From Program Verification to Program Synthesis

Overview

Jaak Ristioja

March 30, 2010

Reference

From Program Verification to Program Synthesis.

@ POPL'10; January 17-23, 2010

Saurabh Srivastava,
University of Maryland, College Park
Sumit Gulwani,
Microsoft Research, Redmond
Jeffrey S. Foster
University of Maryland, College Park

doi:10.1145/1706299.1706337

(including numerous typos and ambiguities)

Reference

From Program Verification to Program Synthesis.

@ POPL'10; January 17-23, 2010

Saurabh Srivastava,
University of Maryland, College Park
Sumit Gulwani,
Microsoft Research, Redmond
Jeffrey S. Foster
University of Maryland, College Park

doi:10.1145/1706299.1706337

(including numerous typos and ambiguities)

Introduction

Automated program synthesis

- Correct-by-construction
- Eases task of programming
  - Automated debugging
  - Programmer only deals with high-level design
- New non-trivial algorithms could be discovered
- Difficult to implement

Program verification

- synthesizes program proofs from programs
- for loops it uses
  - inductive invariants for partial correctness
  - ranking functions for termination
- does verification

Synthesis problem $\rightarrow$ verification problem

- encoding guards and statements etc as logical facts
- using verification tools for synthesis
- by verification we infer statements, guards etc

Proof-theoretic synthesis

- Proof for the program is synthesized alongside the program

Introduction

Verification and synthesis

Program verification

- synthesizes program proofs from programs
- for loops it uses
  - inductive invariants for partial correctness
  - ranking functions for termination
- does verification

Synthesis problem $\longrightarrow$ verification problem

- encoding guards and statements etc as logical facts
- using verification tools for synthesis
- by verification we infer statements, guards etc

Proof-theoretic synthesis

- Proof for the program is synthesized alongside the program

Introduction

Verification and synthesis

Program verification

- synthesizes program proofs from programs
- for loops it uses
  - inductive invariants for partial correctness
  - ranking functions for termination
- does verification

Synthesis problem $\longrightarrow$ verification problem

- encoding guards and statements etc as logical facts
- using verification tools for synthesis
- by verification we infer statements, guards etc

Proof-theoretic synthesis

- Proof for the program is synthesized alongside the program

Motivating example

Bresenham's line drawing algorithm

Pre- and post-condition for a line drawing program:

$$\tau_{pre}: \quad 0 < Y \leq X$$
$$\tau_{post}: \quad \forall k : 0 \leq k \leq X \Rightarrow 2|out[k] - (Y/X)k| \leq 1$$

and resource constraints, for example constraints for

- control flow,
- stack space,
- available operations, etc

can we synthesize the program?

Motivating example

Bresenham's line drawing algorithm

Given the specification for a line drawing program

$$\tau_{pre}: \quad 0 < Y \leq X$$
$$\tau_{post}: \quad \forall k : 0 \leq k \leq X \Rightarrow 2|out[k] - (Y/X)k| \leq 1$$

and resource constraints, for example constraints for

- control flow,
- stack space,
- available operations, etc

can we synthesize the program?

Motivating example

Bresenham's line drawing algorithm

Example

```
Bresenhams(int X, int Y)
    v1 := 2Y − X; y := 0; x := 0;
    while (x <= X)
    |    out[x] := y;
    |    if (v1 < 0)
    |    |    v1 := v1 + 2Y;
    |    else
    |    |    v1 := v1 + 2(Y − X); y++;
    |    x++;
    return out;
```

Motivating example

Bresenham's line drawing algorithm

Observations

- We can write statements as equality predicates
- We can write acyclic program fragments as transition systems

Example

- x := e becomes an equality predicate $x' = e$ where
  - x' is a renaming of x to its output value
  - e is the expression over the non-primed values
- y := x; x := y becomes $y' = x \land x' = y'$
- **if** (x > 0) x := y; **else skip**; becomes

$$[]\, x > 0 \rightarrow x' = y$$
$$[]\, x \leq 0 \rightarrow \textbf{true}$$

Motivating example

Bresenham's line drawing algorithm

Example

```
[]true  →  v1' = 2Y − X  ∧  y' = 0  ∧  x' = 0
while (x <= X)
|      []v1 < 0  →  out' = upd(out, x, y)  ∧
|                   v1' = v1 + 2Y ∧
|                   y' = y ∧
|                   x' = x + 1
|      []v1 ≥ 0  →  out' = upd(out, x, y)  ∧
|                   v1' = v1 + 2(Y − X)  ∧
|                   y' = y + 1 ∧
|                   x' = x + 1
```

Motivating example

Bresenham's line drawing algorithm

To prove partial correctness, we can write down the inductive loop invariant for the **while**-loop:

$$
\begin{aligned}
\tau : 0 < Y \leq X \land \\
v_1 = 2\,(x+1)\,Y - (2y+1)\,X \land \\
2\,(Y - X) \leq v_1 \leq 2Y \land \\
\forall k : 0 \leq k < x \Rightarrow 2\,|out[k] - (Y/X)k| \leq 1
\end{aligned}
$$

and the verification condition can be written as four implications of four paths in the program:

$$
\begin{aligned}
\tau_{pre} \land s_{entry} &\Rightarrow \tau' \\
\tau \land \neg g_{loop} &\Rightarrow \tau_{post} \\
\tau \land g_{loop} \land g_{body1} \land s_{body1} &\Rightarrow \tau' \\
\tau \land g_{loop} \land g_{body2} \land s_{body2} &\Rightarrow \tau'
\end{aligned}
$$

where τ' is the renamed version of the loop invariant.

Motivating example

Bresenham's line drawing algorithm

$$
\begin{aligned}
s_{entry} &: v_1' = 2Y - X \;\wedge\; y' = 0 \;\wedge\; x' = 0 \\
g_{loop} &: x \leq X \\
g_{body1} &: v_1 < 0 \\
s_{body1} &: out' = \mathrm{upd}(out, x, y) \;\wedge \\
&\quad v_1' = v_1 + 2Y \;\wedge \\
&\quad y' = y \;\wedge \\
&\quad x' = x + 1 \\
g_{body2} &: v_1 \geq 0 \\
s_{body2} &: out' = \mathrm{upd}(out, x, y) \;\wedge \\
&\quad v_1' = v_1 + 2(Y - X) \;\wedge \\
&\quad y' = y + 1 \;\wedge \\
&\quad x' = x + 1
\end{aligned}
$$

Motivating example

One can *validate* that the loop invariant τ satisfies the verification condition.

- e.g. by using SMT (Satisfiability Modulo Theory) solvers

There are also powerful program verification tools that can prove total correctness by

- automatically generating fixed-point solutions for loop invariants, such as τ
- inferring ranking functions (φ) to prove termination

So if we can infer the verification condition, perhaps we can also infer

- the guards g_i's and
- the statements s_i's

at the same time?

Motivating example

One can *validate* that the loop invariant τ satisfies the verification condition.

- e.g. by using SMT (Satisfiability Modulo Theory) solvers

There are also powerful program verification tools that can prove total correctness by

- automatically generating fixed-point solutions for loop invariants, such as τ
- inferring ranking functions (φ) to prove termination

So if we can infer the verification condition, perhaps we can also infer

- the guards g_i's and
- the statements s_i's

at the same time?

