Lecture 7
More Recursion. Bounded Model Checking

Viktor Kuncak

Summary: Least Fixpoint as Meaning of Recursion

A recursive program is a recursive definition of a relation $E(r) = r$

We define the intended meaning as $s = \bigcup_{i \geq 0} E(\emptyset)$, which satisfies $E(s) = s$ and also is the least among all relations r such that $E(r) \subseteq r$ (therefore, also the least among r for which $E(r) = r$)

We picked **least** fixpoint, so if the execution cannot terminate on a state x, then there is no x' such that $(x, x') \in s$.
This model is simple (just relations on states) though it has some limitations: let q be a program that never terminates, then

- $\rho(q) = \emptyset$ and $\rho(c \,\square\, q) = \rho(c) \cup \emptyset = \rho(c)$
  (we cannot observe optional non-termination in this model)
- also, $\rho(q) = \rho(\Delta_{\emptyset})$ (assume(false)), so the absence of results due to path conditions and infinite loop are represented in the same way

Alternative: error states for non-termination (we will not pursue)

Procedure Meaning is the Least Relation

```
def f =
  if (x > 0) {
    x = x - 1
    f
    y = y + 2
  }
```

$$\begin{aligned} E(r_f) = \ & (\Delta_{x \tilde{>} 0} \circ (\\ & \quad \rho(x = x - 1) \circ \\ & \quad r_f \circ \\ & \quad \rho(y = y + 2)) \\ &) \cup \Delta_{x \tilde{\leq} 0} \end{aligned}$$

What does it mean that $E(r) \subseteq r$?

Procedure Meaning is the Least Relation

```
def f =
  if (x > 0) {
    x = x − 1
    f
    y = y + 2
  }
```

$$E(r_f) = \Big(\Delta_{x \tilde{>} 0} \circ \big(\rho(x = x - 1) \circ r_f \circ \rho(y = y + 2)\big) \Big) \cup \Delta_{x \tilde{\leq} 0}$$

What does it mean that $E(r) \subseteq r$?
Plugging r instead of the recursive call results in something that conforms to r

Justifies modular reasoning for recursive functions

To prove that recursive procedure with body E satisfies specification r, show

- $E(r) \subseteq r$
- then because procedure meaning s is least, $s \subseteq r$

Proving that recursive function meets specification

Prove that if s is the relation denoting the recursive function below, then

$$((x, y), (x', y')) \in s \to y' \geq y$$

```
def f =
  if (x > 0) {
    x = x − 1
    f
    y = y + 2
  }
```

$$\begin{aligned} E(r_f) = \ & (\Delta_{x \tilde{>} 0} \circ (\\ & \quad \rho(x = x - 1) \circ \\ & \quad r_f \circ \\ & \quad \rho(y = y + 2)) \\ &) \cup \Delta_{x \tilde{\leq} 0} \end{aligned}$$

Proving that recursive function meets specification

Prove that if s is the relation denoting the recursive function below, then

$$((x,y),(x',y')) \in s \rightarrow y' \geq y$$

```
def f =
  if (x > 0) {
    x = x − 1
    f
    y = y + 2
  }
```

$$\begin{aligned} E(r_f) = \; & (\Delta_{x \tilde{>} 0} \circ (\\ & \quad \rho(x = x - 1) \circ \\ & \quad r_f \circ \\ & \quad \rho(y = y + 2)) \\ &) \cup \Delta_{x \tilde{\leq} 0} \end{aligned}$$

Solution: let specification relation be $q = \{((x,y),(x',y')) \mid y' \geq y\}$

Proving that recursive function meets specification

Prove that if s is the relation denoting the recursive function below, then

$$((x, y), (x', y')) \in s \rightarrow y' \geq y$$

```
def f =
  if (x > 0) {
    x = x − 1
    f
    y = y + 2
  }
```

$$\begin{aligned} E(r_f) = \; & (\Delta_{x \tilde{>} 0} \circ (\\ & \quad \rho(x = x - 1) \circ \\ & \quad r_f \circ \\ & \quad \rho(y = y + 2)) \\ &) \cup \Delta_{x \tilde{\leq} 0} \end{aligned}$$

Solution: let specification relation be $q = \{((x, y), (x', y')) \mid y' \geq y\}$
Prove $E(q) \subseteq q$ - given by a quantifier-free formula

Formula for Checking Specification

```
def f =
  if (x > 0) {
    x = x − 1
    f
    y = y + 2
  }
```

