Converting Imperative Programs to Formulas

Viktor Kunčak

Verification-Condition Generation

Given:

- program and
- specification

how to express program correctness as a **verification condition**, a formula implying that program satisfies the specification?

Verification-Condition Generation

Given:

- program and
- specification

how to express program correctness as a **verification condition**,
a formula implying that program satisfies the specification?

How to implement verification condition generation as an algorithm?

Verification-Condition Generation

Given:

- program and
- specification

how to express program correctness as a **verification condition**,
a formula implying that program satisfies the specification?

How to implement verification condition generation as an algorithm?

Example program that verifies in Stainless:

```
import stainless.lang._
import stainless.lang.StaticChecks._
case class FirstExample(var x: BigInt, var y: BigInt) {
  def increase : Unit = {
    x = x + 2 // change the value of x
    y = x + 10 // refer to changed value
  }.ensuring(_ => old(this).x > 0 ==> (x > 2 && y > 12)) // relates old and new values
}
```

Programs are Formulas. Specifications are Formulas

A program fragment can be represented by a formula relating initial and final state.
Consider a program with variables x, y

$$\begin{array}{rc} \text{program:} & x = x + 2; y = x + 10 \\ \text{relation:} & \{((x,y),(x',y')) \,|\, x' = x + 2 \land y' = x + 12\} \\ \text{formula:} & x' = x + 2 \land y' = x + 12 \end{array}$$

Specification was: $old(this).x > 0 \rightarrow (x > 2 \land y > 12)$
We express that program satisfies the postcondition using **relation subset**:

$$\begin{array}{rl} & \{((x,y),(x',y')) \,|\, x' = x + 2 \land y' = x + 12\} \\ \subseteq & \{((x,y),(x',y')) \,|\, x > 0 \rightarrow (x' > 2 \land y' > 12)\} \end{array}$$

which reduces to the **validity of the following implication**:

$$\begin{array}{rl} & x' = x + 2 \land y' = x + 12 \\ \rightarrow & (x > 0 \rightarrow (x' > 2 \land y' > 12)) \end{array}$$

Simple Imperative Programs

F - formulas, t - terms (with only pure mathematical operations)
Fixed number of mutable variables $V = \{x_1, \ldots, x_n\}$
Imperative statements:

- $\mathbf{x = t}$: change $x \in V$ to have value given by t; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if(F) c_1 \ else \ c_2}$: if F holds, execute c_1 else execute c_2
- $\mathbf{c_1 ; c_2}$: first execute c_1, then execute c_2

Statements for introducing and restricting non-determinism:

- $\mathbf{havoc(x)}$: non-deterministically change $x \in V$ to have an arbitrary value; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if(*) \ c_1 \ else \ c_2}$: arbitrarily choose to run c_1 or c_2
- $\mathbf{assume(F)}$: block all executions where F does not hold

Given such loop-free program c with conditionals, compute a polynomial-sized formula $R(c)$ of form: $\exists \bar{z}. F(\bar{x}, \bar{z}, \bar{x}')$ describing relation between initial values of variables $x_1, \ldots, x_n$ and final values of variables $x_1', \ldots, x_n'$

Construction Formula that Describe Relations

c - imperative command

$R(c)$ - formula describing relation between initial and final states of execution of c

If $\rho(c)$ describes the relation, then $R(c)$ is formula such that

$$\rho(c) = \{(\bar{x}, \bar{x}') \mid R(c)\}$$

$R(c)$ is a formula between unprimed variables $\bar{x}$ and primed variables $\bar{x}'$

Formula for Assignment

$$x = t$$

Formula for Assignment

$$x = t$$

$R(x = t)$:

$$x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Note that the formula must explicitly state which variables remain the same (here: all except x). Otherwise, those variables would not be constrained by the relation, so they could take arbitrary value in the state after the command.

