Lecture 4
Refinement, Equivalence, and Synthesis

Viktor Kuncak

Local Mutable Variables

Local Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\mathbf{var}\ y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope.
Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $\quad z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) =$
$y' = x + 3 \wedge z' = 2x + 3 + z \wedge x' = x$

$R(\{\mathbf{var}\ y; y = x + 3; z = x + y + z\}) = \ z' = 2x + 3 + z \wedge x' = x$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V
For example,

$$R_V(x = t) \; = \; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define
$R_V(\{var\ y; P\}) \; = \; \exists y, y'. R_{V \cup \{y\}}(P)$

Exercise: express havoc(x) using var.

$$R_V(havoc(x)) \iff R_V(\{var\ y;\ x = y\})$$

Exercise: give transformation that lifts all variables to be global

Expressing Specifications as Commands

Shorthand: Havoc Multiple Variables at Once

Variables $V = \{x_1, \ldots, x_n\}$
Translation of $R(havoc(y_1, \ldots, y_m))$:

$$\bigwedge_{v \in V \setminus \{y_1, \ldots, y_m\}} v' = v$$

Exercise: the resulting formula is the same as for:

$$havoc(y_1); \ldots; havoc(y_m)$$

Thus, the order of distinct havoc-s does not matter.

Programs and Specs are Relations

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

Specification:

$$z' = z \land (x > 0 \to (x' > 0 \land y' > 0)$$

Adhering to specification is relation subset:

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

Non-deterministic programs are a way of writing specifications

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$
Formula for relation (talks only about resulting state):

$$z' = z \land x' > 0 \land y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \land y > 0)$$

Formula for relation:

$$z' = z \land x' > x \land y' > y$$

Corresponding program?
Use local variables to store initial values.

```
{ var x0; var y0;
  x0 = x; y0 = y;
  havoc(x,y);
  assume(x > x0 && y > y0)
}
```

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \ldots, x_n\}$
Specification

$$F(x_1, \ldots, x_n, x_1', \ldots, x_n')$$

Becomes

$$\begin{array}{l} \{\ \mathit{var}\ \ y_1, \ldots, y_n; \\ \quad y_1 = x_1; \ldots; y_n = x_n; \\ \quad \mathit{havoc}(x_1, \ldots, x_n); \\ \quad \mathit{assume}(F(y_1, \ldots, y_n, x_1, \ldots, x_n))\ \} \end{array}$$

Program Refinement and Equivalence

For two programs, define **refinement** $P_1 \sqsubseteq P_2$ iff

$$R(P_1) \rightarrow R(P_2)$$

is a valid formula.

(Some books use the opposite meaning of $\sqsubseteq$.)

As usual, $P_2 \sqsupseteq P_1$ iff $P_1 \sqsubseteq P_2$.

- $P_1 \sqsubseteq P_2$ iff $\rho(P_1) \subseteq \rho(P_2)$

Define **equivalence** $P_1 \equiv P_2$ iff $P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$

- $P_1 \equiv P_2$ iff $\rho(P_1) = \rho(P_2)$

Example for $V = \{x, y\}$

$$\{var\ x0; x0 = x; havoc(x); assume(x > x0)\} \sqsupseteq (x = x + 1)$$

Proof: Use R to compute formulas for both sides and simplify.

$$x' = x + 1 \wedge y' = y \ \rightarrow \ x' > x \wedge y' = y$$

Stepwise Refinement Methodology

Start form a possibly non-deterministic specification P_0
Refine the program until it becomes deterministic and efficiently executable.

$$P_0 \sqsupseteq P_1 \sqsupseteq \ldots \sqsupseteq P_n$$

Example:

$$\begin{array}{ll} & havoc(x); assume(x > 0); havoc(y); assume(x < y) \\ \sqsupseteq & havoc(x); assume(x > 0); y = x + 1 \\ \sqsupseteq & x = 42; y = x + 1 \\ \sqsupseteq & x = 42; y = 43 \end{array}$$

In the last step program equivalence holds as well

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \to (p_1 \circ p) \subseteq (p_2 \circ p)$

Theorem: if $P_1 \sqsubseteq P_2$ then $(P; P_1) \sqsubseteq (P; P_2)$
Version for relations: $(p_1 \subseteq p_2) \to (p \circ p_1) \subseteq (p \circ p_2)$

Theorem: if $P_1 \sqsubseteq P_2$ and $Q_1 \sqsubseteq Q_2$ then

$$(\mathit{if}\ (*) P_1\ \mathit{else}\ Q_1) \sqsubseteq (\mathit{if}\ (*) P_2\ \mathit{else}\ Q_2)$$

Version for relations:
$(p_1 \subseteq p_2) \wedge (q_1 \subseteq q_2) \quad \to \quad (p_1 \cup q_1) \subseteq (p_2 \cup q_2)$

