Lecture 2
Completing QE for Presburger Arithmetic
Converting Programs to Formulas

Viktor Kuncak

Lower and upper bounds:

Consider the coefficient next to x in $0 < t$. If it is -1, move the term to left side. If it is 1, move the remaining terms to the left side. We obtain formula $F_1(x)$ of the form

$$\bigwedge_{i=1}^{L} a_i < x \wedge \bigwedge_{j=1}^{U} x < b_j \wedge \bigwedge_{i=1}^{D} K_i \mid (x + t_i)$$

If there are no divisibility constraints ($D = 0$), what is the formula equivalent to?

Lower and upper bounds:

Consider the coefficient next to x in $0 < t$. If it is -1, move the term to left side. If it is 1, move the remaining terms to the left side. We obtain formula $F_1(x)$ of the form

$$\bigwedge_{i=1}^{L} a_i < x \wedge \bigwedge_{j=1}^{U} x < b_j \wedge \bigwedge_{i=1}^{D} K_i \mid (x + t_i)$$

If there are no divisibility constraints ($D = 0$), what is the formula equivalent to?

$$\max_i a_i + 1 \leq \min_j b_j - 1 \text{ which is equivalent to } \bigwedge_{ij} a_i + 1 < b_j$$

Replacing variable by test terms

There is a an alternative way to express the above condition by replacing $F_1(x)$ with $\bigvee_k F_1(t_k)$ where t_k do not contain x. This is a common technique in quantifier elimination. Note that if $F_1(t_k)$ holds then certainly $\exists x. F_1(x)$.

What are example terms t_i when $D = 0$ and $L > 0$? Hint: ensure that at least one of them evaluates to $\max a_i + 1$.

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

What if $D > 0$ i.e. we have additional divisibility constraints?

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

What is N? least common multiple of $K_1, \ldots, K_D$

Note that if $F_1(u)$ holds then also $F_1(u - N)$ holds.

Back to Example

$$\exists x. -10 + 10w < x \wedge x < 90 + 15z \wedge 24 \mid x + 6 \wedge 30 | x$$

Back to Example

$$\exists x. -10 + 10w < x \land x < 90 + 15z \land 24 \mid x + 6 \land 30 | x$$

$$\bigvee_{i=1}^{120} 10w + i < 100 + 15z \land 0 < i \land 24 \mid 10w - 4 + i \land 30 | 10w - 10 + i$$

Special cases

What if $L = 0$? We first drop all constraints except divisibility, obtaining $F_2(x)$

$$\bigwedge_{i=1}^{D} K_i \mid (x + t_i)$$

and then eliminate quantifier as

$$\bigvee_{i=1}^{N} F_2(i)$$

It works

We finished describing a complete quantifier elimination algorithm for Presburger Arithmetic!

It works

We finished describing a complete quantifier elimination algorithm for Presburger Arithmetic!
This algorithm and its correctness prove that:

- PA admits quantifier elimination
- Satisfiability, validity, entailment, equivalence of PA formulas is decidable
  We can use the algorithm to prove verification conditions. Even if not the most efficient way, it gives us insights on which we can later build to come up with better algorithms.
- Quantified and quantifier-free formulas have the same expressive power

Many other properties follow (e.g. interpolation).

Interpolation For Logical Theories

Interpolation can be useful in generalizing counterexamples to invariants.

Universal **Entailment**: we will write $F_1 \models F_2$ to denote that for all free variables of F_1 and F_2, if F_1 holds then F_2 holds.

Given two formulas such that

$$F_0(\bar{x}, \bar{y}) \models F_1(\bar{y}, \bar{z})$$

an interpolant for F_1, F_2 is a formula $I(\bar{y})$, which has only variables common to F_0 and F_1, such that

- $F_0(\bar{x}, \bar{y}) \models I(\bar{y})$, and
- $I(\bar{y}) \models F_1(\bar{y}, \bar{z})$

In other words, the entailment between F_0 and F_1 can be explained through $I(\bar{y})$.

Logic has **interpolation property** if, whenever $F_0 \models F_1$, then there exists an interpolant for F_0, F_1.

We often wish to have *simple* interpolants, for example ones that are quantifier free.

Quantifier Elimination Implies Interpolation

If logic has QE, it also has quantifier-free interpolants.
Consider the formula

$$\forall \bar{x}, \bar{y}, \bar{z}.\ F_0(\bar{x}, \bar{y}) \rightarrow F_1(\bar{y}, \bar{z})$$

pushing $\bar{x}$ into assumption we get

$$\forall \bar{y}, \bar{z}.\ (\exists \bar{x}. F_0(\bar{x}, \bar{y})) \rightarrow F_1(\bar{y}, \bar{z})$$

and pushing $\bar{z}$ into conclusion we get

$$\forall \bar{x}, \bar{y}.\ F_0(\bar{x}, \bar{y}) \rightarrow (\forall \bar{z}. F_1(\bar{y}, \bar{z}))$$

Given two formulas F_0 and F_1, each of the formulas satisfies properties of interpolation:

- $\exists \bar{x}. F_0(\bar{x}, \bar{y})$
- $\forall \bar{z}. F_1(\bar{y}, \bar{z})$

Applying QE to them, we obtain quantifier-free interpolants.

