Lecture 3
From (Integer) Programs to Formulas

Viktor Kuncak

Verification Condition Generation Example

We examine algorithms for going from programs to their verification conditions.

Program and postcondition:

```
def f(x : Int) : Int = {
  if (x > 0)
    2*x + 1
  else 42
} ensuring (res => res > 0)
```

Verification condition saying "program satisfies postcondition":

$$\left[((x > 0) \wedge res = 2x + 1) \vee (\neg(x > 0) \wedge res = 42)\right] \;\rightarrow\; res > 0$$

We check validity: all variables are universally quantified

Verification Condition Generation (VCG) For Functions

def f($\bar{x}$: Int^n) : Int = {
  b($\bar{x}$)
} **ensuring** (res **=>** Post($\bar{x}$, res))

- Function f with arguments $\bar{x}$ and body $b(\bar{x})$, built from:
  - Presburger Arithmetic (PA) expressions, as well as x/K, $x\%K$
  - **if** statement, and local value definitions (**val** in Scala)
- Postcondition $Post(\bar{x}, res)$ written in quantifier-free PA

Claim: there is **polynomial-time** algorithm to construct formula $V(\bar{x})$ such that

- the execution of f on input $\bar{x}$ meets the Post iff $V(\bar{x})$
  Hence, it always meets postcondition iff $\forall \bar{x}. V(\bar{x})$
- $V(\bar{x})$ is quantifier-free or has only top-level $\forall$ quantifiers

Idea: perhaps $V(\bar{x})$ could be $Post(\bar{x}, b(\bar{x}))$? Yes, if it was in PA

PA with x/K, $x\%K$, **if**, **val**

Context-Free grammar (syntax) of extended PA formulas

F : Boolean, t : Int

$$
\begin{array}{rcl}
F & ::= & b \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \mid t_1 < t_2 \mid t_1 = t_2 \\
 & \mid & \{\mathbf{val\ x = t;\ F}\} \mid \{\mathbf{val\ b = F_1;\ F}\} \\
t & ::= & x \mid K \mid t_1 + t_2 \mid K \cdot t \\
 & \mid & \mathbf{t/K} \mid \mathbf{t\ \%\ K} \mid \mathbf{if\,(F)\,t_1\ else\ t_2} \mid \{\mathbf{val\ x = t_1;\ t_2}\}
\end{array}
$$

We can translate x/K, $x\%K$, **if**, **val** into other constructs

- in polynomial time
- without changing the meaning of a formula
- without adding alternations of quantifiers

Notation: Free Variables

$FV(t), FV(F)$ denotes free variables in term t or formula F
Normally we just collect all variables:

$$FV(x + y < z) = \{x, y, z\}$$

We do not count quantified occurrences of variables:

$$FV(\exists x.\ x + y < z) = \{y, z\}$$

Even if it occurs quantified somewhere, if there is a path in formula tree that reaches it without being blocked by quantifiers, then the variables is free:

$$FV((\exists x.\exists y.x < y + u) \wedge (\exists y.x + y < z + 100)) = \{u, x, z\}$$

General rules are of two kinds: operations and binders

$$FV(F_1 \odot F_2) = FV(F_1) \cup FV(F_2)$$
$$FV(Qx.F) = FV(F) \setminus \{x\}$$

Notation: Substitutions

One possible convention: write $F(x)$ and later $F(t)$. Then F is not a formula but function from terms to formulas
(Or we do not even know what F is.)
Alternative notation: write F, and instead of $F(t)$ write $F[x := t]$

- closer to a typical implementation

Definition:

$$
\begin{array}{l}
(F_1 \odot F_2)[x := t] \leadsto F_1[x := t] \odot F_2[x := t] \\
(Qy.F)[x := t] \leadsto Qy.(F[x := t])
\end{array}
$$

Capture:
The following formula is true in integers for all x: $\exists y.x < y$
If we naively substitute x with $y + 1$ we obtain: $\exists y.\ y + 1 < y$
Problem: t has y free. A solution: rename y to fresh y_1

$$(Qy.F)[x := t] \leadsto (Qy_1.F[y := y_1])[x := t] \leadsto Qy_1.(F[y := y_1][x := t])$$