Motivating example

How to infer guards and statements

1. encode programs as transition systems
2. assert appropriate constraints
3. use verification tools to systematically infer solutions for the unknowns in the constraints. The unknowns are
   - invariants
   - statements
   - guards

Types of constraints

- well-formedness constraints to get solutions corresponding to real-life programs
- progress constraints to ensure termination

Motivating example

How to infer guards and statements

1. encode programs as transition systems
2. assert appropriate constraints
3. use verification tools to systematically infer solutions for the unknowns in the constraints. The unknowns are
   - invariants
   - statements
   - guards

Types of constraints

- well-formedness constraints to get solutions corresponding to real-life programs
- progress constraints to ensure termination

Specification for proof-theoretic approach

For synthesis we first need a specification for the program we want to construct.

Synthesis scaffold

$$\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$$

- $\mathcal{F}$ - functional specification
- $\mathcal{D}$ - domain constraints
- $\mathcal{R}$ - resource constraints

Specification for proof-theoretic approach

Synthesis scaffold

Functional specification $\mathcal{F}$

Let $\vec{v_{in}}$ and $\vec{v_{out}}$ be vectors containing the input and output variables.

$$\mathcal{F} = \left(F_{pre}\left(\vec{v_{in}}\right), F_{post}\left(\vec{v_{out}}\right)\right)$$

where $F_{pre}\left(\vec{v_{in}}\right)$ and $F_{post}\left(\vec{v_{out}}\right)$ are formulas that hold at the program entry and exit locations, respectively.

Specification for proof-theoretic approach

Synthesis scaffold

Domain constraints $\mathcal{D}$

$$\mathcal{D} = (D_{exp}, D_{grd})$$

where D_{exp} is the domain of expressions in the program and D_{grd} is the domain of boolean expressions used in program guards.

Proof domain D_{prf}

- Proof-theoretic synthesis needs to synthesize proof terms from a proof domain D_{prf}.
- D_{prf} needs to be **at least** as expressive as D_{exp} and D_{grd}.
- We need a solver capable of handling D_{prf}.

Specification for proof-theoretic approach

Synthesis scaffold

Domain constraints $\mathcal{D}$

$$\mathcal{D} = (D_{exp}, D_{grd})$$

where D_{exp} is the domain of expressions in the program and D_{grd} is the domain of boolean expressions used in program guards.

Proof domain D_{prf}

- Proof-theoretic synthesis needs to synthesize proof terms from a proof domain D_{prf}.
- D_{prf} needs to be **at least** as expressive as D_{exp} and D_{grd}.
- We need a solver capable of handling D_{prf}.

Specification for proof-theoretic approach

Synthesis scaffold

Resource constraints $\mathcal{R}$

$$\mathcal{R} = (R_{flow}, R_{stack}, R_{comp})$$

- R_{flow} is a flowgraph template from the grammar $T ::= \circ \mid *(T) \mid T; T$
- $R_{stack} : type \rightarrow \mathbb{N}_1$ is a mapping indicating the number of extra temporary variables of each type available to the program.
- $R_{comp} : op \rightarrow \mathbb{N}_0$ is a mapping defining how many operations of each type can be included in the program. $R_{comp} = \emptyset$ indicates no constraints.

Specification for proof-theoretic approach

Synthesis scaffold

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- D_{exp} limited to linear arithmetic (LA) expressions (no $\sqrt{\ }$)
- D_{grd} limited to quantifier-free first-order logic (FOL) over LA
- $R_{flow} = (\circ; *(\circ); \circ)$, $R_{stack} = \{(\mathbf{int}, 1)\}$, $R_{comp} = \emptyset$

```
IntSqrt(int x)
    v := 1; i := 1;
    while^{τ,φ} (v ≤ x)
    |     v := v + 2i + 1;  i++;
    return i − 1;
```

- Invariant $\tau : v = i^2 \wedge x \geq (i-1)^2 \wedge i \geq 1$
- Ranking function $\varphi : x - (i-1)^2$

Synthesis conditions

Transition systems for acyclic code

One way to infer a set of acyclic statements that transform a precondition to a postcondition would be to use assignments:

$$\{\phi_{pre}\}\, x := e_x;\, y := e_y;\, \{\phi_{post}\}$$

Using Hoare's axiom for assignment, we can generate the assignment condition

$$\phi_{pre} \Rightarrow (\phi_{post}\,[x \mapsto e_x])\,[y \mapsto e_y]$$

Shortcomings in respect to our task:

- substitutions are hard to reason about
- order of assignment matters
- we need more than a fixed number of statements

Synthesis conditions

Transition systems for acyclic code

One way to infer a set of acyclic statements that transform a precondition to a postcondition would be to use assignments:

$$\{\phi_{pre}\}\, x := e_x;\, y := e_y;\, \{\phi_{post}\}$$

Using Hoare's axiom for assignment, we can generate the assignment condition

$$\phi_{pre} \Rightarrow (\phi_{post}\,[x \mapsto e_x])\,[y \mapsto e_y]$$

Shortcomings in respect to our task:

- substitutions are hard to reason about
- order of assignment matters
- we need more than a fixed number of statements

Synthesis conditions

Transition systems for acyclic code

One way to infer a set of acyclic statements that transform a precondition to a postcondition would be to use assignments:

$$\{\phi_{pre}\}\, x := e_x;\, y := e_y;\, \{\phi_{post}\}$$

Using Hoare's axiom for assignment, we can generate the assignment condition

$$\phi_{pre} \Rightarrow (\phi_{post}\,[x \mapsto e_x])\,[y \mapsto e_y]$$

Shortcomings in respect to our task:

- substitutions are hard to reason about
- order of assignment matters
- we need more than a fixed number of statements

Synthesis conditions

Transition systems for acyclic code

Transitions

A transition is a (possibly parallel) mapping of input variables (x) to output variables (x').

$$\{\phi_{pre}\} \left\langle x', y' \right\rangle = \langle e_x, e_y \rangle \{\phi'_{post}\}$$

Corresponding verification condition:

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y \Rightarrow \phi'_{post}$$

Every assignment (state update) can be written as a single transition

Example

For $x := e_x; y := e_y$ we will have

$$\{\phi_{pre}\} \left\langle x', y' \right\rangle = \langle e_x, e_y[x \mapsto e_x] \rangle \{\phi'_{post}\}$$

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post}$$

Synthesis conditions

Transition systems for acyclic code

Transitions

A transition is a (possibly parallel) mapping of input variables (x) to output variables (x').

$$\{\phi_{pre}\}\left\langle x', y'\right\rangle = \langle e_x, e_y\rangle \left\{\phi'_{post}\right\}$$

Corresponding verification condition:

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y \Rightarrow \phi'_{post}$$

Every assignment (state update) can be written as a single transition

Example

For $x := e_x; y := e_y$ we will have

$$\{\phi_{pre}\}\left\langle x', y'\right\rangle = \langle e_x, e_y[x \mapsto e_x]\rangle \left\{\phi'_{post}\right\}$$

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post}$$

Synthesis conditions

Transition systems for acyclic code

Transitions

A transition is a (possibly parallel) mapping of input variables (x) to output variables (x').

$$\{\phi_{pre}\} \left\langle x', y' \right\rangle = \langle e_x, e_y \rangle \left\{\phi'_{post}\right\}$$

Corresponding verification condition:

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y \Rightarrow \phi'_{post}$$