More on QE: One Direction to Make it More Efficient

Avoid transforming to conjunctions of literals: work directly on negation-normal form. The technique is similar to what we described for conjunctive normal form.

+ no need for DNF
- we may end up trying irrelevant bounds

This is the Cooper's algorithm:

- Reddy, Loveland: Presburger Arithmetic with Bounded Quantifier Alternation. (Gives a slight improvement of the original Cooper's algorithm.)
- Section 7.2 of the Calculus of Computation Textbook

Eliminate Quantifiers: Example

$$\exists y. \exists x. \;\; x < -2 \wedge 1 - 5y < x \wedge 1 + y < 13x$$

Check whether the formula is satisfiable

$$x < y + 2 \ \wedge \ y < x + 1 \wedge x = 3k \wedge (y = 6p + 1 \vee y = 6p - 1)$$

Apply quantifier elimination

$$\exists x. \;\; (3x + 1 < 10 \vee 7x - 6 < 7) \;\wedge\; 2 \mid x$$

Another Direction for Improvement

Handle a system of equalities more efficiently, without introducing divisibility constraints too eagerly.

Hermite normal form of an integer matrix.

Eliminate variables x and y

$$5x + 7y = a \wedge x \leq y \wedge 0 \leq x$$

Quantifier Elimination for Linear Rational Arithmetic

Consider first-order formulas with equality and $<$ relation, interpreted over rationals.
This theory is called **dense linear order without endpoints**
For example:

$$\forall \varepsilon . \exists \delta . \left(|x_1 - x_2| < \delta \wedge |y_1 - y_2| < \delta \rightarrow |3x_1 + 4y_1 - 3x_2 - 4y_2| < \varepsilon \right)$$

(i) Show that absolute value can be defined in first-order logic in terms of other linear operations and comparison.

Quantifier Elimination for Linear Rational Arithmetic

Consider first-order formulas with equality and $<$ relation, interpreted over rationals.
This theory is called **dense linear order without endpoints**
For example:

$$\forall \varepsilon . \exists \delta . \big(|x_1 - x_2| < \delta \wedge |y_1 - y_2| < \delta \rightarrow |3x_1 + 4y_1 - 3x_2 - 4y_2| < \varepsilon \big)$$

(i) Show that absolute value can be defined in first-order logic in terms of other linear operations and comparison.
Answer: replace $F(|t|)$ with, for example

$$(t > 0 \wedge F(t)) \vee (\neg(t > 0) \wedge F(-t))$$

Is there a way to remove $|...|$ while increasing formula size only linearly?

Quantifier Elimination for Linear Rational Arithmetic

Consider first-order formulas with equality and $<$ relation, interpreted over rationals.
This theory is called **dense linear order without endpoints**
For example:

$$\forall \varepsilon . \exists \delta . \big(|x_1 - x_2| < \delta \land |y_1 - y_2| < \delta \to |3x_1 + 4y_1 - 3x_2 - 4y_2| < \varepsilon \big)$$

(i) Show that absolute value can be defined in first-order logic in terms of other linear operations and comparison.
Answer: replace $F(|t|)$ with, for example

$$(t > 0 \land F(t)) \lor (\neg(t > 0) \land F(-t))$$

Is there a way to remove $|...|$ while increasing formula size only linearly?
(ii) Give quantifier elimination algorithm for this theory.

Quantifier Elimination for Linear Rational Arithmetic

Consider first-order formulas with equality and $<$ relation, interpreted over rationals.
This theory is called **dense linear order without endpoints**
For example:

$$\forall \varepsilon . \exists \delta . \big(|x_1 - x_2| < \delta \wedge |y_1 - y_2| < \delta \rightarrow |3x_1 + 4y_1 - 3x_2 - 4y_2| < \varepsilon \big)$$

(i) Show that absolute value can be defined in first-order logic in terms of other linear operations and comparison.
Answer: replace $F(|t|)$ with, for example

$$(t > 0 \wedge F(t)) \vee (\neg(t > 0) \wedge F(-t))$$

Is there a way to remove $|...|$ while increasing formula size only linearly?
(ii) Give quantifier elimination algorithm for this theory.
Solution is simpler than for Presburger arithmetic—no divisibility.