How to Translate Value Definitions

Construct: $\{val\ x = t;\ F\}$ where we require $x \notin FV(t)$
(otherwise just rename it to $\{val\ x_1 = t;\ F[x := x_1]\}$)

Example

$$\{val\ x = y + 1;\ x < 2x + 5\}$$

Becomes one of these:

How to Translate Value Definitions

Construct: $\{val\ x = t;\ F\}$ where we require $x \notin FV(t)$
(otherwise just rename it to $\{val\ x_1 = t;\ F[x := x_1]\}$)

Example

$$\{val\ x = y + 1;\ x < 2x + 5\}$$

Becomes one of these:

$(y+1) < 2(y+1) + 5$	substitution
$\exists x.\ x = y + 1 \land x < 2x + 5$	one-point rule
$\forall x.\ x = y + 1 \rightarrow x < 2x + 5$	dual one-point rule

Rule to Translate Value Definitions

In general, for $x \notin FV(t)$

$$\{val\ x = t;\ F\}$$

Becomes one of these:

Rule to Translate Value Definitions

In general, for $x \notin FV(t)$

$$\{val\ x = t;\ F\}$$

Becomes one of these:

$F[x := t]$	substitution
$\exists x.\ x = t \wedge F$	one-point rule
$\forall x.\ x = t \rightarrow F$	dual one-point rule

Substitution can square formula size

- Do it several times $\rightsquigarrow$ exponential increase

The other rules add quantified variables

- but we can choose which way they are quantified, to avoid adding quantifier alternations

Flattening: Remove All Nested Terms

Similar to compilation
Example:

$$x + 3y < z$$

flattening $3y$ and denoting it by y_1 we get

$$\{val\ y_1 = 3y;\ x + y_1 < z\}$$

and then flattening $x + y_1$ denoting it by y_2 we get

$$\{val\ y_1 = 3y;\ \{val\ y_2 = x + y_1;\ y_2 < z\}\}$$

which we may write as

```
{ val y1=3y
  val y2=x+y1
  y2 < z
}
```

Flattening Rule

Suppose F contains $t_1 \odot t_2$ somewhere and we wish to pull it out. For some fresh y_1 then F becomes

Flattening Rule

Suppose F contains $t_1 \odot t_2$ somewhere and we wish to pull it out. For some fresh y_1 then F becomes

$$\{ val\ y_1 = t_1 \odot t_2;\ \ F[t_1 \odot t_2 := y_1]\ \}$$

We can now handle val for formulas. What about terms?

Lifting val-s outside until they reach formulas

$$\{val\ x = a + 1;\ 2x\} + 5 < y$$

becomes

We can now handle val for formulas. What about terms?

Lifting val-s outside until they reach formulas

$$\{val\ x = a + 1;\ 2x\} + 5 < y$$

becomes

$$\{val\ x = a + 1;\ 2x + 5 < y\}$$

val given by val rule

$$\{val\ x = \{val\ y = a + 1;\ y + y\};\ x < 2x\}$$

becomes

val given by val rule

$$\{val\ x = \{val\ y = a + 1;\ y + y\};\ x < 2x\}$$

becomes

$$\{val\ y = a + 1;\ \{val\ x = y + y;\ x < 2x\}\}$$

which we pretty-print as

$$\{val\ y = a + 1;\ \ val\ x = y + y;\ \ x < 2x\}$$

Flat form:

- each operation $\odot$ is inside a $\{val\ x = y_1 \odot y_2;\ F\}$
- atomic formulas only use variables
- val applies to formulas only (not terms)

Translating **if**

F : Boolean, t : Int

$$
\begin{array}{rcl}
F & ::= & b \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \mid t_1 < t_2 \mid t_1 = t_2 \\
 & \mid & \{\mathbf{val\ x = t;\ F}\} \mid \{\mathbf{val\ b = F_1;\ F}\} \\
t & ::= & x \mid K \mid t_1 + t_2 \mid K \cdot t \\
 & \mid & \mathbf{t/K} \mid \mathbf{t\ \%\ K} \mid \mathbf{if\,(F)\,t_1\ else\ t_2} \mid \{\mathbf{val\ x = t_1;\ t_2}\}
\end{array}
$$