Every assignment (state update) can be written as a single transition

Example

For $x := e_x; y := e_y$ we will have

$$\{\phi_{pre}\} \left\langle x', y' \right\rangle = \langle e_x, e_y[x \mapsto e_x] \rangle \left\{\phi'_{post}\right\}$$

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post}$$

Synthesis conditions

Transition systems for acyclic code

Transitions

A transition is a (possibly parallel) mapping of input variables (x) to output variables (x').

$$\{\phi_{pre}\} \left\langle x', y' \right\rangle = \langle e_x, e_y \rangle \left\{\phi'_{post}\right\}$$

Corresponding verification condition:

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y \Rightarrow \phi'_{post}$$

Every assignment (state update) can be written as a single transition

Example

For $x := e_x; y := e_y$ we will have

$$\{\phi_{pre}\} \left\langle x', y' \right\rangle = \langle e_x, e_y[x \mapsto e_x] \rangle \left\{\phi'_{post}\right\}$$

$$\phi_{pre} \wedge x' = e_x \wedge y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post}$$

Synthesis conditions

Transition systems for acyclic code

Guarded transitions

Lets extend transitions with guarded transitions $[]\, g \rightarrow s$ meaning that statements s are only executed if the quantifier-free g holds.

Transition systems

We can represent arbitrary acyclic program fragments using sets of guarded transitions:

$$\{\phi_{pre}\}\, \{[]\, g_i \rightarrow s_i\}_i\, \{\phi'_{post}\}$$

The corresponding verification for is:

$$\bigwedge_i (\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \phi'_{post})$$

Note that this is much simpler:

- no reasoning about statement ordering to puzzle us
- guards g_i and statements s_i are facts just like pre- and postconditions.

Synthesis conditions

Transition systems for acyclic code

Guarded transitions

Lets extend transitions with guarded transitions $[]\, g \rightarrow s$ meaning that statements s are only executed if the quantifier-free g holds.

Transition systems

We can represent arbitrary acyclic program fragments using sets of guarded transitions:

$$\{\phi_{pre}\}\,\{[]\, g_i \rightarrow s_i\}_i\,\{\phi'_{post}\}$$

The corresponding verification for is:

$$\bigwedge_i \left(\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \phi'_{post}\right)$$

Note that this is much simpler:

- no reasoning about statement ordering to puzzle us
- guards g_i and statements s_i are facts just like pre- and postconditions.

Synthesis conditions

Transition systems for acyclic code

Guarded transitions

Lets extend transitions with guarded transitions $[]\, g \rightarrow s$ meaning that statements s are only executed if the quantifier-free g holds.

Transition systems

We can represent arbitrary acyclic program fragments using sets of guarded transitions:

$$\{\phi_{pre}\}\,\{[]\, g_i \rightarrow s_i\}_i\,\{\phi'_{post}\}$$

The corresponding verification for is:

$$\bigwedge_i \left(\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \phi'_{post}\right)$$

Note that this is much simpler:

- no reasoning about statement ordering to puzzle us
- guards g_i and statements s_i are facts just like pre- and postconditions.

Synthesis conditions

Transition systems for acyclic code

Guarded transitions

Lets extend transitions with guarded transitions $[]\, g \rightarrow s$ meaning that statements s are only executed if the quantifier-free g holds.

Transition systems

We can represent arbitrary acyclic program fragments using sets of guarded transitions:

$$\{\phi_{pre}\}\,\{[]\, g_i \rightarrow s_i\}_i\,\{\phi'_{post}\}$$

The corresponding verification for is:

$$\bigwedge\nolimits_i \left(\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \phi'_{post}\right)$$

Note that this is much simpler:

- no reasoning about statement ordering to puzzle us
- guards g_i and statements s_i are facts just like pre- and postconditions.

Synthesis conditions

- Program verification tools find fixed-point solutions (invariants) to satisfy verification conditions
  - These conditions have known statements and guards.
- For synthesis, we need to generalize this problem
  - We make statements and guards also unknowns in the formulas.

Synthesis conditions

- Program verification tools find fixed-point solutions (invariants) to satisfy verification conditions
  - These conditions have known statements and guards.
- For synthesis, we need to generalize this problem
  - We make statements and guards also unknowns in the formulas.

- Verification conditions for verification
- Synthesis conditions for synthesis

Synthesis conditions

- Program verification tools find fixed-point solutions (invariants) to satisfy verification conditions
  - These conditions have known statements and guards.
- For synthesis, we need to generalize this problem
  - We make statements and guards also unknowns in the formulas.

- If a program is correct (verifiable), then its verification condition is valid.
- If a valid program exists for a scaffold, then its synthesis condition has a satisfying solution.

Synthesis

Expanding the flowgraph

Transition system language (TSL)

$$
\begin{aligned}
p \;::=\;& \textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i \\
\mid\;& \textbf{while}^{\tau,\varphi}\ (g)\ \{p\} \\
\mid\;& p;\ p
\end{aligned}
$$

Synthesis

Expand function

$$Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(\circ) = \textbf{choose} \ \{[]\, g_i \rightarrow s_i\}_{i=1\ldots n}$$

$$Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(*(T)) = \textbf{while}^{\tau,\varphi} \ (g) \ \left\{Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T)\right\}$$

$$Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T_1; T_2) = Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T_1)\ ;\ Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T_2)$$

where all g_i, s_i, g, τ and φ are new generated unknowns.

$$s \in \bigwedge_i x_i = e_i \qquad \text{where } x_i \in V,\ e_i \in D_{exp}|_V$$

$$\tau \in D_{prf}|_V \qquad g \in D_{grd}|_V$$

and $V = \vec{v_{in}} \cup \vec{v_{out}} \cup T \cup L$ where

- T is subject to R_{stack}
- e_i is subject to R_{comp}
- L is the set of iteration counters and ranking function tracker variables

Synthesis

Expanding the flowgraph

Expand function

$$Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(\circ) = \textbf{choose } \{[]\, g_i \rightarrow s_i\}_{i=1\ldots n}$$

$$Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(*(T)) = \textbf{while}^{\tau,\varphi}\ (g)\ \left\{Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T)\right\}$$

$$Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T_1; T_2) = Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T_1)\ ;\ Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(T_2)$$

where all g_i, s_i, g, τ and φ are new generated unknowns and

$$s \in \bigwedge_i x_i = e_i \qquad \text{where } x_i \in V,\ e_i \in D_{exp}|_V$$

$$\tau \in D_{prf}|_V \qquad g \in D_{grd}|_V$$

and $V = \vec{v_{in}} \cup \vec{v_{out}} \cup T \cup L$ where

- T is subject to R_{stack}
- e_i is subject to R_{comp}
- L is the set of iteration counters and ranking function tracker variables

Synthesis

Expanding the flowgraph

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- D_{exp} limited to linear arithmetic (LA) expressions (no $\sqrt{\ }$)
- D_{grd} limited to quantifier-free first-order logic (FOL) over LA
- $R_{flow} = (\circ; *(\circ)\,; \circ)$, $R_{stack} = \{(\mathbf{int}, 1)\}$, $R_{comp} = \emptyset$

For $n = 1$ and FOL over quadratic expressions as D_{prf} we get:

$$exp_{sqrt} = Expand_{\mathcal{D},\mathcal{R}}^{n,D_{prf}}(R_{flow}) =$$
$$\textbf{choose}\ \{[]\, g_1 \to s_1\}\,;$$
$$\textbf{while}^{\tau,\varphi}\ (g_0)\ \{\ \textbf{choose}\ \{[]\, g_2 \to s_2\}\,;\ \}\,;$$
$$\textbf{choose}\ \{[]\, g_3 \to s_3\}\,;$$

where $\vec{v_{in}} = \vec{v_{out}} = \{x\}$, $T = \{v\}$, $L = \{i, r\}$.