Specification: $q = \{((x, y), (x', y')) \mid y' \geq y\}$
Formula to prove, generated by representing $E(q) \subseteq q$:

$$\begin{array}{l}\big[(x > 0 \land x_1 = x - 1 \land y_1 = y \land y_2 \geq y_1 \land y' = y_2 + 2) \\ \lor(\neg(x > 0) \land x' = x \land y' = y)\big) \ \rightarrow \ y' \geq y\end{array}$$

- Because q appears as $E(q)$ and q, the condition appears twice.
- Proving $f \subseteq q$ by $E(q) \subseteq q$ is always sound, whether or not function f terminates; the meaning of f talks only about properties of terminating executions (relations can be partial)

Multiple Procedures: Functions on Pairs of Relations

Two mutually recursive procedures $r_1 = E_1(r_1, r_2), \quad r_2 = E_2(r_1, r_2)$
We extend the approach to work on pairs of relations:

$$(r_1, r_2) = (E_1(r_1, r_2), E_2(r_1, r_2))$$

Define $\bar{E}(r_1, r_2) = (E_1(r_1, r_2), E_2(r_1, r_2))$, let $\bar{r} = (r_1, r_2)$. We define semantics of procedures as the least solution of

$$\bar{E}(\bar{r}) = \bar{r}$$

where $(r_1, r_2) \sqsubseteq (r_1', r_2')$ means $r_1 \subseteq r_1'$ and $r_2 \subseteq r_2'$
Even though pairs of relations are not sets but pairs of sets, we can define set-like operations on them, e.g.

$$(r_1, r_2) \sqcup (r_1', r_2') = (r_1 \cup r_1',\ r_2 \cup r_2')$$

The entire theory works when we have a partial order $\sqsubseteq$ with some "good properties". (**Lattice** elements are a generalization of sets.)

Multiple Procedures: Least Fixedpoint and Consequences

Two mutually recursive procedures $r_1 = E_1(r_1, r_2), \quad r_2 = E_2(r_1, r_2)$
For $E(r_1, r_2) = (E_1(r_1, r_2), E_2(r_1, r_2))$, semantics is

$$(s_1, s_2) = \bigsqcup_{i \geq 0} \bar{E}^i(\emptyset, \emptyset)$$

It follows that for any c_1, c_2 if

$$E_1(c_1, c_2) \subseteq c_1 \;\text{ and }\; E_2(c_1, c_2) \subseteq c_2$$

then $s_1 \subseteq c_1$ and $s_2 \subseteq c_2$.

Induction-like principle: To prove that mutually recursive relations satisfy two contracts, prove those contracts for the relation body definitions in which recursive calls are replaced by those contracts.

Replacing Calls by Contracts: Example

```
def r1 = {
  if (x % 2 == 1) {
    x = x − 1
  }
  y = y + 2
  r2
} ensuring(y > old(y))
```

```
def r2 = {
  if (x != 0) {
    x = x / 2
    r1
  }
} ensuring(y >= old(y))
```

Replacing Calls by Contracts: Example

```
def r1 = {
  if (x % 2 == 1) {
    x = x − 1
  }
  y = y + 2
  r2
} ensuring(y > old(y))
```

```
def r2 = {
  if (x != 0) {
    x = x / 2
    r1
  }
} ensuring(y >= old(y))
```

Reduces to checking these two non-recursive procedures:

```
def r1 = {
  if (x % 2 == 1) {
    x = x − 1
  }
  y = y + 2
  { val x0 = x; y0 = y
    havoc(x,y)
    assume(y >= y0) }
} ensuring(y > old(y))
```

```
def r2 = {
  if (x != 0) {
    x = x / 2
    val x0 = x; y0 = y
    havoc(x,y)
    assume(y > y0)
  }
} ensuring(y >= old(y))
```

Bounded Model Checking and k-Induction

Concrete program semantics and verification

For each program there is a (monotonic, ω-continuous) function $F : C^n \to C^n$ such that

$$\bar{c}_* = \bigcup_{i \geq 0} F^i(\emptyset, \ldots, \emptyset)$$

describes the set of reachable states for each program point.
(Safety) verification can be stated as saying that the semantics remains within the set of good states G, that is $c_* \subseteq G$, or

$$\left(\bigcup_{i \geq 0} F^i(\emptyset, \ldots, \emptyset) \right) \subseteq G$$