Suppose terms are in flat form. We only need to handle:

$$\{val\ x = (if\,(b_1)\ t_1\ else\ t_2);\ F\}$$

Note that the logical equality

$$x = (if\,(b_1)\ t_1\ else\ t_2) \qquad (*)$$

is equivalent to

$$(b_1 \land x = t_1) \lor (\neg b_1 \land x = t_2)$$

as well as to:

$$((b_1 \to x = t_1) \land (\neg b_1 \to x = t_2))$$

Translating **if**

From two one-point rule translations of val, we can thus transform

$$\{val\ x = (if(b_1)\ t_1\ else\ t_2);\ F\}$$

into any of these:

$$\begin{array}{l}
\exists x.\ \big[((b_1 \land x = t_1) \lor (\neg b_1 \land x = t_2)) \land F\big] \\
\exists x.\ \big[((b_1 \rightarrow x = t_1) \land (\neg b_1 \rightarrow x = t_2)) \land F\big] \\
\forall x.\ \big[((b_1 \land x = t_1) \lor (\neg b_1 \land x = t_2)) \rightarrow F\big] \\
\forall x.\ \big[((b_1 \rightarrow x = t_1) \land (\neg b_1 \rightarrow x = t_2)) \rightarrow F\big]
\end{array}$$

This translates if-else without duplicating sub-formulas
(thanks to boolean variable b_1).

Integer Division by a Constant

Consider

$$\{val\ q = p/K;\ F\}$$

The corresponding equality $q = p/K$ is equivalent to

$$Kq \leq p \wedge p < K(q+1)$$

Which gives corresponding translations:

$$\begin{array}{l} \exists x.\ \left[Kq \leq p \wedge p < K(q+1) \wedge F\right] \\ \forall x.\ \left[(Kq \leq p \wedge p < K(q+1)) \to F\right] \end{array}$$

Remainder Modulo a Constant

$$\{val\ r = p\%K;\ F\}$$

Remainder Modulo a Constant

$$\{val\ r = p\%K;\ F\}$$

One way:

$$\{val\ r = p - K(p/K);\ F\}$$

Quantifier-Free Polynomial-Sized VC

def f($\bar{x}$: Int^n) : Int = {
  b($\bar{x}$)
} **ensuring** (res **=>** Post($\bar{x}$, res))

VC in quantifier-free PA extended with val, if, /, % :

$$res = b(\bar{x}) \rightarrow Post(res, \bar{x})$$

Quantifier-Free Polynomial-Sized VC

def f($\bar{x}$: Int^n) : Int = {
  b($\bar{x}$)
} **ensuring** (res => Post($\bar{x}$, res))

VC in quantifier-free PA extended with val, if, /, % :

$$res = b(\bar{x}) \rightarrow Post(res, \bar{x})$$

Eliminate extensions, choosing always existential quantifiers for new variables $\bar{z}$. Moreover, such existentials can be pulled to top-level, because we only introduced $\vee, \wedge$ and never $\neg$ for sub-formulas. We obtain:

$$(\exists \bar{z}.F(res, \bar{x}, \bar{z})) \rightarrow Post(res, \bar{x})$$

which is equivalent to

$$\forall \bar{z}.[F(res, \bar{x}, \bar{z}) \rightarrow Post(res, \bar{x})]$$

So, all variables are universally quantified.