Synthesis

Expanding the flowgraph

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- D_{exp} limited to linear arithmetic (LA) expressions (no $\sqrt{}$)
- D_{grd} limited to quantifier-free first-order logic (FOL) over LA
- $R_{flow} = (\circ; *(\circ)\,;\circ)$, $R_{stack} = \{(\textbf{int}, 1)\}$, $R_{comp} = \emptyset$

For $n = 1$ and FOL over quadratic expressions as D_{prf} we get:

$$
\begin{aligned}
exp_{sqrt} = Expand^{n,D_{prf}}_{\mathcal{D},\mathcal{R}}(R_{flow}) = \\
\quad \textbf{choose}\ \{[]\, g_1 \rightarrow s_1\}\,; \\
\quad \textbf{while}^{\tau,\varphi}\ (g_0)\ \{\ \textbf{choose}\ \{[]\, g_2 \rightarrow s_2\}\,;\ \}\,; \\
\quad \textbf{choose}\ \{[]\, g_3 \rightarrow s_3\}\,;
\end{aligned}
$$

where $\vec{v_{in}} = \vec{v_{out}} = \{x\}$, $T = \{v\}$, $L = \{i, r\}$.

Synthesis

Safety conditions

To encode a formula for the validity of a Hoare triple, we define

$$PathC : \phi \times \textsc{Tsl} \times \phi \to \phi$$

which takes a precondition, a sequence of statements and a postcondition, and returns the safety condition.

$$PathC(\phi_{pre}, \textbf{choose}\ \{[]\, g_i \to s_i\}_i, \phi_{post}) =$$
$$\bigwedge_i (\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \phi'_{post})$$
$$PathC(\phi_{pre}, \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p}_l\}, \phi_{post}) =$$
$$\phi_{pre} \Rightarrow \tau' \wedge PathC(\tau \wedge g, \vec{p}_l, \tau) \wedge (\tau \wedge \neg g \Rightarrow \phi'_{post})$$

Encoding sequences of statements a bit more difficult because of variable renaming (primed versions of τ and ϕ_{post}).

Synthesis

Safety conditions

To encode a formula for the validity of a Hoare triple, we define

$$PathC : \phi \times \textsc{Tsl} \times \phi \rightarrow \phi$$

which takes a precondition, a sequence of statements and a postcondition, and returns the safety condition:

$$PathC(\phi_{pre}, \textbf{choose } \{[]\, g_i \rightarrow s_i\}_i, \phi_{post}) =$$
$$\bigwedge_i (\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \phi'_{post})$$
$$PathC(\phi_{pre}, \textbf{while}^{\tau,\varphi} \ (g) \ \{\vec{p}_l\}, \phi_{post}) =$$
$$\phi_{pre} \Rightarrow \tau' \wedge PathC(\tau \wedge g, \vec{p}_l, \tau) \wedge (\tau \wedge \neg g \Rightarrow \phi'_{post})$$

Encoding sequences of statements a bit more difficult because of variable renaming (primed versions of τ and ϕ_{post}).

Synthesis

Safety conditions

To encode a formula for the validity of a Hoare triple, we define

$$PathC : \phi \times \textrm{TSL} \times \phi \to \phi$$

which takes a precondition, a sequence of statements and a postcondition, and returns the safety condition:

$$PathC(\phi_{pre}, \textbf{choose}\ \ \{[]\, g_i \to s_i\}_i\, , \phi_{post}) = \bigwedge_i (\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \phi'_{post})$$

$$PathC(\phi_{pre}, \textbf{while}^{\tau,\varphi}\ \ (g)\ \ \{\vec{p_l}\}\, , \phi_{post}) = \phi_{pre} \Rightarrow \tau' \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge \left(\tau \wedge \neg g \Rightarrow \phi'_{post}\right)$$

Encoding sequences of statements a bit more difficult because of variable renaming (primed versions of τ and ϕ_{post}).

Synthesis

Safety conditions

Note. Any 2 consecutive acyclic fragments with n_1 and n_2 transitions can be collapsed into one with $n_1 \cdot n_2$ transitions.

$$PathC(\phi_{pre}, \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}\ ;\ \vec{p}, \phi_{post}) = (\phi_{pre} \Rightarrow \tau') \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge PathC(\tau \wedge \neg g, \vec{p}, \phi_{post})$$

$$PathC(\phi_{pre}, \textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i\ ;\ \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}, \phi_{post}) = \bigwedge_i (\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \tau') \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge (\tau \wedge \neg g \Rightarrow \phi'_{post})$$

$$PathC(\phi_{pre}, \textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i\ ;\ \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}\ ;\ \vec{p}, \phi_{post}) = \bigwedge_i (\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \tau') \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge PathC(\tau \wedge \neg g, \vec{p}, \phi_{post})$$

$$SafetyCond(exp, \mathcal{F}) = PathC(F_{pre}, exp, F_{post})$$

Synthesis

Safety conditions

Note. Any 2 consecutive acyclic fragments with n_1 and n_2 transitions can be collapsed into one with $n_1 \cdot n_2$ transitions.

$$PathC(\phi_{pre}, \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}\ ;\ \vec{p}, \phi_{post}) =$$
$$\left(\phi_{pre} \Rightarrow \tau'\right) \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge PathC(\tau \wedge \neg g, \vec{p}, \phi_{post})$$

$$PathC(\phi_{pre}, \textbf{choose}\ \{[]\, g_i \to s_i\}_i\ ;\ \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}, \phi_{post}) =$$
$$\bigwedge_i \left(\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \tau'\right) \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge \left(\tau \wedge \neg g \Rightarrow \phi'_{post}\right)$$

$$PathC(\phi_{pre}, \textbf{choose}\ \{[]\, g_i \to s_i\}_i\ ;\ \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}\ ;\ \vec{p}, \phi_{post}) =$$
$$\bigwedge_i \left(\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \tau'\right) \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge$$
$$PathC(\tau \wedge \neg g, \vec{p}, \phi_{post})$$

$$SafetyCond(exp, \mathcal{F}) = PathC(F_{pre}, exp, F_{post})$$

Synthesis

Safety conditions

Note. Any 2 consecutive acyclic fragments with n_1 and n_2 transitions can be collapsed into one with $n_1 \cdot n_2$ transitions.

$$PathC(\phi_{pre}, \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}\ ;\ \vec{p}, \phi_{post}) =$$
$$\left(\phi_{pre} \Rightarrow \tau'\right) \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge PathC(\tau \wedge \neg g, \vec{p}, \phi_{post})$$

$$PathC(\phi_{pre}, \textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i\ ;\ \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}, \phi_{post}) =$$
$$\bigwedge_i \left(\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \tau'\right) \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge \left(\tau \wedge \neg g \Rightarrow \phi'_{post}\right)$$

$$PathC(\phi_{pre}, \textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i\ ;\ \textbf{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}\ ;\ \vec{p}, \phi_{post}) =$$
$$\bigwedge_i \left(\phi_{pre} \wedge g_i \wedge s_i \Rightarrow \tau'\right) \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge$$
$$PathC(\tau \wedge \neg g, \vec{p}, \phi_{post})$$

$$SafetyCond(exp, \mathcal{F}) = PathC(F_{pre}, exp, F_{post})$$

Synthesis

Safety conditions

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- $exp_{sqrt} =$

  choose $\{[]\, g_1 \rightarrow s_1\}$;
  while$^{\tau,\varphi}$ (g_0) $\{$ **choose** $\{[]\, g_2 \rightarrow s_2\}$; $\}$;
  choose $\{[]\, g_3 \rightarrow s_3\}$;

$$SafetyCond(exp_{sqrt}, \mathcal{F}) =$$
$$(x \geq 1 \wedge g_1 \wedge s_1 \Rightarrow \tau') \wedge$$
$$(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow \tau') \wedge$$
$$\left(\tau \wedge \neg g_0 \wedge g_3 \wedge s_3 \Rightarrow \left(i'-1\right)^2 \leq x' < i'^2\right)$$

where g_i, s_i and τ are all unknowns.