Checking Commutativity of Commands

Associativity of Commands

Under what conditions on commands c_1, c_2 is

$$c_1; (c_2; c_3) \;\equiv\; (c_1; c_2); c_3$$

always

Commutativity of Commands

Under what conditions on commands c_1, c_2 is

$$c_1; c_2 \equiv c_2; c_1$$

In general, when the resulting relations are equal and formulas equivalent, i.e. iff

$$R(c_1; c_2) \iff R(c_2; c_1)$$

is a valid formula (true for all variables).
Example: does this hold?

$$(x = x + 1; y = x + 2) \equiv (y = x + 2; x = x + 1)$$

Show formulas for each sides—not equivalent:

$$x' = x + 1 \wedge y' = x + 3 \qquad x' = x + 1 \wedge y' = x + 2$$

Examples of Commutativity of Commands

Show the formula for each example and check if the commutativity equivalence holds

Example 1:

$$(x = 2*x+7*z;\ y = 5*y+z) \ \equiv \ (y = 5*y+z;\ x = 2*x+7*z)$$

Can you state a generalization of the above example?
Example 2:

$$(x = x + 1; x = x + 5) \ \equiv \ (x = x + 5; x = x + 1)$$

Requires knowing properties of $+$.

Preserving Domain in Refinement

What is the domain of a relation?

Given relation $r \subseteq A \times B$ for any sets A, B, we define domain of r as

$$dom(r) = \{a \mid \exists b.\ (a, b) \in r\}$$

when r is a total function, then $dom(r) = A$

- a typical case if r is an entire program

Let $r = \{(\bar{x}, \bar{x}') \mid F\}$, $FV(F) \subseteq Var \cup Var'$, $Var' = \{x' \mid x \in Var\}$. Then, $dom(r) = \{\bar{x} \mid \exists \bar{x}'.F\}$

- computing domain = existentially quantifying over primed vars

Example: for $Var = \{x, y\}$, $R(x = x + 1) = x' = x + 1 \wedge y' = y$. The formula for the domain is: $\exists x', y'.\ x' = x + 1 \wedge y' = y$, which, after one-pint rule, reduces to true.

- All assignments have true as domain.

Preserving Domain

It is not interesting program development step $P \sqsupseteq P'$ is P' is false, or is false for most inputs.
Example ($Var = \{x, y\}$)

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

Refinement $P \sqsupseteq Q$, ensures $R(Q) \to R(P)$. A consequence is $(\exists \bar{x}'.R(Q)) \to (\exists \bar{x}'.R(P))$.
We additionally wish to preserve the *domain* of the relation between $\bar{x}, \bar{x}'$

- if P has some execution from $\bar{x}$ ending in $\bar{x}'$
- then Q should also have some execution, ending in some (possibly different) $\bar{x}'$ (even if it has fewer choices)

$$(\exists \bar{x}'.R(P)) \leftrightarrow (\exists \bar{x}'.R(Q))$$

So, we want relations to be smaller or equal, but domains equal.

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold? yes

Now consider the right hand side:

- domain of P is $\exists x', y'. x' + x' = y \wedge y' = y$
- equivalent to: $y\%2 = 0$
- domain of P is: $\exists x', y'. x' = 3 \wedge y' = 6 \wedge y' = y$
- equivalent to: $y = 6$

Does domain formula of P' imply the domain formula of P? no

Preserving Domain: Exercise

Given P:

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

Find P_1 and P_2 such that

- $P \sqsupseteq P_1 \sqsupseteq P_2$
- no two programs among P, P_1, P_2 are equivalent
- programs P, P_1 and P_2 have equivalent domains
- the relation described by P_2 is a partial function

Synthesis from Relations

Software Synthesis Procedures
Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, Philippe Suter
Communications of the ACM, Vol. 55 No. 2, Pages 103-111
http://doi.org/10.1145/2076450.2076472

Example of Synthesis

Input:

```
val (hours, minutes, seconds) = choose((h: Int, m: Int, s: Int) => (
  h * 3600 + m * 60 + s == totsec
  && 0 <= m && m < 60
  && 0 <= s && s < 60))
```

Output:

```
val (hours, minutes, seconds) = {
val loc1 = totsec div 3600
val num2 = totsec + ((−3600) * loc1)
val loc2 = min(num2 div 60, 59)
val loc3 = totsec + ((−3600) * loc1) + (−60 * loc2)
  (loc1, loc2, loc3)
}
```

Complete Functional Synthesis

Domain-preserving refinement algorithm that produces a partial function

- assignment: $\mathbf{res} = \mathbf{choose\ x.\ F}$
- corresponds to: $\{\mathbf{var\ x}; \mathbf{assume}(\mathbf{F}); \mathbf{res} = \mathbf{x}\}$
- we refine it preserving domain into: $\mathbf{assume}(\mathbf{D}); \mathbf{res} = \mathbf{t}$ (where t does not have 'choose')

More abstractly, given formula F and variable x find

- formula D
- term t not containing x

such that, for all free variables:

- $D \rightarrow F[x := t]$ (t is a term such that refinement holds)
- $D \iff \exists x.F$ (D is the domain, says when t is correct)

Consequence of the definition: $D \iff F[x := t]$

From Quantifier Elimination to Synthesis

Quantifier Elimination

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

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

Synthesis

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

gives:

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

Handling Disjunctions

We had

$$\exists x.(F_1(x) \vee F_2(x))$$

is equivalent to

$$(\exists x.F_1(x)) \vee (\exists x.F_2(x))$$

Now:

$$\textit{choose}\ x.(F_1(x) \vee F_2(x))$$

becomes:

$$\textit{if}\ \ (D_1)\ (\textit{choose}\ x.F_1(x))\ \textit{else}\ (\textit{choose}\ x.F_2(x))$$

where D_1 is the domain, equivalent to $\exists x.F_1(x)$ and computed while computing *choose* $x.F_1(x)$.