which is equivalent to

$$\forall n.\ F^n(\emptyset, \ldots, \emptyset) \subseteq G$$

Unfolding for Counterexamples: Bounded Model Checking

$$\forall n.\ F^n(\emptyset, \dots, \emptyset) \subseteq G$$

The above condition is false iff there exists k and $\bar{c} \in C^n$ such that

$$\bar{c} \in F^k(\emptyset, \dots, \emptyset) \wedge \bar{c} \notin G$$

For a fixed k this can often be expressed as a quantifier-free formula.
Example: replace a loop $([c]s) * [!c]$ with finite unrolding $([c]s)^k[!c]$
Specifically, for $n = 1$, $S = \mathbb{Z}^2$, $C = 2^S$, and $F : C \to C$ describes the program: x=0;while(*)x=x+y

$$F(B) = \{(x, y) \mid x = 0\} \cup \{(x + y, y) \mid (x, y) \in B\}$$

We have $F(\emptyset) = \{(x, y) \mid x = 0\} = \{(0, y) \mid y \in \mathbb{Z}\}$

$$F^2(\emptyset) = \{(0, y) \mid y \in \mathbb{Z}\} \cup \{(y, y) \mid y \in \mathbb{Z}\}$$

$$F^3(\emptyset) = \{(x, y) \mid x = 0 \vee x = y \vee x = 2 * y\}$$

Formula for Bounded Model Checking

Let $P_B(x, y)$ be a formula in Presburger arithmetic such that $B = \{(x, y) \mid P_B(x, y)\}$ then the formula

$$x = 0 \vee (\exists x_0, y_0 . x = x_0 + y_0 \wedge y = y_0 \wedge P_B(x_0, y_0))$$

describes $F(B)$. Suppose the set $F^k(B)$ can be described by a PA formula P_k. If G is given by a formula P_G then the program can reach error in k steps iff

$$P_k \wedge \neg P_G$$

is satisfiable.
Suppose P_G is $x \leq y$. For $k = 3$ we obtain

$$(x = 0 \vee x = y \vee x = 2 * y) \wedge \neg(x \leq y)$$

By checking satisfiability of the formula we obtain counterexample values $x = -1, y = -2$.

Bounded Model Checking Algorithm

```
$B = \emptyset$
**while** $(*)$ {
  checksat(!($B \subseteq G$)) **match**
    **case** Assignment(v) **=>** **return** Counterexample(v)
    **case** Unsat **=>**
      $B' = F(B)$
      **if** $(B' \subseteq B)$ **return** Valid
      **else** $B = B'$
}
```

Good properties

- subsumes testing up to given depth for all possible initial states
- for a buggy program k, can be small, tools can find many bugs fast
- a semi-decision procedure for finding all error inputs

Bounded Model Checking is Bounded

Bad properties

- can prove correctness only if $F^{n+1}(\emptyset) = F^n(\emptyset)$ for a finite n
- errors after initializations of long arrays require unfolding for large n. This program requires unfolding past all loop iterations, even if the property does not depend on the loop:

```
i = 0
z = 0
while (i < 1000) {
  a(i) = 0
}
y = 1/z
```

- For large k formula F^k becomes large, so deep bugs are hard to find

Unfolding for Proving Correctness: k-Induction

$$\text{Goal:} \quad \forall n.\ F^n(\emptyset, \ldots, \emptyset) \subseteq G \tag{1}$$

Suppose that, for some $k \geq 1$

$$F^k(G) \subseteq G \tag{2}$$

By induction on p, for every $p \geq 1$,

$$F^{pk}(G) \subseteq G$$

By monotonicity of F, if $n \leq pk$ then

$$F^n(\bar{\emptyset}) \subseteq F^{pk}(\bar{\emptyset}) \subseteq F^{pk}(G) \subseteq G$$

Therefore, (1) holds.
Algorithm: check (2) for increasing $k \in \{1, 2, \ldots\}$

Summary: Using F^k for Proofs and Counterexamples

Exact semantics is: $\bigcup_{n\geq 0} F^n(\bar{\emptyset})$
Specification is G
If for some k:

- $\neg(F^k(\bar{\emptyset}) \subseteq G)$ then we prove that specification **does not** hold (and there is a "k-step" execution in $G \subseteq F^k(\bar{\emptyset})$ showing this)
- $F^k(G) \subseteq G$, then we prove that specification **holds** by showing that it holds in all base cases up to k and assuming it holds for all recursive steps at depth k and deeper (k-induction)