Synthesis

Safety conditions

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- $exp_{sqrt} =$

  choose $\{[]\, g_1 \rightarrow s_1\}$;
  while$^{\tau,\varphi}$ (g_0) { **choose** $\{[]\, g_2 \rightarrow s_2\}$; } ;
  choose $\{[]\, g_3 \rightarrow s_3\}$;

$$\begin{aligned} & SafetyCond(exp_{sqrt}, \mathcal{F}) = \\ & \quad (x \geq 1 \wedge g_1 \wedge s_1 \Rightarrow \tau') \wedge \\ & \quad (\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow \tau') \wedge \\ & \quad \left(\tau \wedge \neg g_0 \wedge g_3 \wedge s_3 \Rightarrow \left(i' - 1\right)^2 \leq x' < i'^2\right) \end{aligned}$$

where g_i, s_i and τ are all unknowns.

Synthesis

Well-formedness conditions

$$WellFormTS(\{[]\, g_i \rightarrow s_i\}_i) \doteq \left(\bigwedge_i valid(s_i)\right) \wedge \left(\bigvee_i g_i\right)$$

where

- $valid(s_i)$ ensures that each variable is assigned only once in s_i
- $(\bigvee_i g_i)$ guarantees all space is covered by the guards g_i
- guards do not have to be mutually exclusive

$$WellFormCond(exp) = \bigwedge_{(\textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i) \in cond(exp)} WellFormTS(\{[]\, g_i \rightarrow s_i\}_i)$$

where $cond(exp)$ is the set of all **choose** statements in the expanded scaffold exp.

This is called non-iterative upper bounded search. Iterative lower bounded search is also possible (remember parameter n at expansion).

Synthesis

Well-formedness conditions

$$WellFormTS(\{[]\, g_i \rightarrow s_i\}_i) \doteq \left(\bigwedge_i valid(s_i)\right) \wedge \left(\bigvee_i g_i\right)$$

where

- $valid(s_i)$ ensures that each variable is assigned only once in s_i
- $(\bigvee_i g_i)$ guarantees all space is covered by the guards g_i
- guards do not have to be mutually exclusive

$$WellFormCond(exp) = \bigwedge_{(\textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i) \in cond(exp)} WellFormTS(\{[]\, g_i \rightarrow s_i\}_i)$$

where $cond(exp)$ is the set of all **choose** statements in the expanded scaffold *exp*.

This is called non-iterative upper bounded search. Iterative lower bounded search is also possible (remember parameter n at expansion).

Synthesis

Well-formedness conditions

$$WellFormTS(\{[]\, g_i \rightarrow s_i\}_i) \doteq \left(\bigwedge_i valid(s_i)\right) \wedge \left(\bigvee_i g_i\right)$$

where

- $valid(s_i)$ ensures that each variable is assigned only once in s_i
- $(\bigvee_i g_i)$ guarantees all space is covered by the guards g_i
- guards do not have to be mutually exclusive

$$WellFormCond(exp) = \bigwedge_{(\textbf{choose}\ \{[]\, g_i \rightarrow s_i\}_i) \in cond(exp)} WellFormTS(\{[]\, g_i \rightarrow s_i\}_i)$$

where $cond(exp)$ is the set of all **choose** statements in the expanded scaffold *exp*.

This is called non-iterative upper bounded search. Iterative lower bounded search is also possible (remember parameter n at expansion).

Example

- $exp_{sqrt} =$

  $\textbf{choose}\ \{[]\, g_1 \rightarrow s_1\}\,;$
  $\textbf{while}^{\tau,\varphi}\ (g_0)\ \{\ \textbf{choose}\ \{[]\, g_2 \rightarrow s_2\}\,;\ \}\,;$
  $\textbf{choose}\ \{[]\, g_3 \rightarrow s_3\}\,;$

$$WellFormCond(exp_{sqrt}) = valid(s_1) \wedge valid(s_2) \wedge valid(s_3) \wedge g_1 \wedge g_2 \wedge g_3$$

Example

- $exp_{sqrt} =$

  $$\textbf{choose}\ \{[]\, g_1 \rightarrow s_1\}\,;$$
  $$\textbf{while}^{\tau,\varphi}\ (g_0)\ \{\ \textbf{choose}\ \{[]\, g_2 \rightarrow s_2\}\,;\ \}\,;$$
  $$\textbf{choose}\ \{[]\, g_3 \rightarrow s_3\}\,;$$

$$WellFormCond(exp_{sqrt}) = valid(s_1) \wedge valid(s_2) \wedge valid(s_3) \wedge g_1 \wedge g_2 \wedge g_3$$

Synthesis

Progress conditions

$$prog(\textbf{while}^{\tau,\varphi}(g)\ \ \{\vec{p}\}) \doteq (r = \varphi) \wedge (\tau \Rightarrow r \geq 0) \wedge$$
$$PathC(\tau_{end} \wedge g, end(\vec{p}), r > \varphi)$$

where

- r is a new progress tracking variable (not an unknown)
- τ_{end} is the invariant for the last loop in $\vec{p}$
  - Meaning, that we require intermediate loop invariants to carry enough information
- $end(\vec{p})$ is the fragment of $\vec{p}$ after the last loop

$$RankCond(exp) = \bigwedge_{(\textbf{while}^{\tau,\varphi}(g)\ \ \{\vec{p}\}) \in loops(exp)} prog(\textbf{while}^{\tau,\varphi}(g)\ \ \{\vec{p}\})$$

where $loops(exp)$ is the set of all **while** statements in the expanded scaffold exp.

Synthesis

Progress conditions

$$prog(\mathbf{while}^{\tau,\varphi}(g)\ \{\vec{p}\}) \doteq (r = \varphi) \wedge (\tau \Rightarrow r \geq 0) \wedge$$
$$PathC(\tau_{end} \wedge g, end(\vec{p}), r > \varphi)$$

where

- r is a new progress tracking variable (not an unknown)
- τ_{end} is the invariant for the last loop in $\vec{p}$
  - Meaning, that we require intermediate loop invariants to carry enough information
- $end(\vec{p})$ is the fragment of $\vec{p}$ after the last loop

$$RankCond(exp) = \bigwedge_{(\mathbf{while}^{\tau,\varphi}(g)\ \{\vec{p}\}) \in loops(exp)} prog(\mathbf{while}^{\tau,\varphi}(g)\ \{\vec{p}\})$$

where $loops(exp)$ is the set of all **while** statements in the expanded scaffold exp.