Examples:
$R(x = x + 2) \; = \; x' = x + 2 \;\wedge\; y' = y$
$R(y = x + 10) \; = \; x' = x \;\wedge\; y' = x + 10$

Formula for if-else

$$\mathit{if}\,(b)\ c_1\ \mathit{else}\ c_2$$

Formula for if-else

$$\mathit{if}\,(b)\ c_1\ \mathit{else}\ c_2$$

$R(\mathit{if}\,(b)\ c_1\ \mathit{else}\ c_2)$:

$$(b \wedge R(c_1)) \vee (\neg b \wedge R(c_2))$$

Command semicolon

$$c_1; c_2$$

Command semicolon

$$c_1; c_2$$

Corresponds to relation composition:

$$r_1 \circ r_2 = \{(\bar{x}, \bar{x}') \mid \exists \bar{x}''. (\bar{x}, \bar{x}'') \in r_1 \land (\bar{x}'', \bar{x}') \in r_2\}$$

Command semicolon

$$c_1; c_2$$

Corresponds to relation composition:

$$r_1 \circ r_2 = \{(\bar{x}, \bar{x}') \mid \exists \bar{x}''.(\bar{x}, \bar{x}'') \in r_1 \wedge (\bar{x}'', \bar{x}') \in r_2\}$$

What are $R(c_1)$ and $R(c_2)$ and in terms of which variables they are expressed?

Command semicolon

$$c_1; c_2$$

Corresponds to relation composition:

$$r_1 \circ r_2 = \{(\bar{x}, \bar{x}') \mid \exists \bar{x}''. (\bar{x}, \bar{x}'') \in r_1 \wedge (\bar{x}'', \bar{x}') \in r_2\}$$

What are $R(c_1)$ and $R(c_2)$ and in terms of which variables they are expressed? Each in terms of $\bar{x}$ and $\bar{x}'$.
Let $r_1 = \{(\bar{x}, \bar{x}') \mid R(c_1)\}$, $r_2 = \{(\bar{x}, \bar{x}') \mid R(c_2)\}$
Thus, $(\bar{x}, \bar{x}'') \in r_1 \longleftrightarrow (\bar{x}, \bar{x}'') \in \{(\bar{x}, \bar{x}') \mid R(c_1)\} \longleftrightarrow R(c_1)[\bar{x}' := \bar{x}'']$
Similarly, $(\bar{x}'', \bar{x}') \in r_2 \longleftrightarrow R(c_2)[\bar{x}' := \bar{x}'']$

Command semicolon

$$c_1; c_2$$

Corresponds to relation composition:

$$r_1 \circ r_2 = \{(\bar{x}, \bar{x}') \mid \exists \bar{x}''. (\bar{x}, \bar{x}'') \in r_1 \land (\bar{x}'', \bar{x}') \in r_2\}$$

What are $R(c_1)$ and $R(c_2)$ and in terms of which variables they are expressed? Each in terms of $\bar{x}$ and $\bar{x}'$.
Let $r_1 = \{(\bar{x}, \bar{x}') \mid R(c_1)\}$, $r_2 = \{(\bar{x}, \bar{x}') \mid R(c_2)\}$
Thus, $(\bar{x}, \bar{x}'') \in r_1 \longleftrightarrow (\bar{x}, \bar{x}'') \in \{(\bar{x}, \bar{x}') \mid R(c_1)\} \longleftrightarrow R(c_1)[\bar{x}' := \bar{x}'']$
Similarly, $(\bar{x}'', \bar{x}') \in r_2 \longleftrightarrow R(c_2)[\bar{x}' := \bar{x}'']$
$R(c_1; c_2) \longleftrightarrow (\bar{x}, \bar{x}') \in r_1 \circ r_2 \longleftrightarrow$

$$\exists \bar{x}''.\ \ R(c_1)[\bar{x}' := \bar{x}''] \land R(c_2)[\bar{x} := \bar{x}'']$$

where $\bar{x}''$ are freshly picked names of intermediate states.

- a useful convention: $\bar{x}''$ refer to position in program source code, $\bar{x}^i$

Computing relation for the example from before

$$
\begin{aligned}
& R(x=x+2; y=x+10) \ = \exists \bar{x}''.\ \ R(c_1)[\bar{x}' := \bar{x}''] \wedge R(c_2)[\bar{x} := \bar{x}''] \\
& = \exists x'', y''. \quad (x'=x+2 \ \wedge \ y'=y)[x' := x'', y' := y''] \ \wedge \\
& \qquad\qquad\quad (x'=x \ \wedge \ y'=x+10)[x := x'', y := y''] \\
& = \exists x'', y''. \quad (x''=x+2 \ \wedge \ y''=y) \ \wedge \\
& \qquad\qquad\quad (x'=x'' \ \wedge \ y'=x''+10) \qquad\qquad\qquad (*) \\
& \longleftrightarrow \quad (x'=x+2 \ \wedge \ y'=x+2+10) \\
& \longleftrightarrow \quad (x'=x+2 \ \wedge \ y'=x+12)
\end{aligned}
$$

Where at step $(*)$ we used (twice) the “one-point rule” of logic with equality:

$$(\exists u.(u=t \wedge F)) \ \longleftrightarrow \ F[u := t]$$

if $u \notin FV(t)$.

havoc

Definition of HAVOC

1. wide and general destruction: devastation
2. great confusion and disorder

Example of use:

$$\mathrm{y} = 12;\ \mathrm{havoc(x)};\ \mathrm{assume(x + x = y)}$$

ends up dividing x by two!

