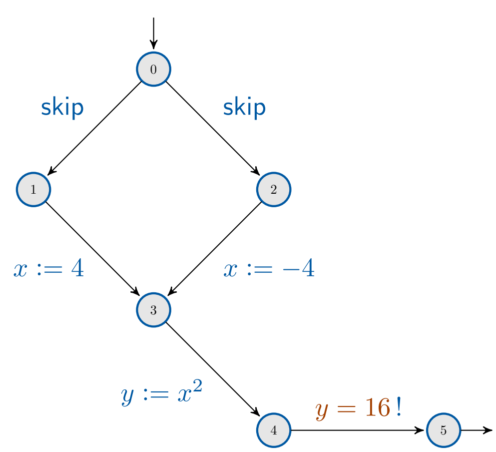
- For a complete lattice $\mathbb{D}$, we solved

$$\begin{aligned} x_s &\sqsupseteq d_s \\ x_q &\sqsupseteq [\![c]\!]\, x_p \quad (p, c, q) \in E \end{aligned}$$

- But we are really interested in:

$$y_p = \bigsqcup \{ [\![\pi]\!]\, d_s \mid \pi\colon s \to p \}$$

Example: Merge Over All Paths



When do solutions coincide?

- For our collecting semantics, they do.
- All functions $[\![c]\!]$ are *distributive*.
- In reality, we compute an abstract semantics.

$$\begin{aligned} x_s &\sqsupseteq d_s \\ x_q &\sqsupseteq [\![c]\!]^\sharp\, x_p \quad (p, c, q) \in E \end{aligned}$$

- Transfer functions $[\![c]\!]^\sharp \colon \mathbb{D} \to \mathbb{D}$ are monotonic.

Soundness of LFP Solutions

Theorem (Kam, Ullman, 1975)
Let x_i satisfy the following constraint system:

$$\begin{aligned} x_s &\sqsupseteq d_s \\ x_q &\sqsupseteq [\![c]\!]^\sharp\, x_p \quad (p, c, q) \in E \end{aligned}$$

where $[\![c]\!]^\sharp$ are monotonic. Then, for every $p \in N$, we have

$$x_p \sqsupseteq \bigsqcup \{ [\![\pi]\!]^\sharp\, d_s \mid \pi \colon s \to p \}$$

Intraprocedural Coincidence

Theorem (Kildall, 1972)
Let x_i satisfy the following constraint system:

$$\begin{aligned} x_s &\sqsupseteq d_s \\ x_q &\sqsupseteq [\![c]\!]^\sharp\, x_p \quad (p, c, q) \in E \end{aligned}$$

where $[\![c]\!]^\sharp$ are distributive. Then, for every $p \in N$, we have

$$x_p = \bigsqcup \{ [\![\pi]\!]^\sharp\, d_s \mid \pi \colon s \to p \}$$

4.1 Assertion Checking

Assertion Checking

- Track values of variables.
- Combine with WP computation.
- Infer invariants for loops.

Value Domains

- Characterize the possible values of variables whenever we reach program point p.
- A non-relational value domain:

$$\mathbb{D} = V \to \mathbb{D}_z$$

- We consider two simple value domains:
    1. Kildall's constant propagation domain.
    2. The Interval Domain.

Non-relational Domains

- For a complete lattice $\mathbb{D}$ and finite set V,
- the set of functions $\mathbb{D} \to V$ with the point-wise ordering

$$f_1 \sqsubseteq f_2 \iff \forall v \in V : f_1(v) \sqsubseteq f_2(v)$$

  is also a complete lattice.
- For example: $\mathbb{D} = V \to 2^{\mathbb{Z}}$.

Abstract Evaluation

- Just like for concrete state $\sigma \in V \to \mathbb{Z}$:

$$\begin{aligned} [\![z]\!]\,\sigma &= z \\ [\![x]\!]\,\sigma &= \sigma\, x \\ [\![e_1 + e_2]\!]\,\sigma &= [\![e_1]\!]\,\sigma + [\![e_2]\!]\,\sigma \end{aligned}$$

- Now, we need *abstract* operators such that for $d \in \mathbb{D} = V \to \mathbb{D}_z$, we evaluate:

$$\begin{aligned} [\![z]\!]^\sharp\, d &= z^\sharp \\ [\![x]\!]^\sharp\, d &= d\, x \\ [\![e_1 + e_2]\!]^\sharp\, d &= [\![e_1]\!]^\sharp\, d +^\sharp [\![e_2]\!]^\sharp\, d \end{aligned}$$

What the domain must supply

1. Lattice operations.
2. Lifting of constants:

$$\forall z \in \mathbb{Z} : z^\sharp \in \mathbb{D}_z$$

3. Abstract operations:

$$\forall z_1, z_2 \in \mathbb{D}_z : z_1 +^\sharp z_2 \in \mathbb{D}_z$$

   (not just for $+$; also unary, comparisons, logical, etc.)

