

Lecture 12

Synthesis from Input-Output Relations

Viktor Kuncak

Programs and Specs are Relations

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

Specification:

$$z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0))$$

Adhering to specification is relation subset:

$$\{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \subseteq \{(x, y, z, x', y', z') \mid z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0))\}$$

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):

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

Corresponding program:

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

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

Corresponding program:

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

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

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

Corresponding program:

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

Formula for relation:

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

Corresponding program?

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

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

Corresponding program:

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

Formula for relation:

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

Corresponding program?

Use local variables to store initial values.

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

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

Corresponding program:

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

Formula for relation:

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

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 $V = \{x_1, \dots, x_n\}$

Specification

$$F(x_1, \dots, x_n, x'_1, \dots, x'_n)$$

Becomes

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \dots, x_n\}$

Specification

$$F(x_1, \dots, x_n, x'_1, \dots, x'_n)$$

Becomes

```
{ var y1, ..., yn;  
  y1 = x1; ...; yn = xn;  
  havoc(x1, ..., xn);  
  assume(F(y1, ..., yn, x1, ..., xn)) }
```

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 $V = \{x, y\}$

$$\{\text{var } x_0; x_0 = x; \text{havoc}(x); \text{assume}(x > x_0)\} \sqsupseteq (x = x + 1)$$

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

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 $V = \{x, y\}$

$$\{\text{var } x_0; x_0 = x; \text{havoc}(x); \text{assume}(x > x_0)\} \sqsupseteq (x = x + 1)$$

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

$$P_0 \sqsupseteq P_1 \sqsupseteq \dots \sqsupseteq P_n$$

Example:

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

In the last step program equivalence holds as well

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

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

when r is a total function, then $\text{dom}(r) = A$

- ▶ a typical case if r is an entire program

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

- ▶ computing domain = existentially quantifying over primed vars

Example: for $\text{Var} = \{x, y\}$, $R(x = x + 1) = x' = x + 1 \wedge y' = y$.
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 $P \sqsupseteq P'$ if P' is false, or is false for most inputs.

Example ($Var = \{x, y\}$)

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

Refinement $P \sqsupseteq Q$, ensures $R(Q) \rightarrow R(P)$. A consequence is $(\exists \bar{x}'. R(Q)) \rightarrow (\exists \bar{x}'. R(P))$.

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 $P \sqsubseteq P'$

$$(havoc(x); assume(x + x = y)) \sqsubseteq (assume(y = 6); x = 3)$$

► $R(P) =$

Domains in the Example

Consider our example $P \sqsubseteq P'$

$$(havoc(x); assume(x + x = y)) \sqsubseteq (assume(y = 6); x = 3)$$

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

Domains in the Example

Consider our example $P \sqsupseteq P'$

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

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

Does $P \sqsupseteq P'$ really hold?

Domains in the Example

Consider our example $P \sqsupseteq P'$

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

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

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

Now consider the right hand side:

- ▶ domain of P is

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$(\text{havoc}(x); \text{assume}(x + x = y)) \sqsupseteq (\text{assume}(y = 6); x = 3)$$

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

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:

Domains in the Example

Consider our example $P \sqsupseteq P'$

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

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

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:

Domains in the Example

Consider our example $P \sqsupseteq P'$

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

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

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:

Domains in the Example

Consider our example $P \sqsupseteq P'$

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

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

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 ?

Domains in the Example

Consider our example $P \sqsupseteq P'$

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

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

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 :

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

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: **res = choose x. F**
- ▶ corresponds to: $\{\mathbf{var\ } x; \mathbf{assume}(F); \mathbf{res} = x\}$
- ▶ we refine it preserving domain into: **assume(D); res = 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

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:

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

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

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

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

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

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

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:

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

Then apply:

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:

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

Then apply:

- ▶ rule for conditionals

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:

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

Then apply:

- ▶ rule for conditionals
- ▶ one-point rule

Synthesis using Test Terms

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

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

if ($F_1[x := a_1 + 1]$) $a_1 + 1$
elseif ($F_1[x := a_1 + 2]$) $a_1 + 2$
...
elseif ($F_1[x := a_k + i]$) $a_k + i$
...
elseif ($F_1[x := a_L + N]$) $a_L + N$

Linear search over the possible values, taking the first one that works.

This could be optimized in many cases.

Synthesizing a Tuple of Outputs

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

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?

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?

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

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?

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:

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:

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

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:

choose x . F

What is the verification condition that checks if there are cases when no solution x exists?

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:

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

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.