Translation, $R(havoc(x))$:

havoc

Definition of HAVOC

1. wide and general destruction: devastation
2. great confusion and disorder

Example of use:

y = 12; havoc(x); assume(x + x = y)

ends up dividing x by two!

Translation, $R(havoc(x))$:

$$\bigwedge_{v \in V \setminus \{x\}} v' = v$$

This again illustrates “politically correct” approach to describing the destruction of values of variables: just do not mention them.

Non-deterministic choice

$$\mathit{if}\,(*)\ c_1\ \mathit{else}\ c_2$$

Non-deterministic choice

$$if\,(*)\ c_1\ \ else\ \ c_2$$

$R(if\,(*)\ c_1\ \ else\ \ c_2)$:

$$R(c_1) \vee R(c_2)$$

- translation is simply a disjunction – this is why construct is interesting
- corresponds to branching in control-flow graphs

assume

$$assume(F)$$

assume

$R(assume(F))$:

$$assume(F)$$

$$F \wedge \bigwedge_{v \in V} v' = v$$

assume

$$assume(F)$$

$R(assume(F))$:

$$F \land \bigwedge_{v \in V} v' = v$$

- This command does not change any state.

assume

$$assume(F)$$

$R(assume(F))$:

$$F \wedge \bigwedge_{v \in V} v' = v$$

- This command does not change any state.
- If F does not hold, it stops with "instantaneous success".

Example of Translation

$$
\begin{array}{l}
{}^{0}\\
(\mathit{if}\ (b)\ x = x + 1\ \mathit{else}\ y = x + 2);\\
{}^{1}\\
x = x + 5;\\
{}^{2}\\
(\mathit{if}\ (*)\ y = y + 1\ \mathit{else}\ x = y)\\
{}^{3}
\end{array}
$$

becomes

$$
\begin{aligned}
\exists x_1, y_1, x_2, y_2.\ & ((b \wedge \mathbf{x_1} = \mathbf{x} + \mathbf{1} \wedge y_1 = y) \vee (\neg b \wedge x_1 = x \wedge \mathbf{y_1} = \mathbf{x} + \mathbf{2}))\\
& \wedge\ (\mathbf{x_2} = \mathbf{x_1} + \mathbf{5} \wedge y_2 = y_1)\\
& \wedge\ ((x' = x_2 \wedge \mathbf{y}' = \mathbf{y_2} + \mathbf{1}) \vee (\mathbf{x}' = \mathbf{y_2} \wedge y' = y_2))
\end{aligned}
$$

Think of execution trace $(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)$ where

- (x_0, y_0) is denoted by (x, y)
- (x_3, y_3) is denoted by (x', y')

Justifying the name for assume(F)

Compute and simplify as much as possible each of the following expressions:

1. $R(assume(F); c)$

Justifying the name for assume(F)

Compute and simplify as much as possible each of the following expressions:

1. $R(\mathit{assume}(F); c) \quad = \quad F \wedge R(c)$
2. $R(c; \mathit{assume}(F))$

Justifying the name for assume(F)

Compute and simplify as much as possible each of the following expressions:

1. $R(\mathit{assume}(F); c) \quad = \quad F \wedge R(c)$
2. $R(c; \mathit{assume}(F)) \quad = \quad R(c) \wedge F[\bar{x} := \bar{x}']$
   where $F[\bar{x} := \bar{x}']$ denotes F with all variables replaced with primed versions

Expressing **if** through non-deterministic choice and assume

Expressing **if** through non-deterministic choice and assume

```
if (b) c1 else c2

    |||

if (*) {
  assume(b);
  c1
} else {
  assume(!b);
  c2
}
```

Indeed, apply translation to both sides and observe that generated formulas are equivalent.

Expressing assignment through havoc and assume

Expressing assignment through havoc and assume

x = e

$|||$

havoc(x);
assume(x == e)

Under what conditions this holds?