Explaining $(\exists F) \to G$

Indeed, from first-order logic we have these equivalent formulas:

$$
\begin{array}{l}
(\exists \bar{z}.F(res, \bar{x}, \bar{z})) \to Post(res, \bar{x}) \\
\neg(\exists \bar{z}.F(res, \bar{x}, \bar{z})) \lor Post(res, \bar{x}) \\
(\forall \bar{z}.\neg F(res, \bar{x}, \bar{z})) \lor Post(res, \bar{x}) \\
\forall \bar{z}.[\neg F(res, \bar{x}, \bar{z}) \lor Post(res, \bar{x})] \\
\forall \bar{z}.[F(res, \bar{x}, \bar{z}) \to Post(res, \bar{x})]
\end{array}
$$

Checking validity is same as showing that

$$F(res, \bar{x}, \bar{z}) \to Post(res, \bar{x})$$

is true for all values of variables, or that

$$F(res, \bar{x}, \bar{z}) \land \neg Post(res, \bar{x})$$

has no satisfying assignments.

VC Generation for Imperative Non-Deterministic Programs

Program can be represented by a formula relating initial and final state.

program:	$x = x + 2; y = x + 10$
relation:	$\{(x, y, z, x', y', z') \mid x' = x + 2 \land y' = x + 12 \land z' = z\}$
formula:	$x' = x + 2 \land y' = x + 12 \land z' = z$

Specification: $z = old(z) \land (old(x) > 0 \to (x > 0 \land y > 0))$
Adhering to specification is relation subset:

$$\begin{array}{ll} & \{(x, y, z, x', y', z') \mid x' = x + 2 \land y' = x + 12 \land z' = z\} \\ \subseteq & \{(x, y, z, x', y', z') \mid z' = z \land (x > 0 \to (x' > 0 \land y' > 0))\} \end{array}$$

or validity of the following implication:

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

Adding State and Non-Determinism

Imperative Presburger Arithmetic Programs

F - formulas, t - terms - as in functional programs so far
Fixed number of mutable integer 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:

- **havoc**$(\mathbf{x})$: non-deterministically change $x \in V$ to have an arbitrary value; leave vars in $V \setminus \{x\}$ unchanged
- **if**$(*)$ $\mathbf{c_1}$ **else** $\mathbf{c_2}$: arbitrarily choose to run c_1 or c_2
- **assume**$(\mathbf{F})$: 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

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

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

Formula for Assignment

$$x = t$$

Formula for Assignment

$$x = t$$

$R(x = t)$:

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

Formula for if-else

After flattening,

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

Formula for if-else

After flattening,

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

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

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

Command semicolon

$$c_1 \,;\, c_2$$

Command semicolon

$$c_1 ; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \wedge (b, c) \in r_2\}$$

Command semicolon

$$c_1\,; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \land (b, c) \in r_2\}$$

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

Command semicolon

$$c_1 ; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \wedge (b, c) \in r_2\}$$

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

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

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

havoc

Definition of HAVOC

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

Example of use:

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

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)

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

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

Non-deterministic choice

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

Non-deterministic choice

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

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

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

assume

$$assume(F)$$

assume

$$assume(F)$$

$R(assume(F))$:

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

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

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

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')

Imperative Presburger Arithmetic Programs

F - formulas, t - terms - as in functional programs so far
Fixed number of mutable integer variables $V = \{x_1, \ldots, x_n\}$
Imperative statements:

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

Statements for introducing and restricting non-determinism:

- $\mathbf{havoc}(\mathbf{x})$: non-deterministically change $x \in V$ to have an arbitrary value; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if}(*)$ $\mathbf{c_1}$ **else** $\mathbf{c_2}$: arbitrarily choose to run c_1 or c_2
- $\mathbf{assume}(\mathbf{F})$: 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 $x_1', \ldots, x_n'$

Justifying the name for assume(F)

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

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

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
}
```

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 $x_{fresh} = x + 1; x = x_{fresh}$

Synthesis: From Specification to Code

From Quantifier Elimination to Synthesis

Quantifier Elimination

If $\bar{y}$ is a tuple of variables not containing x, then

$$\exists x.(x = t(\bar{y}) \land F(x, \bar{y})) \iff F(t(\bar{y}), \bar{y})$$

Synthesis

$$\mathit{choose}\ x.(x = t(\bar{y}) \land F(x, \bar{y}))$$

gives:

- precondition $F(t(\bar{y}), \bar{y})$, as before, but also
- program that realizes x, in this case, $t(\bar{y})$