Least fixedpoint of F^k is the same as least fixedpoint of F: $F^i(\bar{\emptyset}) \subseteq F^{ki}(\bar{\emptyset})$, so $\bigcup$ gives same result as sequences are monotonic.
Each F^k defines the program with the meaning same as F but syntactically more obvious as k grows and we unfold more.

k-induction Algorithm

For monotonic F, prove or find counterexample for:

$$\forall n.\ F^n(\emptyset, \ldots, \emptyset) \subseteq G$$

```
$Fk = F$
**while** $(*)$ {
  checksat(!($Fk(G) \subseteq G$)) **match**
    **case** Unsat => **return** Valid
    **case** Assignment(v0) =>
      checksat(!($Fk(\emptyset) \subseteq G$)) **match**
        **case** Assignment(v) => **return** Counterexample(v)
        **case** Unsat => $Fk = Fk \circ F'$ // unfold one more
}
```

$F'(c)$ can be $F(c)$ or, thanks to previous checks, $F(c) \cap G$
Save work: preserve solver state in checksats across different k
Lucky test: if (!($lfp(F)(initState(v0)) \subseteq G$)) return Counterexample(v0)

Explanation for Sequences in k-Induction

$\bar{\emptyset} \subseteq F(\bar{\emptyset})$, so $F^i(\bar{\emptyset}) \subseteq F^{i+1}(\bar{\emptyset})$. We have an *ascending* sequence:

$$\bar{\emptyset} \subseteq F(\bar{\emptyset}) \subseteq F^2(\bar{\emptyset}) \subseteq \ldots \subseteq F^i(\bar{\emptyset}) \subseteq F^{i+1}(\bar{\emptyset}) \subseteq \ldots$$

In general, it need not be $G \subseteq F(G)$ nor $F(G) \subseteq G$.
Define $F'(c) = F(c) \cap G$. Clearly $F'(c) \subseteq F(c)$. Moreover,

$$c_1 \subseteq c_2 \rightarrow F'(c_1) \subseteq F'(c_2)$$

$$F'(G) = F(G) \cap G \subseteq G$$

So F' is monotonic and $F'(G) \subseteq G$. We have *descending* sequence:

$$\ldots \subseteq (F')^{i+1}(G) \subseteq (F')^i(G) \subseteq \ldots \subseteq F'(G) \subseteq G$$

Divergence in k-Induction

```
$Fk = F$
while $(*)$ {
  checksat($!(Fk(G) \subseteq G)$) match
    case Unsat => return Valid
    case Assignment(v0) =>
      checksat($!(Fk(\emptyset) \subseteq G)$) match
        case Assignment(v) => return Counterexample(v)
        case Unsat => $Fk = Fk \circ F'$ // unfold one more
}
```

Subsumes bounded model checking, so finds all counterexamples
But, it often *cannot* find proofs when $lfp(F) \subseteq G$. G may be too weak to be inductive, $(F')^n(G)$ may remain too weak:

$$F^n(\bar{\emptyset}) \subseteq lfp(F) \subseteq (F')^n(G) \subseteq F^n(G)$$

Need weakening of $F^n(\emptyset)$ or strengthening of $(F')^n(G)$

Approximate Postconditions

Suppose we did not find counterexample yet and we have sequence

$$c_0 \subseteq c_1 \subseteq \ldots c_k \subseteq G$$

where $c_i = F^i(\bar{\emptyset})$, so $F(c_i) = c_{i+1}$
Instead of simply increasing k, we try to obtain larger values by finding another sequence a_i satisfying $a_i \subseteq a_{i+1}$ and

$$F(a_i) \subseteq a_{i+1}$$

for $0 \leq i \leq k$, and with $a_k \subseteq G$.
$c_0 \subseteq a_0$ and, by induction, $c_i \subseteq a_i$
If $a_{i+1} = a_i$ for some i, then $F(a_i) = a_i$ so

$$lfp(F) \subseteq a_i \subseteq a_k \subseteq G$$

so we have proven $lfp(F) \subseteq G$, i.e., program satisfies spec.
We can also dually require $a_{i-1} \subseteq F(a_i)$, ensuring $a_i \subseteq F^{k-i}(G)$.

Abstract Interpretation

A Method for Constructing Inductive Invariants

Basic idea of abstract interpretation

Abstract interpretation is a way to infer properties of program computations.
Consider the assignment: $z = x + y$.