Expressing assignment through havoc and assume

x = e

$|||$

havoc(x);
assume(x == e)

Under what conditions this holds?
$x \notin FV(e)$

Illustration of the problem: $havoc(x);\ assume(x == x + 1)$

Expressing assignment through havoc and assume

x = e

$|||$

havoc(x);
assume(x == e)

Under what conditions this holds?
$x \notin FV(e)$

Illustration of the problem: $havoc(x);\ assume(x == x+1)$

Luckily, we can rewrite it into $x_{fresh} = x+1; x = x_{fresh}$

Loop-Free Programs as Relations: Summary

command c	$R(c)$	$\rho(c)$
$(x = t)$	$x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$	
$c_1 ; c_2$	$\exists \bar{z}.\ \ R(c_1)[\bar{x}' := \bar{z}] \wedge R(c_2)[\bar{x} := \bar{z}]$	$\rho(c_1) \circ \rho(c_2)$
if$(*)$ c_1 **else** c_2	$R(c_1) \vee R(c_2)$	$\rho(c_1) \cup \rho(c_2)$
assume$(\mathbf{F})$	$F \wedge \bigwedge_{v \in V} v' = v$	$\Delta_{S(F)}$

$\rho(x_i = t) = \{((x_1, \ldots, x_i, \ldots, x_n), (x_1, \ldots, x_i', \ldots, x_n) \mid x_i' = t\}$

$S(F) = \{\bar{x} \mid F\}, \quad \Delta_A = \{(\bar{x}, \bar{x}) \mid \bar{x} \in A\}$ (diagonal relation on A)

Δ (without subscript) is identity on entire set of states (no-op)

We always have: $\rho(c) = \{(\bar{x}, \bar{x}') \mid R(c)\}$

Shorthands:

if$(*)$ c_1 **else** c_2	$c_1 \,\square\, c_2$
assume(F)	$[F]$

Examples:

$$\textbf{if}\,(F)\ c_1\ \textbf{else}\ c_2 \ \equiv\ [F]; c_1 \ \square\ [\neg F]; c_2$$
$$\textbf{if}\,(F)\ c \ \equiv\ [F]; c \ \square\ [\neg F]$$

Program Paths

Loop-Free Programs

c - a loop-free program whose assignments, havocs, and assumes are $c_1, \ldots, c_n$

The relation $\rho(c)$ is of the form $E(\rho(c_1), \ldots, \rho(c_n))$; it composes meanings of $c_1, \ldots, c_n$ using union ($\cup$) and composition ($\circ$)

```
(if (x > 0)
   x = x − 1
else
   x = 0
);
(if (y > 0)
   y = y − 1
else
   y = x + 1
)
```

```
([x > 0]; x = x − 1
 []
 ([¬(x>0)]; x = 0)
);
([y > 0]; y = y − 1
 []
 [¬(y>0)]; y = x+1
)
```

$$\Big(\Delta_{S(x>0)} \circ \rho(x = x - 1) \;\cup\; \Delta_{S(\neg(x>0))} \circ \rho(x = 0)\Big) \circ \Big(\Delta_{S(y>0)} \circ \rho(y = y - 1) \;\cup\; \Delta_{S(\neg(y>0))} \circ \rho(y = x + 1)\Big)$$

Note: $\circ$ binds stronger than $\cup$, so $r \circ s \cup t = (r \circ s) \cup t$

Normal Form for Loop-Free Programs

Composition distributes through union:

$$(r_1 \cup r_2) \circ (s_1 \cup s_2) = r_1 \circ s_1 \ \cup \ r_1 \circ s_2 \ \cup \ r_2 \circ s_1 \ \cup \ r_2 \circ s_2$$

Example corresponding to two if-else statements one after another:

$$\begin{array}{l} \big(\Delta_1 \circ r_1 \\ \quad \cup \\ \ \Delta_2 \circ r_2 \\ \big) \circ \\ \big(\Delta_3 \circ r_3 \\ \quad \cup \\ \ \Delta_4 \circ r_4 \\ \big) \end{array} \qquad \equiv \qquad \begin{array}{l} \Delta_1 \circ r_1 \circ \Delta_3 \circ r_3 \ \cup \\ \Delta_1 \circ r_1 \circ \Delta_4 \circ r_4 \ \cup \\ \Delta_2 \circ r_2 \circ \Delta_3 \circ r_3 \ \cup \\ \Delta_2 \circ r_2 \circ \Delta_4 \circ r_4 \end{array}$$

Sequential composition of basic statements is called basic path.
Loop-free code describes finitely many (exponentially many) paths.