Kildall's Domain

1. Lattice is the flat lattice.
2. Constants are already elements of $\mathbb{D}_z$:

$$z^\sharp = z$$

3. Operators are essentially lifted:

$$a +^\sharp b = \begin{cases} \bot & \text{if } a = \bot \text{ or } b = \bot \\ \top & \text{if } a = \top \text{ or } b = \top \\ a + b & \text{otherwise} \end{cases}$$

(More precise, e.g., for multiplication?)

Interval Domain

1. Lattice is $\mathbb{Z} \times \mathbb{Z}$ with $\langle l_1, u_1 \rangle \sqsubseteq \langle l_2, u_2 \rangle$ if

$$\langle l_2 \leq l_1 \rangle \wedge \langle u_1 \leq u_2 \rangle$$

2. Constants are singleton intervals:

$$z^\sharp = \langle z, z \rangle$$

3. Operators are generally defined as:

$$\begin{aligned} \langle l_1, u_1 \rangle *^\sharp \langle l_2, u_2 \rangle &= \langle l, u \rangle \text{ where} \\ l &= \min\{a * b \mid a \in \{l_1, u_1\},\, b \in \{l_2, u_2\}\} \\ u &= \max\{a * b \mid a \in \{l_1, u_1\},\, b \in \{l_2, u_2\}\} \end{aligned}$$

The Analysis

- We define abstract transfer functions.
- The simple ones:

$$\begin{aligned} [\![\mathsf{skip}]\!]^\sharp\, d &= d \\ [\![x := e]\!]^\sharp\, d &= d\,[x \mapsto [\![e]\!]^\sharp\, d] \end{aligned}$$

- Much like the concrete semantics:

$$\begin{aligned} [\![\mathsf{skip}]\!]\, S &= S \\ [\![x := e]\!]\, S &= \{\sigma\,[x \mapsto [\![e]\!]\,\sigma] \mid \sigma \in S\} \end{aligned}$$

The Bottom Value

- The bottom element is the mapping

$$d\,v = \bot \ (\forall v \in V)$$

- As soon as $\exists v$ with $d\,v = \bot$, we would set all variables to $\bot$.
- This bottom value denotes non-reachability.
- All transfer functions let $\bot$ pass through (strict).
- Conceptually, we do not need $\bot$ in the value domain at all:

$$(V \to \mathbb{Z}^\top)_\bot \qquad \text{instead of} \qquad V \to \mathbb{Z}^\top_\bot$$

Boolean values

- Booleans are also handled by the value domain. (e.g., when analysing C, there is no other option.)
- We simply need representatives for true and false:

$$\mathit{true}^\sharp \in \mathbb{D}_z \qquad \mathit{false}^\sharp \in \mathbb{D}_z$$

- and abstract versions of the boolean operators.

Assume edges

- The concrete semantics:

$$\begin{aligned}[\![e\,?]\!]\, S = &\{\sigma \mid \sigma \in S_p,\ [\![e]\!]\,\sigma = \mathit{true}\}\\ &\cup \{\bot \mid \bot \in S_p\}\end{aligned}$$

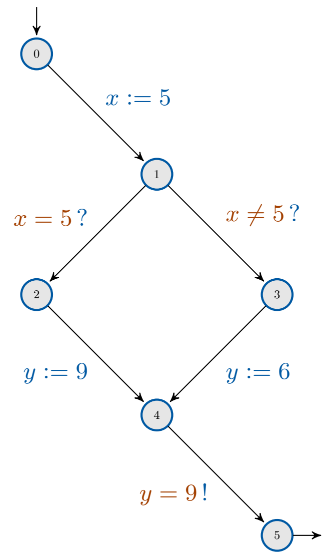
- We will handle errors separately.
- Abstract value sets:

$$[\![e\,?]\!]^\sharp\, d = \begin{cases} \bot & \text{if } [\![e]\!]^\sharp\, d \sqsubseteq \mathit{false}^\sharp \\ d \sqcap d_t & \text{otherwise} \end{cases}$$