Example

- $exp_{sqrt} =$

  $\textbf{choose} \ \{[]\, g_1 \rightarrow s_1\}\,;$

  $\textbf{while}^{\tau,\varphi} \ (g_0) \ \{ \ \textbf{choose} \ \{[]\, g_2 \rightarrow s_2\}\,; \ \}\,;$

  $\textbf{choose} \ \{[]\, g_3 \rightarrow s_3\}\,;$

$$RankCond(exp_{sqrt}) = \ (r = \varphi) \wedge (\tau \Rightarrow r \geq 0) \wedge$$
$$\left(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow r' > \varphi'\right)$$

Example

- $exp_{sqrt} =$

  $\textbf{choose}\ \{[]\, g_1 \rightarrow s_1\}\,;$

  $\textbf{while}^{\tau,\varphi}\ (g_0)\ \{\ \textbf{choose}\ \{[]\, g_2 \rightarrow s_2\}\,;\ \}\,;$

  $\textbf{choose}\ \{[]\, g_3 \rightarrow s_3\}\,;$

$$RankCond(exp_{sqrt}) = (r = \varphi) \wedge (\tau \Rightarrow r \geq 0) \wedge \big(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow r' > \varphi'\big)$$

Synthesis

Entire synthesis algorithm

- **Input:**
  - Scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$,
  - Maximum number of transitions n
  - Proof domain D_{prf}
- **Output:** Executable program or FAIL

```
exp := Expand^{n,D_prf}_{D,R}(R_flow);
sc  := SafetyCond(exp, F) ∧
       WellFormCond(exp) ∧
       RankCond(exp);
π   := Solver(sc);

if (unsat(π))
|    return FAIL;

return Exe^π(exp);
```

Synthesis

Concretization algorithm

$$
\begin{aligned}
Exe^{\pi}(p;\vec{p}) &= Exe^{\pi}(p)\ ;\ Exe^{\pi}(\vec{p})\\
Exe^{\pi}(\mathbf{while}^{\tau,\varphi}\,(g)\ \{\vec{p}\}) &= \mathbf{while}^{\pi(\tau),\pi(\varphi)}\,(\pi\,(g))\ \{Exe^{\pi}(\vec{p})\}\\
Exe^{\pi}(\mathbf{choose}\,\{[]\,g \rightarrow s\}) &= \mathbf{if}\ \ (\pi\,(g)\,)\ \ \{Stmt(\pi\,(s))\,\}\\
&\quad\ \mathbf{else}\ \ \{\mathbf{skip}\}\\
Exe^{\pi}(\mathbf{choose}\,\{[]\,g_i \rightarrow s_i\}_{i=1\dots n}) &= \mathbf{if}\ \ (\pi\,(g)\,)\ \ \{Stmt(\pi\,(s))\,\}\\
&\quad\ \mathbf{else}\ \ \{Exe^{\pi}(\mathbf{choose}\,\{[]\,g_i \rightarrow s_i\}_{i=2\dots n})\,\}
\end{aligned}
$$

$$
\begin{aligned}
Stmt\Big(\bigwedge\nolimits_{i=1\dots n} x_i = e_i\Big) &= t_1 := e_1\,;\ \dots\,;t_n := e_n\,;\\
&\quad\ x_1 := t_1\,;\ \dots\,;x_n := t_n\,;
\end{aligned}
$$

Synthesis

Concretization algorithm

$$
\begin{aligned}
Exe^{\pi}(p;\vec{p}) &= Exe^{\pi}(p)\ ;\ Exe^{\pi}(\vec{p})\\
Exe^{\pi}(\mathbf{while}^{\tau,\varphi}\,(g)\ \{\vec{p}\}) &= \mathbf{while}^{\pi(\tau),\pi(\varphi)}\,(\pi\,(g))\ \{Exe^{\pi}(\vec{p})\}\\
Exe^{\pi}(\mathbf{choose}\,\{[]\,g \to s\}) &= \mathbf{if}\ \ (\pi\,(g)\,)\ \ \{Stmt(\pi\,(s))\,\}\\
&\quad\ \mathbf{else}\ \ \{\mathbf{skip}\}\\
Exe^{\pi}(\mathbf{choose}\,\{[]\,g_i \to s_i\}_{i=1\ldots n}) &= \mathbf{if}\ \ (\pi\,(g)\,)\ \ \{Stmt(\pi\,(s))\,\}\\
&\quad\ \mathbf{else}\ \ \{Exe^{\pi}(\mathbf{choose}\,\{[]\,g_i \to s_i\}_{i=2\ldots n})\,\}
\end{aligned}
$$

$$
\begin{aligned}
Stmt\Big(\bigwedge\nolimits_{i=1\ldots n} x_i = e_i\Big) &= t_1 := e_1\,;\ \ldots\,; t_n := e_n\,;\\
&\quad\ x_1 := t_1\,;\ \ldots\,; x_n := t_n\,;
\end{aligned}
$$

Synthesis

Example

$$\left(x \geq 1 \wedge g_1 \wedge s_1 \Rightarrow \tau'\right) \wedge \left(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow \tau'\right) \wedge \left(\tau \wedge \neg g_0 \wedge g_3 \wedge s_3 \Rightarrow \left(i' - 1\right)^2 \leq x' < i'^2\right) \wedge$$

$$valid(s_1) \wedge valid(s_2) \wedge valid(s_3) \quad \wedge \quad (r = \varphi) \wedge (\tau \Rightarrow r \geq 0) \wedge \left(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow r' > \varphi'\right)$$

$$\tau : \ (v = i^2) \wedge (x \geq (i-1)^2) \wedge (i \geq 1)$$

$$g_0 : v \leq x$$

$$\varphi : x - (i-1)^2$$

$$s_1 : \ (v' = 1) \wedge (i' = 1) \wedge (x' = x) \wedge (r' = r)$$

$$s_2 : \ (v' = v + 2i + 1) \wedge (i' = i + 1) \wedge (x' = x) \wedge (r' = r)$$

$$s_3 : \ (v' = v) \wedge (i' = i) \wedge (x' = x) \wedge (r' = r)$$

Synthesis

Example

$$\left[\left(x \geq 1 \wedge g_1 \wedge s_1 \Rightarrow \tau'\right) \wedge \left(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow \tau'\right) \wedge \left(\tau \wedge \neg g_0 \wedge g_3 \wedge s_3 \Rightarrow \left(i' - 1\right)^2 \leq x' < i'^2\right)\right] \wedge$$
$$\left[valid(s_1) \wedge valid(s_2) \wedge valid(s_3)\right] \wedge \left[(r = \varphi) \wedge (\tau \Rightarrow r \geq 0) \wedge \left(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow r' > \varphi'\right)\right]$$