Framework for Synthesis Procedures

We define the framework as a transformation

- from specification formula F to
- the maximal domain D where the result x can be found, and the program t that computes the result

$\langle D \mid t \rangle$ denotes: the domain (formula) D and program (term) t
Main transformation relation $\vdash$

$$choose\ x.F \ \vdash \ \langle D \mid t \rangle$$

means

- $D \rightarrow F[x := t]$ (t is a term such that refinement holds)
- $D \iff \exists x.F$ (D is the domain, says when t is correct)

Because $F[x := t]$ implies $\exists x.F$, the above definition implies that D, $F[x := t]$ and $\exists x.F$ are all equivalent.

Rule for Synthesizing Conditionals

$$\frac{\textit{choose } x.F_1 \vdash \langle D_1 \mid t_1 \rangle \quad \textit{choose } x.F_2 \vdash \langle D_2 \mid t_2 \rangle}{\textit{choose } x.(F_1 \vee F_2) \ \vdash \ \langle D_1 \vee D_2 \mid \textit{if } (D_1)\ t_1 \ \textit{else } t_2 \rangle}$$

To synthesize the thing below the — , synthesize the things above and put the pieces together.

Test Terms Methods for Presburger Arithmetic Synthesis

Recall that the most complex step in QE for PA was replacing

$$\exists x. F_1(x)$$

with

$$\bigvee_{k=1}^{L} \bigvee_{i=1}^{N} F_1(a_k + i)$$

Now we transform *choose* $x.F_1(x)$ first into:

$$\mathit{choose}\ x. \bigvee_{k=1}^{L} \bigvee_{i=1}^{N} (x = a_k + i \wedge F_1(x))$$

Then apply:

- rule for conditionals
- one-point rule

Synthesis using Test Terms

$$\textit{choose } x. \bigvee_{k=1}^{L} \bigvee_{i=1}^{N} (x = a_k + i \wedge F_1)$$

produces the same precondition as the result of QE, and the generated term is:

$$\begin{array}{l}
\textit{if } \ (F_1[x := a_1 + 1]) \ a_1 + 1 \\
\textit{elseif } \ (F_1[x := a_1 + 2]) \ a_1 + 2 \\
\ldots \\
\textit{elseif } \ (F_1[x := a_k + i]) \ a_k + i \\
\ldots \\
\textit{elseif } \ (F_1[x := a_L + N]) \ a_L + N
\end{array}$$

Linear search over the possible values, taking the first one that works.
This could be optimized in many cases.

Synthesizing a Tuple of Outputs

$$\frac{\textit{choose } x.F \ \vdash \ \langle D_1 \mid t_1 \rangle \quad \textit{choose } y.D_1 \ \vdash \ \langle D_2 \mid t_2 \rangle}{\textit{choose } (x, y).F \ \vdash \ \langle D_2 \mid (t_1[y := t_2], \ t_2) \rangle}$$

Note that y can appear inside D_1 and t_1, but not in D_2 or t_2

Substitution of Variables

In quantifier elimination, we used a step where we replace $M \cdot x$ with y. Let F be a formula in which x occurs only in the form $M \cdot x$.

What is the corresponding rule?

$$\frac{\mathit{choose}\ y.(F[(M \cdot x) := y] \wedge (M|y)) \ \vdash\ \langle D \mid t \rangle}{\mathit{choose}\ x.F \ \vdash\ \langle D \mid t[y := t/M] \rangle}$$

Automated Checks for Specifications: Uniqueness

Suppose we wish to give a warning if the specification F allows two different solutions.

Let the variables in scope be denoted by z and consider the synthesis problem:

$$choose\ x.\ F$$

What is the verification condition that checks whether the solution for x is unique?
Solution is **not** unique if this PA formula is satisfiable:

$$F \wedge F[x := y] \wedge x \neq y$$

If we find such x, y, z we report z as an example input for which there are two possible outputs, x and y.

Automated Checks for Specifications: Totality

Suppose we wish to give a warning if in some cases the solution does not exist.

Let the variables in scope be denoted by z and consider the synthesis problem:

$$\mathit{choose}\ x.\ F$$

What is the verification condition that checks if there are cases when no solution x exists?
Check satisfiability of this PA formula:

$$\neg\exists x.F$$

If there is a satisfying value for this formula, z, report it as an example for which no solution for x exists.