From (Integer) Programs to Formulas

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":

$$\big[((x > 0) \land res = 2x + 1) \lor (\neg(x > 0) \land res = 42)\big] \;\rightarrow\; res > 0$$

For above formula, we would check *validity*: all variables are universally quantified

What Relations Can Represent

Let r be $\rho(c)$, the relation describing the command c.
For an initial state s, we can compute the set of states that the system can end up after executing c:

$$r[\{s\}] = \{s' \mid (s, s') \in r\}$$

This set of states can be

- a singleton set $\{s'\}$, meaning that precisely one result is possible
-

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,b : Boolean, t : Int

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

We show how to translate x/K, $x\%K$, **if**, **val** into other constructs

- without changing the meaning of a formula
- without adding alternations of quantifiers
- in time polynomial in input
  (result is thus also in polynomial size)

Reminder: Free Variables and Substitutions

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:

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

If it occurs quantified somewhere it can still be free overall:

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

Rules for FV are of two kinds: operations $\odot$ (e.g., $\land$, $<$, $+$) and binders Q (e.g. $\forall$, $\exists$, val)

$$\begin{array}{l} FV(x) = \{x\},\ \text{if } x \text{ is a variable} \\ FV(F_1 \odot F_2) = FV(F_1) \cup FV(F_2) \\ FV(Qx.F) = FV(F) \setminus \{x\} \end{array}$$

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.)
Our notation: write F, and instead of $F(t)$ write $F[x := t]$

- closer to a typical implementation

Definition of substitution:

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

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

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

Summary of Our Translation Goal

Transform logic of this grammar
F,b : Boolean, t : Int

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

Into a logic for which we did quantifier elimination, which omits the bold symbols:

- val (let) definitions in formulas and terms
- conditionals
- division by a constant
- computing modulo by a constant as a term

About val Definitions

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

Equivalent ways of saying:

- in the rest of the block, introduce read-only variable x with value equal to t
- let x have the value t in E (written so in ML, Haskell)
- E, where x has the value E (math, Haskell's where clause)
- in lambda calculus: $(\lambda x.E)t$

Slightly different cases depending on whether types of t and E (each of which can be Boolean or Int)

Note: x is bound to t inside E, but not inside t or anywhere else

Free Variables and Substitution for val

Computing free variable:

$$FV(\{val\ x = t;\ E\}) = FV(t) \cup (FV(E) \setminus \{x\})$$

Substitution, for $x \neq y$, $x \notin FV(s)$ (otherwise, rename x):

$$(\{val\ x = t;\ E\})[y := s] = \{val\ x = t[y := s];\ (E[y := s])\}$$

Renaming means transforming $\{val\ x = t;\ E\}$ into $\{val\ x_1 = t;\ E[x := x_1]\}$ where x_1 is different from other relevant variables (clear from the context)

How to Translate Value Definitions

Construct: $\{val\ x = t;\ F\}$ where we additionally require $x \notin FV(t)$ (otherwise just rename x)

Example

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

Becomes one of these:

How to Translate Value Definitions

Construct: $\{val\ x = t;\ F\}$ where we additionally require $x \notin FV(t)$ (otherwise just rename x)

Example

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

Becomes one of these:

$(y+1) < 2(y+1) + 5$	substitution
$\exists x.\ x = y + 1 \wedge x < 2x + 5$	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:

$$\begin{array}{ll} F[x := t] & \text{substitution} \\ \exists x.\ x = t \wedge F & \text{one-point rule} \\ \forall x.\ x = t \rightarrow F & \text{dual one-point rule} \end{array}$$

Substitution can square formula size

- Do it several times $\leadsto$ exponential increase

The other rules add quantified variables

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

Dual of val elimination is flattening: remove 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,b : Boolean, t : Int

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

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

$$(b_1 \wedge x = t_1) \vee (\neg b_1 \wedge x = t_2)$$

as well as to:

$$((b_1 \rightarrow x = t_1) \wedge (\neg b_1 \rightarrow x = t_2))$$

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:

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

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:

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

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, /, % :

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

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:

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

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

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

has no satisfying assignments.

Adding State and Non-Determinism

VC Generation for Imperative Non-Deterministic Programs

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

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

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

or validity of the following implication:

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

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

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

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

Formula for Assignment

$$x = t$$

Formula for Assignment

$$x = t$$

$R(x = t)$:

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

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.

Formula for if-else

After flattening,

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

Formula for if-else

After flattening,

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

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

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

Command semicolon

$$c_1\,;\,c_2$$

Command semicolon

$$c_1\,; c_2$$

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:

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

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

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

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

- a useful convention: $\bar{z}$ refer to position in program source code

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, $R(havoc(x))$:

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

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

$$assume(F)$$

$R(assume(F))$:

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

assume

$$assume(F)$$

$R(assume(F))$:

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

- This command does not change any state.

assume

$$assume(F)$$

$R(assume(F))$:

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

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

Example of Translation

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

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')

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(x)**: non-deterministically change $x \in V$ to have an arbitrary value; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if}(*)\ \mathbf{c_1\ else\ c_2}$: arbitrarily choose to run c_1 or c_2
- **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 $x'_1, \ldots, x'_n$

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. $R(assume(F); c) \quad = \quad F \wedge R(c)$
2. $R(c; assume(F))$

Justifying the name for assume(F)

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

1. $R(assume(F); c) \quad = \quad F \wedge R(c)$
2. $R(c; assume(F)) \quad = \quad R(c) \wedge F[\bar{x} := \bar{x}']$
   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: $\mathit{havoc}(x);\ \mathit{assume}(x == x + 1)$

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