$\tau : \ (v = i^2) \wedge (x \geq (i-1)^2) \wedge (i \geq 1)$

$g_0 : v \leq x$

$\varphi : x - (i-1)^2$

$s_1 : \ (v' = 1) \wedge (i' = 1) \wedge (x' = x) \wedge (r' = r)$

$s_2 : \ (v' = v + 2i + 1) \wedge (i' = i + 1) \wedge (x' = x) \wedge (r' = r)$

$s_3 : \ (v' = v) \wedge (i' = i) \wedge (x' = x) \wedge (r' = r)$

Synthesis

Example

$$\Big[\left(x \geq 1 \wedge g_1 \wedge s_1 \Rightarrow \tau'\right) \wedge \left(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow \tau'\right) \wedge \left(\tau \wedge \neg g_0 \wedge g_3 \wedge s_3 \Rightarrow \left(i' - 1\right)^2 \leq x' < i'^2\right)\Big] \wedge$$
$$\Big[valid(s_1) \wedge valid(s_2) \wedge valid(s_3)\Big] \wedge \Big[(r = \varphi) \wedge (\tau \Rightarrow r \geq 0) \wedge \left(\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow r' > \varphi'\right)\Big]$$

$$\tau : \left(v = i^2\right) \wedge \left(x \geq (i-1)^2\right) \wedge (i \geq 1)$$
$$g_0 : v \leq x$$
$$\varphi : x - (i-1)^2$$
$$s_1 : \left(v' = 1\right) \wedge \left(i' = 1\right) \wedge \left(x' = x\right) \wedge \left(r' = r\right)$$
$$s_2 : \left(v' = v + 2i + 1\right) \wedge \left(i' = i + 1\right) \wedge \left(x' = x\right) \wedge \left(r' = r\right)$$
$$s_3 : \left(v' = v\right) \wedge \left(i' = i\right) \wedge \left(x' = x\right) \wedge \left(r' = r\right)$$

Synthesis

Requirements for solvers

- Support for multiple positive and negative unknowns
  - $(\tau \wedge g \Rightarrow \tau') \wedge (\tau \wedge \neg g \Rightarrow \phi_{post})$
- Solutions are maximally weak,
  - ensuring that the non-standard conditions $valid(s_i)$ will hold.

Synthesis

Requirements for solvers

- Support for multiple positive and negative unknowns
  - $(\tau \wedge g \Rightarrow \tau') \wedge (\tau \wedge \neg g \Rightarrow \phi_{post})$
- Solutions are maximally weak,
  - ensuring that the non-standard conditions $valid(s_i)$ will hold.

The VS^3 project

- Arithmetic verification tool VS^3_{LIA}
  - works over the theory of linear arithmetic
  - discovers (quantifier-free) invariants in DNF form with linear inequalities over program variables as the atomic facts
  - supports limits on data size in bits and a limit on the number of conjunctions/disjunctions
- $VS^3_{QA} = VS^3_{LIA} +$ quadratic expressions (incomplete)
- Predicate abstraction verification tool VS^3_{PA}
  - works over a combination of the theories of equality with uninterpreted functions, arrays, and linear arithmetic
  - discovers (possibly) quantified invariants
  - requires a boolean template for the invariant and a set of predicates to put into template holes
    - e.g. $[-] \wedge \forall k : [-] \Rightarrow [-]$
- $VS^3_{AX} = VS^3_{PA} +$ user-specified axioms over uninterpreted symbols

The VS^3 project

- Arithmetic verification tool VS^3_{LIA}
  - works over the theory of linear arithmetic
  - discovers (quantifier-free) invariants in DNF form with linear inequalities over program variables as the atomic facts
  - supports limits on data size in bits and a limit on the number of conjunctions/disjunctions
- $VS^3_{QA} = VS^3_{LIA} +$ quadratic expressions (incomplete)
- Predicate abstraction verification tool VS^3_{PA}
  - works over a combination of the theories of equality with uninterpreted functions, arrays, and linear arithmetic
  - discovers (possibly) quantified invariants
  - requires a boolean template for the invariant and a set of predicates to put into template holes
    - e.g. $[-] \wedge \forall k : [-] \Rightarrow [-]$
- $VS^3_{AX} = VS^3_{PA} +$ user-specified axioms over uninterpreted symbols

The VS^3 project

- Arithmetic verification tool VS^3_{LIA}
  - works over the theory of linear arithmetic
  - discovers (quantifier-free) invariants in DNF form with linear inequalities over program variables as the atomic facts
  - supports limits on data size in bits and a limit on the number of conjunctions/disjunctions
- $VS^3_{QA} = VS^3_{LIA} +$ quadratic expressions (incomplete)
- Predicate abstraction verification tool VS^3_{PA}
  - works over a combination of the theories of equality with uninterpreted functions, arrays, and linear arithmetic
  - discovers (possibly) quantified invariants
  - requires a boolean template for the invariant and a set of predicates to put into template holes
    - e.g. $[-] \wedge \forall k : [-] \Rightarrow [-]$
- $VS^3_{AX} = VS^3_{PA} +$ user-specified axioms over uninterpreted symbols

The VS^3 project

- Arithmetic verification tool VS^3_{LIA}
  - works over the theory of linear arithmetic
  - discovers (quantifier-free) invariants in DNF form with linear inequalities over program variables as the atomic facts
  - supports limits on data size in bits and a limit on the number of conjunctions/disjunctions
- $\mathrm{VS}^3_{QA} = \mathrm{VS}^3_{LIA} +$ quadratic expressions (incomplete)
- Predicate abstraction verification tool VS^3_{PA}
  - works over a combination of the theories of equality with uninterpreted functions, arrays, and linear arithmetic
  - discovers (possibly) quantified invariants
  - requires a boolean template for the invariant and a set of predicates to put into template holes
    - e.g. $[-] \wedge \forall k : [-] \Rightarrow [-]$
- $\mathrm{VS}^3_{AX} = \mathrm{VS}^3_{PA} +$ user-specified axioms over uninterpreted symbols

Experimental case studies

Flowgraphs with initialization and finalization

We instead treat loops ($*(T)$) in *Expand* as $\circ; *(T); \circ$ to make things easier for the verification tools.

Experimental case studies

Swapping of values

Example

- $F_{pre} \doteq (x = c_1) \wedge (y = c_2)$
- $F_{post} \doteq (x = c_2) \wedge (y = c_1)$
- $R_{flow} \doteq \circ$
- $R_{comp} \doteq \emptyset$
- $R_{stack} \doteq \emptyset$

Synthesizer generates various versions, including

```
Swap(int x, int y)
| x := x + y;
| y := x - y;
| x := x - y;
```

Experimental case studies

Swapping of values

Example

- $F_{pre} \doteq (x = c_1) \land (y = c_2)$
- $F_{post} \doteq (x = c_2) \land (y = c_1)$
- $R_{flow} \doteq \circ$
- $R_{comp} \doteq \emptyset$
- $R_{stack} \doteq \emptyset$

Synthesizer generates various versions, including

```
Swap(int x, int y)
| x := x + y;
| y := x − y;
| x := x − y;
```

Experimental case studies

Integral square root

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- $R_{flow} \doteq *(\circ)$ and $R_{comp} \doteq \emptyset$

- $R_{stack} \doteq \{(\mathbf{int}, 1)\}$ + quadratic expressions in D_{exp}, D_{grd} = sequential search
- $R_{stack} \doteq \{(\mathbf{int}, 2)\}$ + linear expressions in D_{exp}, D_{grd} = sequential search

```
v := 1; i := 1;
while^{T,φ} (v ≤ x)
|    v := v + 2i + 1; i++;
return i − 1;
```

- $R_{stack} \doteq \{(\mathbf{int}, 3)\}$ + quadratic + extra assumptions
  - binary search (temporaries hold search range)