Interpreter:

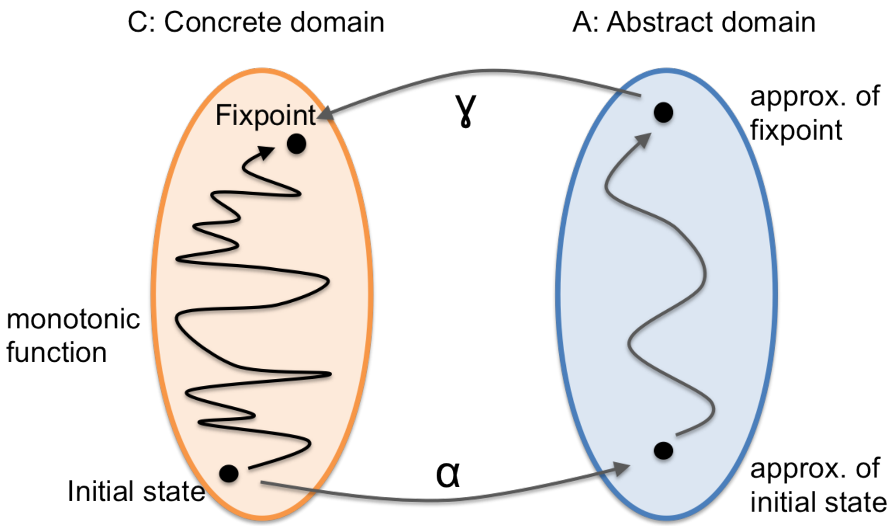
$$\begin{pmatrix} x : 10 \\ y : -2 \\ z : 3 \end{pmatrix} \xrightarrow{z=x+y} \begin{pmatrix} x : 10 \\ y : -2 \\ z : 8 \end{pmatrix}$$

Abstract interpreter:

$$\begin{pmatrix} x \in & [0, 10] \\ y \in & [-5, 5] \\ z \in & [0, 10] \end{pmatrix} \xrightarrow{z=x+y} \begin{pmatrix} x \in & [0, 10] \\ y \in & [-5, 5] \\ z \in & [-5, 15] \end{pmatrix}$$

Each abstract state represents a set of concrete states

Program Meaning is a Fixpoint. We Approximate It.



γ maps abstract states to concrete states

Proving through Fixpoints of Approximate Functions

Meaning of a program (e.g. a relation) is a least fixpoint of F.
Given specification s, the goal is to prove $\mathbf{lfp}(\mathbf{F}) \subseteq \mathbf{s}$

- if $F(s) \subseteq s$ then $lfp(F) \subseteq s$ and we are done
- $lfp(F) = \bigcup_{k \geq 0} F^k(\emptyset)$, but that is too hard to compute because it is infinite union unless, by some luck, $F^{n+1}(\emptyset) = F^n$ for some n

Instead, we search for an inductive strengthening of s: find s' such that:

- $F(s') \subseteq s'$ (s' is inductive). If so, theorem says $lfp(F) \subseteq s'$
- $s' \subseteq s$ (s' implies the desired specification). Then $lfp(F) \subseteq s' \subseteq s$

How to find s'? Iterating F is hard, so we try some simpler function $F_\#$

- suppose $F_\#$ is *approximation*: $F(r) \subseteq F_\#(r)$ for all r
- we can find s' such that: $F_\#(s') \subseteq s'$ (e.g. $s' = F_\#^{n+1}(\emptyset) = F_\#^n(\emptyset)$)

Then: $F(s') \subseteq F_\#(s') \subseteq s' \subseteq s$
Abstract interpretation: automatically construct $F_\#$ from F (and sometimes s)

Programs as control-flow graphs

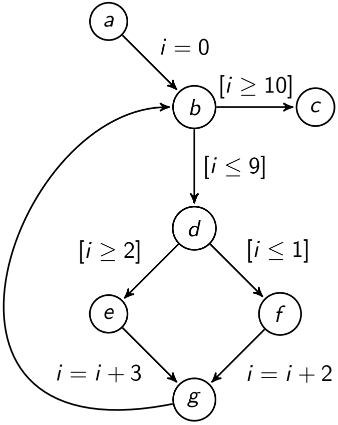
```
//a
i = 0;
     //b
while (i < 10) {
  //d
  if (i > 1)
    //e
    i = i + 3;
  else
    //f
    i = i + 2;
  //g
}
//c
```

One possible corresponding control-flow graph is:

Programs as control-flow graphs

```
//a
i = 0;
      //b
while (i < 10) {
  //d
  if (i > 1)
    //e
    i = i + 3;
  else
    //f
    i = i + 2;
  //g
}
//c
```

One possible corresponding control-flow graph is:



Sets of states at each program point

Suppose that

- program state is given by the value of the integer variable i
- initially, it is possible that i has any value

Compute the set of states at each vertex in the CFG.

```
//a
i = 0;
     //b
while (i < 10) {
  //d
  if (i > 1)
    //e
    i = i + 3;
  else
    //f
    i = i + 2;
  //g
}
//c
```

$$\begin{tikzcd}
a \arrow[dr, "i = 0"] & & & \\
& b \arrow[rr, "{[i \geq 10]}"] \arrow[d, "{[i \leq 9]}"] & & c \\
& d \arrow[dl, "{[i \geq 2]}"'] \arrow[dr, "{[i \leq 1]}"] & & \\
e \arrow[dr, "i = i + 3"'] & & f \arrow[dl, "i = i + 2"] & \\
& g \arrow[uuu, bend left=90] & &
\end{tikzcd}$$

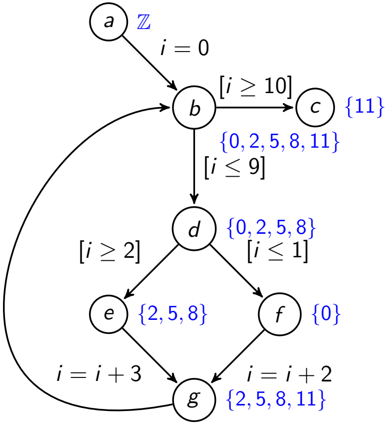
Sets of states at each program point

Suppose that

- program state is given by the value of the integer variable *i*
- initially, it is possible that i has any value

Compute the set of states at each vertex in the CFG.

```
//a
i = 0;
     //b
while (i < 10) {
  //d
  if (i > 1)
    //e
    i = i + 3;
  else
    //f
    i = i + 2;
  //g
}
//c
```



Sets of states at each program point

Running the Program
One way to describe the set of states for each program point: for each initial state, run the CFG with this state and insert the modified states at appropriate points.

Reachable States as A Set of Recursive Equations
If c is the label on the edge of the graph, let $\rho(c)$ denotes the relation between initial and final state that describes the meaning of statement. For example,

$$\begin{aligned}
&\rho(i = 0) = \{(i, i') \mid i' = 0\} \\
&\rho(i = i + 2) = \{(i, i') \mid i' = i + 2\} \\
&\rho(i = i + 3) = \{(i, i') \mid i' = i + 3\} \\
&\rho([i < 10]) = \{(i, i') \mid i' = i \wedge i < 10\}
\end{aligned}$$

Sets of states at each program point

We will write $T(S, c)$ (transfer function) for the image of set S under relation $\rho(c)$. For example,

$$T(\{10, 15, 20\}, i = i + 2) = \{12, 17, 22\}$$

General definition can be given using the notion of strongest postcondition

$$T(S, c) = sp(S, \rho(c))$$

If [p] is a condition (assume(p), coming from 'if' or 'while') then

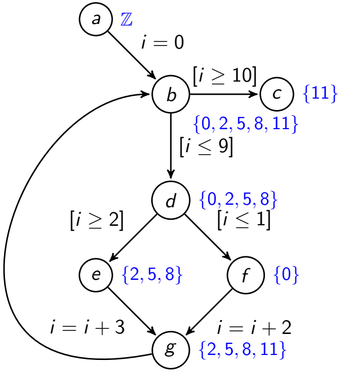
$$T(S, [p]) = \{x \in S \mid p\}$$

If an edge has no label, we denote it skip. So, $T(S, skip) = S$.

Reachable States as A Set of Recursive Equations

Now we can describe the meaning of our program using recursive equations:

$$
\begin{aligned}
S(a) &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
S(b) &= T(S(a), i = 0) \cup T(S(g), skip) \\
S(c) &= T(S(b), [\neg(i < 10)]) \\
S(d) &= T(S(b), [i < 10]) \\
S(e) &= T(S(d), [i > 1]) \\
S(f) &= T(S(d), [\neg(i > 1)]) \\
S(g) &= T(S(e), i = i + 3) \\
&\quad \cup T(S(f), i = i + 2)
\end{aligned}
$$



Our solution is the unique **least** solution of these equations. Can be computed by iterating starting from empty sets as initial solution.

The problem: These exact equations are as difficult to compute as running the program on all possible input states. Instead, we consider **approximate** descriptions of these sets of states.