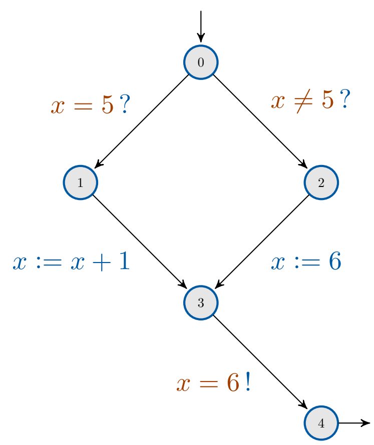
- This d_t depends on the domain, but must satisfy:

$$d_t \sqsupseteq \bigsqcup \mathsf{minimal\_elems}\,\{d \mid \mathit{true}^\sharp \sqsubseteq [\![e]\!]^\sharp\, d\}$$

Example 1: Dead Code



Example 2: Restricting Values



Correctness

- We have a monotonic *concretization* function γ.
- For the value domains $\gamma\colon \mathbb{D}_z \to 2^{\mathbb{Z}}$.

$$\gamma\ z = \begin{cases} \emptyset & \text{if } a = \bot \\ \mathbb{Z} & \text{if } a = \top \\ \{z\} & \text{otherwise} \end{cases}$$

- For the variable assignments:

$$\gamma\ d = \begin{cases} \emptyset & \text{if } \exists v : d\, v = \bot \\ \{\rho \mid \forall v : \rho\, v \in \gamma\,(d\, v)\} & \text{otherwise} \end{cases}$$

Correctness condition

- All our transfer functions need to satisfy:

$$[\![c]\!]\,(\gamma\, d) \sqsubseteq \gamma\,([\![c]\!]^\sharp\, d)$$

- Then, then the least solutions also satisfy:

$$S_p \subseteq \gamma\, x_p$$

- Because if we have $f(\gamma\, x) \sqsubseteq \gamma(f^\sharp\, x)$ and $d = f^\sharp\, d$, then

$$f(\gamma\, d) \sqsubseteq \gamma(f^\sharp\, d) = \gamma\, d$$

Assert edges

- Their effect on values is like assume:

$$\begin{aligned} [\![e\,!]\!]\, S = \{\sigma\ \mid \sigma \in S_p,\ [\![e]\!]\sigma \neq 0\} \\ \cup\ \{\bot \mid \sigma \in S_p,\ [\![e]\!]\sigma = 0\} \end{aligned}$$

- So how to check assertions? (next slide)
- Let x_p be the value analysis:

$$\begin{aligned} &x_0 \sqsupseteq d_0 \\ &x_q \sqsupseteq [\![c]\!]^\sharp\, x_p && \text{for } (p, c, q) \in E \end{aligned}$$

Assertion Checking

- We can just check for each assertion edge $(p, e\,!, q)$

$$1^\sharp \sqsubseteq [\![e]\!]^\sharp\, x_p$$

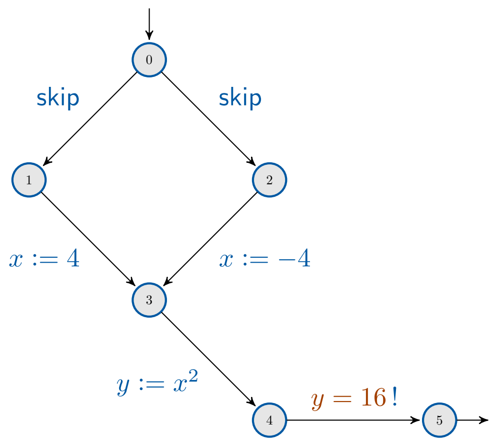
  If the above does not hold, the the assertion definitely fails.

- If we want to be sound:

$$[\![e]\!]^\sharp\, x_p \sqsubseteq 1^\sharp$$

  If this holds, the assertion is verified.

Example 3: Distributivity



Can we do better?