Experimental case studies

Integral square root

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- $R_{flow} \doteq *(\circ)$ and $R_{comp} \doteq \emptyset$
- $R_{stack} \doteq \{(\mathbf{int}, 1)\}$ + quadratic expressions in D_{exp}, D_{grd} = sequential search
- $R_{stack} \doteq \{(\mathbf{int}, 2)\}$ + linear expressions in D_{exp}, D_{grd} = sequential search

```
v := 1; i := 1;
while^{τ,φ} (v ≤ x)
|     v := v + 2i + 1;  i++;
return i − 1;
```

- $R_{stack} \doteq \{(\mathbf{int}, 3)\}$ + quadratic + extra assumptions
  - binary search (temporaries hold search range)

Experimental case studies

Integral square root

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- $R_{flow} \doteq *(\circ)$ and $R_{comp} \doteq \emptyset$

- $R_{stack} \doteq \{(\mathbf{int}, 1)\}$ + quadratic expressions in D_{exp}, D_{grd} = sequential search
- $R_{stack} \doteq \{(\mathbf{int}, 2)\}$ + linear expressions in D_{exp}, D_{grd} = sequential search

```
v := 1; i := 1;
while^{τ,φ} (v ≤ x)
|     v := v + 2i + 1;  i++;
return i − 1;
```

- $R_{stack} \doteq \{(\mathbf{int}, 3)\}$ + quadratic + extra assumptions
  - binary search (temporaries hold search range)

Experimental case studies

Integral square root

Example

- $\mathcal{F} = \left(x \geq 1, (i-1)^2 \leq x < i^2\right)$
- $R_{flow} \doteq *(\circ)$ and $R_{comp} \doteq \emptyset$

- $R_{stack} \doteq \{(\mathbf{int}, 1)\}$ + quadratic expressions in D_{exp}, D_{grd} = sequential search
- $R_{stack} \doteq \{(\mathbf{int}, 2)\}$ + linear expressions in D_{exp}, D_{grd} = sequential search

```
v := 1; i := 1;
while^{τ,φ} (v ≤ x)
|     v := v + 2i + 1;  i++;
return i − 1;
```

- $R_{stack} \doteq \{(\mathbf{int}, 3)\}$ + quadratic + extra assumptions
  - binary search (temporaries hold search range)

Experimental case studies

Non-recursive sorting

Example

- $\mathcal{F} = (\text{true}, \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1])$
- D_{exp} includes swapping of array elements, R_{comp} allows swapping only, $R_{flow} \doteq *(*(\circ))$
- $R_{stack} \doteq \emptyset$: Bubble Sort and a non-standard version of Insertion Sort.
- $R_{stack} \doteq \{(\textbf{int}, 1)\}$: Selection Sort.

Experimental case studies

Non-recursive sorting

Example

- $\mathcal{F} = (\text{true}, \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1])$
- D_{exp} includes swapping of array elements, R_{comp} allows swapping only, $R_{flow} \doteq *(*(\circ))$
- $R_{stack} \doteq \emptyset$: Bubble Sort and a non-standard version of Insertion Sort.
- $R_{stack} \doteq \{(\textbf{int}, 1)\}$: Selection Sort.

Experimental case studies

Non-recursive sorting

Example

- $\mathcal{F} = (\text{true}, \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1])$
- D_{exp} includes swapping of array elements, R_{comp} allows swapping only, $R_{flow} \doteq *(*(\circ))$
- $R_{stack} \doteq \emptyset$: Bubble Sort and a non-standard version of Insertion Sort.
- $R_{stack} \doteq \{(\mathbf{int}, 1)\}$: Selection Sort.

Experimental case studies

Recursive divide-and-conquer sorting

Example

- $\mathcal{F} = (\mathsf{true}, \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1])$
- D_{exp} includes swapping and moving of array elements
- Flowgraph template includes recursive call $\circledast$
- $R_{stack} \doteq \emptyset$, $R_{flow} \doteq \circledast; \circledast; \circ$: Merge Sort.
- $R_{stack} \doteq \{(\mathbf{int}, 1)\}$, $R_{flow} \doteq \circ; \circledast; \circledast$: Quick Sort.

Experimental case studies

Recursive divide-and-conquer sorting

Example

- $\mathcal{F} = (\text{true}, \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1])$
- D_{exp} includes swapping and moving of array elements
- Flowgraph template includes recursive call $\circledast$
- $R_{stack} \doteq \emptyset$, $R_{flow} \doteq \circledast; \circledast; \circ$: Merge Sort.
- $R_{stack} \doteq \{(\textbf{int}, 1)\}$, $R_{flow} \doteq \circ; \circledast; \circledast$: Quick Sort.

Experimental case studies

Recursive divide-and-conquer sorting

Example

- $\mathcal{F} = (\text{true}, \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1])$
- D_{exp} includes swapping and moving of array elements
- Flowgraph template includes recursive call $\circledast$

- $R_{stack} \doteq \emptyset$, $R_{flow} \doteq \circledast; \circledast; \circ$: Merge Sort.
- $R_{stack} \doteq \{(\textbf{int}, 1)\}$, $R_{flow} \doteq \circ; \circledast; \circledast$: Quick Sort.

Experimental case studies

Dynamic programming

Example

- Fibonacci
- Longest Common Subsequence
- Path-finding
  - Checkerboard (least-cost path on rectangular grid)
  - Single Source Shortest Path
  - All-pairs Shortest Path
- Matrix Chain Multiply (minimizing the number of multiplications)

Experimental case studies

Benchmarks

- Synthesis time 0.12-9658.52 seconds (median 14.23)
- Slowdown in respect to verification 1.09-92.28 (median 6.68)

Limitations not easily overcome

- Need to add new assumptions to compensate for incomplete VS^3_{QA} (quadratic expression handling) and inefficient VS^3_{AX}.
- Need a set of candidate predicates for VS^3_{AX}

Scalability

- More efficient verifiers are needed.

Relevance

- Multiple solutions differ in performance, readability.

Experimental case studies

Benchmarks

- Synthesis time 0.12-9658.52 seconds (median 14.23)
- Slowdown in respect to verification 1.09-92.28 (median 6.68)

Limitations not easily overcome

- Need to add new assumptions to compensate for incomplete VS^3_{QA} (quadratic expression handling) and inefficient VS^3_{AX}.
- Need a set of candidate predicates for VS^3_{AX}

Scalability

- More efficient verifiers are needed.

Relevance

- Multiple solutions differ in performance, readability.

Experimental case studies

Benchmarks

- Synthesis time 0.12-9658.52 seconds (median 14.23)
- Slowdown in respect to verification 1.09-92.28 (median 6.68)

Limitations not easily overcome

- Need to add new assumptions to compensate for incomplete VS^3_{QA} (quadratic expression handling) and inefficient VS^3_{AX}.
- Need a set of candidate predicates for VS^3_{AX}

Scalability

- More efficient verifiers are needed.

Relevance

- Multiple solutions differ in performance, readability.

Experimental case studies

Benchmarks

- Synthesis time 0.12-9658.52 seconds (median 14.23)
- Slowdown in respect to verification 1.09-92.28 (median 6.68)

Limitations not easily overcome

- Need to add new assumptions to compensate for incomplete VS^3_{QA} (quadratic expression handling) and inefficient VS^3_{AX}.
- Need a set of candidate predicates for VS^3_{AX}

Scalability

- More efficient verifiers are needed.

Relevance

- Multiple solutions differ in performance, readability.

E O F