A Large Analysis Domain: All Intervals of Integers

For every $L, U \in \mathbb{Z}$ interval:

$$\{x \mid L \leq x \wedge x \leq U\}$$

This domain has infinitely many elements, but is already an approximation of all possible sets of integers.

Smaller Domain: Finitely Many Intervals

We continue with the same example but instead of allowing to denote all possible sets, we will allow sets represented by expressions

$$[L, U]$$

which denote the set $\{x \mid L \leq x \wedge x \leq U\}$.
Example: $[0, 127]$ denotes integers between 0 and 127.

- L is the lower bound and U is the upper bound, with $L \leq U$.
- to ensure that we have only a few elements, we let

$$L, U \in \{\text{MININT}, -128, 1, 0, 1, 127, \text{MAXINT}\}$$

- $[\text{MININT}, \text{MAXINT}]$ denotes all possible integers, denote it $\top$
- instead of writing $[1, 0]$ and other empty sets, we will always write $\bot$

So, we only work with a finite number of sets $1 + \binom{7}{2} = 22$.
Denote the family of these sets by D (domain).

New Set of Recursive Equations

We want to write the same set of equations as before, but because we have only a finite number of sets, we must approximate. We approximate sets with possibly larger sets.

$$
\begin{aligned}
S^{\#}(a) &= \top \\
S^{\#}(b) &= T^{\#}(S^{\#}(a), i = 0) \\
&\sqcup\ T^{\#}(S^{\#}(g), skip) \\
S^{\#}(c) &= T^{\#}(S^{\#}(b), [\neg(i < 10)]) \\
S^{\#}(d) &= T^{\#}(S^{\#}(b), [i < 10]) \\
S^{\#}(e) &= T^{\#}(S^{\#}(d), [i > 1]) \\
S^{\#}(f) &= T^{\#}(S^{\#}(d), [\neg(i > 1)]) \\
S^{\#}(g) &= T^{\#}(S^{\#}(e), i = i + 3) \\
&\sqcup\ T^{\#}(S^{\#}(f), i = i + 2)
\end{aligned}
$$

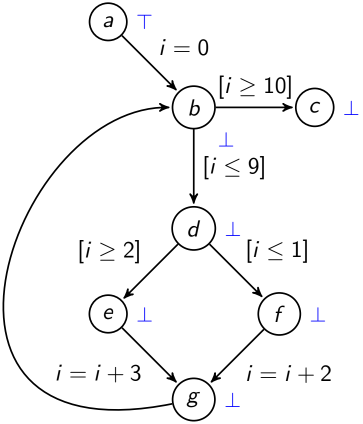
- $S_1 \sqcup S_2$ denotes the approximation of $S_1 \cup S_2$: it is the set that contains both S_1 and S_2, that belongs to D, and is otherwise as small as possible. Here $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$
- We use approximate functions $T^{\#}(S, c)$ that give a result in D.

Updating Sets

We solve the equations by starting in the initial state and repeatedly applying them.

- in the 'entry' point, we put $\top$, in all others we put $\bot$.

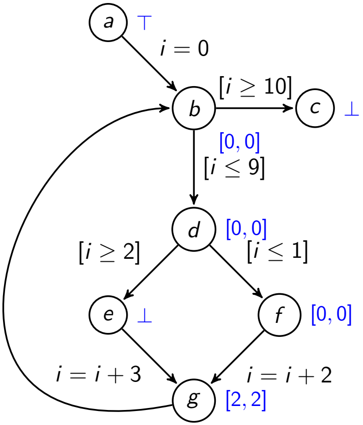
$$
\begin{aligned}
S^\#(a) &= \top \\
S^\#(b) &= T^\#(S^\#(a), i = 0) \\
&\sqcup\ T^\#(S^\#(g), \mathit{skip}) \\
S^\#(c) &= T^\#(S^\#(b), [\neg(i < 10)]) \\
S^\#(d) &= T^\#(S^\#(b), [i < 10]) \\
S^\#(e) &= T^\#(S^\#(d), [i > 1]) \\
S^\#(f) &= T^\#(S^\#(d), [\neg(i > 1)]) \\
S^\#(g) &= T^\#(S^\#(e), i = i + 3) \\
&\sqcup\ T^\#(S^\#(f), i = i + 2)
\end{aligned}
$$



Updating Sets

Sets after a few iterations:

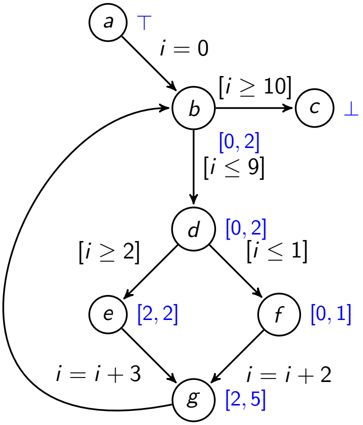
$$
\begin{aligned}
S^\#(a) &= \top \\
S^\#(b) &= T^\#(S^\#(a), i = 0) \\
&\sqcup\ T^\#(S^\#(g), \mathit{skip}) \\
S^\#(c) &= T^\#(S^\#(b), [\neg(i < 10)]) \\
S^\#(d) &= T^\#(S^\#(b), [i < 10]) \\
S^\#(e) &= T^\#(S^\#(d), [i > 1]) \\
S^\#(f) &= T^\#(S^\#(d), [\neg(i > 1)]) \\
S^\#(g) &= T^\#(S^\#(e), i = i + 3) \\
&\sqcup\ T^\#(S^\#(f), i = i + 2)
\end{aligned}
$$



Updating Sets

Sets after a few more iterations:

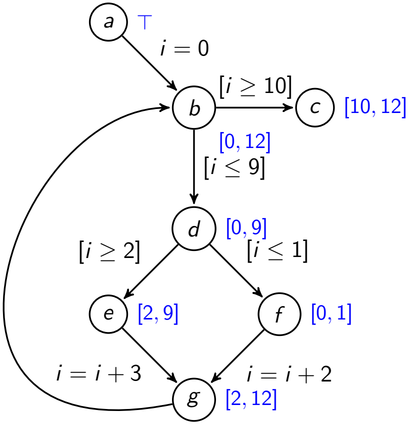
$$
\begin{aligned}
S^{\#}(a) &= \top \\
S^{\#}(b) &= T^{\#}(S^{\#}(a), i = 0) \\
&\quad \sqcup T^{\#}(S^{\#}(g), skip) \\
S^{\#}(c) &= T^{\#}(S^{\#}(b), [\neg(i < 10)]) \\
S^{\#}(d) &= T^{\#}(S^{\#}(b), [i < 10]) \\
S^{\#}(e) &= T^{\#}(S^{\#}(d), [i > 1]) \\
S^{\#}(f) &= T^{\#}(S^{\#}(d), [\neg(i > 1)]) \\
S^{\#}(g) &= T^{\#}(S^{\#}(e), i = i + 3) \\
&\quad \sqcup T^{\#}(S^{\#}(f), i = i + 2)
\end{aligned}
$$



Fixpoint Found

Final values of sets:

$$
\begin{aligned}
S^\#(a) &= \top \\
S^\#(b) &= T^\#(S^\#(a), i = 0) \\
&\sqcup\ T^\#(S^\#(g), \mathit{skip}) \\
S^\#(c) &= T^\#(S^\#(b), [\neg(i < 10)]) \\
S^\#(d) &= T^\#(S^\#(b), [i < 10]) \\
S^\#(e) &= T^\#(S^\#(d), [i > 1]) \\
S^\#(f) &= T^\#(S^\#(d), [\neg(i > 1)]) \\
S^\#(g) &= T^\#(S^\#(e), i = i + 3) \\
&\sqcup\ T^\#(S^\#(f), i = i + 2)
\end{aligned}
$$

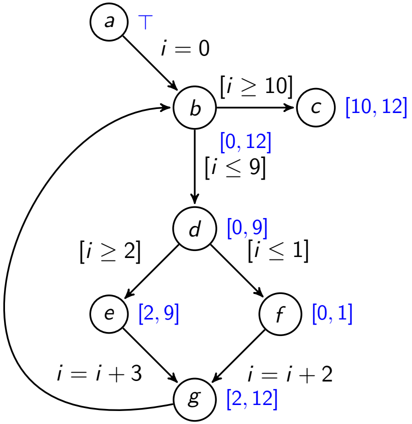


If we map intervals to sets, this is also solution of the original constraints.

Automatically Constructed Hoare Logic Proof

Final values of sets:

```
//a: true
i = 0;
     //b: $0 \le i \le 12$
while (i < 10) {
  //d: $0 \le i \le 9$
  if (i > 1)
    //e: $2 \le i \le 9$
    i = i + 3;
  else
    //f: $0 \le i \le 1$
    i = i + 2;
  //g: $2 \le i \le 12$
}
//c: $10 \le i \le 12$
```



This method constructed a sufficiently annotated program and ensured that all Hoare triples that were constructed hold