- We combine with WP computation.
- Recall the constraint system:

$$\phi_p \Rightarrow \mathsf{WP}\,[\![c]\!]\,\phi_q \qquad\qquad \text{for } (p, c, q) \in E$$

- What is the ordering of the domain?
- How do we combine?
- We can set up such a system for each assertion…

Discussion

- It is safe if we can only approximate implication.
- What is important for soundness?
- Our domain can be sets of conjucts.
- At program point p, we can safely dismiss a conjunct ϕ if

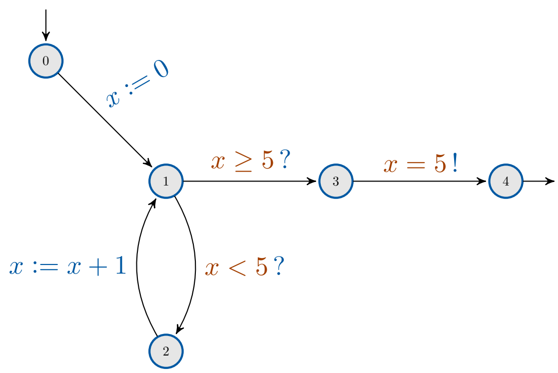
$$[\![\phi]\!]^\sharp\, x_p \sqsubseteq 1^\sharp$$

- If the solution for the system has $\phi_0 \equiv \mathit{true}$, we are happy.

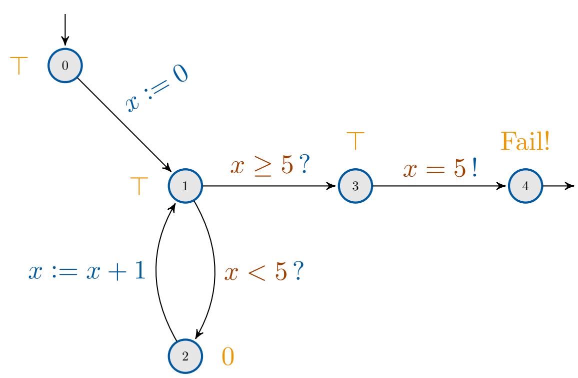
Conclusion

- This works for the simple example.
- WP computation would not terminate for a loop.
- Also, what is the concretization of this combined analysis?

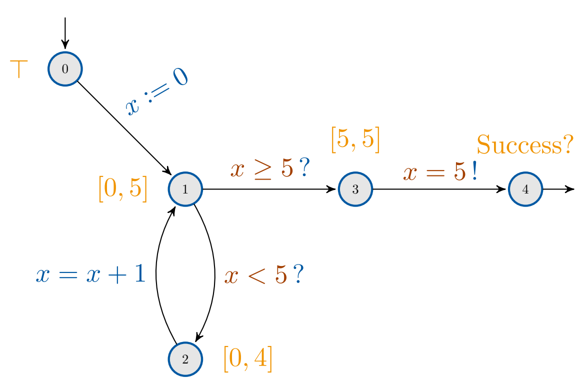
What about loops?



For the Kildall domain:



For the interval domain



Not really...

- This was not really static analysis.
- Termination not guaranteed.
- All ascending chains must stabilize.
- Enforce this by a *widening* operator ∇.
- Then, Kleene iteration will reach a (not necessarily least) fixpoint.

Widening
$\nabla : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is a widening operator if

1. $\forall x, y \in \mathbb{D} : (x \sqsubseteq x \nabla y) \wedge (y \sqsubseteq x \nabla y)$
2. for every chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$,

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= y_0 \nabla x_1 \\ y_2 &= y_1 \nabla x_2 \\ &\dots \end{aligned}$$

is not strictly increasing.

Iteration with widening

- Our non-terminating iteration:

$$\begin{aligned} x_0 &= \bot \\ x_{i+1} &= f(x_i) \end{aligned}$$

- Iteration with widening:

$$\begin{aligned} y_0 &= \bot \\ y_{i+1} &= \begin{cases} y_i & \text{if } f(y_i) \sqsubseteq y_i \\ y_i \nabla f(y_i) & \text{otherwise} \end{cases} \end{aligned}$$

Widening for Intervals

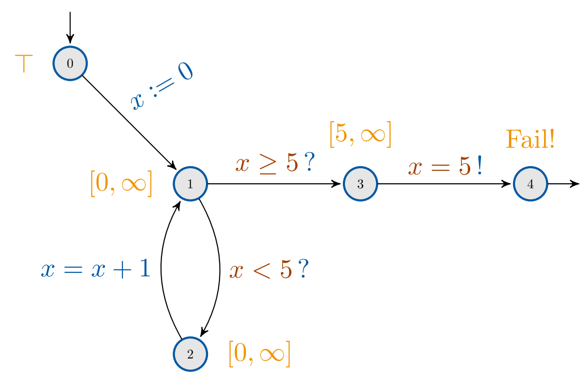
- $[l_1, u_1] \nabla [l_2, u_2] = [l, u]$ where

$$l = \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}$$

$$u = \begin{cases} u_1 & \text{if } u_2 \leq u_1 \\ \infty & \text{otherwise} \end{cases}$$

- This is not commutative
  - First argument: previous iteration.
  - Second argument: new value!
- Idea: give up if bounds are increasing.

Example with widening



Why did we fail?

- We are above the least solution.
- In particular, conditional constraints are over-approximated:

$$\begin{aligned} x_2 &\sqsupseteq [\![x < 5\,?]\!]^\sharp\, x_1 \\ [0, \infty] &\sqsupseteq [\![x < 5\,?]\!]^\sharp\, [0, \infty] \\ [0, \infty] &\sqsupseteq [0, 4] \end{aligned}$$

- Idea: why not just iterate a few times more?

Refining the solution

- Let x denote a solution to our constraint system:

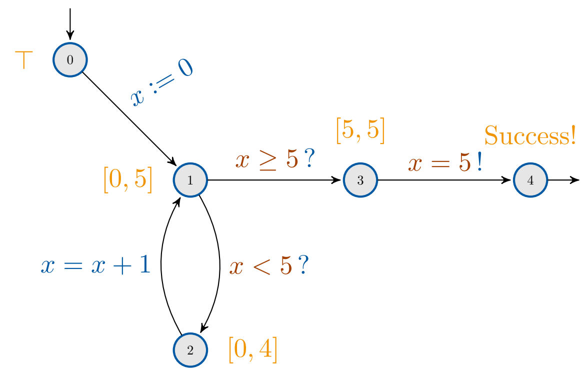
$$x \sqsupseteq f(x)$$

- If f is monotonic, then further iterations are all safe!

$$x \sqsupseteq f(x) \sqsupseteq f^2(x) \sqsupseteq \cdots$$

- We can stop after 5 minutes if we don't hit a fixpoint.

Post-fixpoint iteration



Success finally?

- Well, we were lucky and hit a fix-point.
- Termination for post-fixpoint iteration can be guaranteed.
- We require a narrowing operator $\triangle$.

Narrowing

$\triangle : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is a narrowing operator if

1. $\forall x, y \in \mathbb{D} : (y \sqsubseteq x) \implies (y \sqsubseteq x \triangle y \sqsubseteq x)$
2. for every chain $x_0 \sqsupseteq x_1 \sqsupseteq x_2 \sqsupseteq \cdots$,

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= y_0 \triangle x_1 \\ y_2 &= y_1 \triangle x_2 \\ &\cdots \end{aligned}$$

is not strictly decreasing.

Narrowing iteration

- Let x_0 be a solution, i.e.,

$$x_0 \sqsupseteq f(x_0)$$

- Post-fixpoint iteration with narrowing

$$\begin{aligned} y_0 &= x_0 \\ y_{i+1} &= y_i \triangle f(y_i) \end{aligned}$$

Narrowing for Intervals

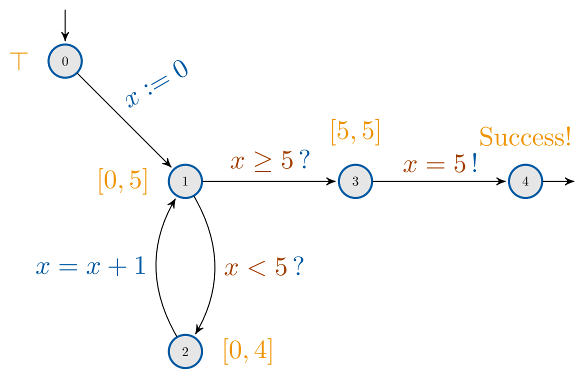
- $[l_1, u_1] \triangledown [l_2, u_2] = [l, u]$ where

$$l = \begin{cases} l_2 & \text{if } l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases}$$

$$u = \begin{cases} u_2 & \text{if } u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}$$

- Idea: Only restore lost bounds.

Replay with Widening/Narrowing



Conclusion

- This example does not require narrowings to enforce termination.
- Can you think of a simple modification to this example where narrowing would be essential?