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
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 → 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 → 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 \(\rightarrow\) 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
Motivating example
Bresenham’s line drawing algorithm

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

\[ \tau_{pre} : \ 0 < Y \leq X \]

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

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

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

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

\textbf{Bresenham's (} \texttt{int X, int Y)}

\begin{itemize}
  \item \texttt{v} \_1 := 2Y - X; \ y := 0; \ x := 0;
  \item \textbf{while (} \ x <= X) \textbf{)}
     \begin{itemize}
       \item \texttt{out[x]} := \ y;
       \item \textbf{if (} \ \texttt{v} \_1 < 0) \textbf{)}
          \begin{itemize}
            \item \texttt{v} \_1 := \texttt{v} \_1 + 2Y;
          \end{itemize}
       \item \textbf{else}
          \begin{itemize}
            \item \texttt{v} \_1 := \texttt{v} \_1 + 2(Y - X); \ y++;
          \end{itemize}
       \item x++;
     \end{itemize}
  \item \textbf{return out};
\end{itemize}
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 \( y' = x \land x' = y' \)
▶ \( \textbf{if } (x > 0) \ x := y; \textbf{else skip} \); becomes

\[
\begin{align*}
[] x > 0 & \rightarrow x' = y \\
[] x \leq 0 & \rightarrow \text{true}
\end{align*}
\]
Motivating example
Bresenham's line drawing algorithm

Example

\[\text{true} \rightarrow v'_1 = 2Y - X \land y' = 0 \land x' = 0\]

\textbf{while} \ (x \leq X)

\[\begin{align*}
\text{[]} v_1 < 0 & \rightarrow \quad \text{out}' = \text{upd}(\text{out}, x, y) \land \\
& \quad v'_1 = v_1 + 2Y \land \\
& \quad y' = y \land \\
& \quad x' = x + 1
\end{align*}\]

\[\begin{align*}
\text{[]} v_1 \geq 0 & \rightarrow \quad \text{out}' = \text{upd}(\text{out}, x, y) \land \\
& \quad v'_1 = v_1 + 2(Y - X) \land \\
& \quad y' = y + 1 \land \\
& \quad x' = x + 1
\end{align*}\]
Motivating example

Bresenham's line drawing algorithm

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

\[ \tau : 0 < Y \leq X \land \]
\[ \nu_1 = 2(x + 1)Y - (2y + 1)X \land \]
\[ 2(Y - X) \leq \nu_1 \leq 2Y \land \]
\[ \forall k : 0 \leq k < x \Rightarrow 2|out[k] - (Y/X)k| \leq 1 \]

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

\[ \tau_{pre} \land s_{entry} \Rightarrow \tau' \]
\[ \tau \land \neg g_{loop} \Rightarrow \tau_{post} \]
\[ \tau \land g_{loop} \land g_{body1} \land s_{body1} \Rightarrow \tau' \]
\[ \tau \land g_{loop} \land g_{body2} \land s_{body2} \Rightarrow \tau' \]

where \( \tau' \) is the renamed version of the loop invariant.
Motivating example

Bresenham’s line drawing algorithm

\[ s_{entry} : v_1' = 2Y - X \land y' = 0 \land x' = 0 \]

\[ g_{loop} : x \leq X \]

\[ g_{body1} : v_1 < 0 \]

\[ s_{body1} : out' = \text{upd}(out, x, y) \land \]
\[ v_1' = v_1 + 2Y \land \]
\[ y' = y \land \]
\[ x' = x + 1 \]

\[ g_{body2} : v_1 \geq 0 \]

\[ s_{body2} : out' = \text{upd}(out, x, y) \land \]
\[ v_1' = v_1 + 2(Y - X) \land \]
\[ y' = y + 1 \land \]
\[ x' = x + 1 \]
Motivating example

One can validate that the loop invariant $\tau$ 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 $\tau$

▶ inferring ranking functions ($\varphi$) 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 $\tau$ 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 $\tau$
- inferring ranking functions ($\varphi$) 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

\[ \langle F, D, R \rangle \]

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

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

$$\mathcal{F} = (F_{pre}(\vec{v}_{in}), F_{post}(\vec{v}_{out}))$$

where $F_{pre}(\vec{v}_{in})$ and $F_{post}(\vec{v}_{out})$ are formulas that hold at the program entry and exit locations, respectively.
Domain constraints $\mathcal{D}$

\[ \mathcal{D} = (D_{\text{exp}}, D_{\text{grd}}) \]

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

Proof domain $D_{\text{prf}}$

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

Synthesis scaffold

Domain constraints $\mathcal{D}$

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

where $D_{\text{exp}}$ is the domain of expressions in the program and $D_{\text{grd}}$ 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 $D_{\text{exp}}$ and $D_{\text{grd}}$.
- We need a solver capable of handling $D_{prf}$. 
Resource constraints $\mathcal{R}$

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

- $R_{flow}$ is a flowgraph template from the grammar
  $$T ::= \diamond \mid \ast(T) \mid T; T$$

- $R_{stack} : type \rightarrow \mathbb{N}_1$ 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

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

▶ \( D_{\text{exp}} \) limited to linear arithmetic (LA) expressions (no \( \sqrt{\cdot} \))

▶ \( D_{\text{grd}} \) limited to quantifier-free first-order logic (FOL) over LA

▶ \( R_{\text{flow}} = (\circ ; \ast (\circ) ; \circ) \), \( R_{\text{stack}} = \{ (\text{int}, 1) \}, R_{\text{comp}} = \emptyset \)

\[
\text{IntSqrt} \ (\text{int} \ x) \\
v := 1; \ i := 1; \\
\text{while}^{\tau,\varphi} (v \leq x) \\
\mid v := v + 2i + 1; \ i++; \\
\text{return} \ i - 1;
\]

▶ Invariant \( \tau : v = i^2 \land x \geq (i - 1)^2 \land i \geq 1 \)

▶ Ranking function \( \varphi : x - (i - 1)^2 \)
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 ($x'$).

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

Corresponding verification condition:

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

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

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

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

\[ \phi_{pre} \land x' = e_x \land y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post} \]
Synthesis conditions

Transition systems for acyclic code

Transitions

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

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

Corresponding verification condition:

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

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

Example

For \( x := e_x; y := e_y \) we will have

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

\[
\phi_{pre} \land x' = e_x \land y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post}
\]
Synthesis conditions
Transition systems for acyclic code

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

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

Corresponding verification condition:

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

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

Example
For \( x := e_x; y := e_y \) we will have

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

\[
\phi_{pre} \land x' = e_x \land y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post}
\]
Synthesis conditions
Transition systems for acyclic code

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

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

Corresponding verification condition:

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

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

Example
For \(x := e_x; y := e_y\) we will have

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

\[
\phi_{pre} \land x' = e_x \land y' = e_y[x \mapsto e_x] \Rightarrow \phi'_{post}
\]
Guarded transitions

Let's 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:

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

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 \([\square g \to s]\) 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:

\[
\{\phi_{\text{pre}}\} \{\square g_i \to s_i\}_i \{\phi'_{\text{post}}\}
\]

The corresponding verification for is:

\[
\bigwedge_i (\phi_{\text{pre}} \land g_i \land s_i \Rightarrow \phi'_{\text{post}})
\]

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 $\Gamma g \rightarrow s$ 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:

$$\{\phi_{pre}\} \{\Gamma g_i \rightarrow s_i\}_i \{\phi'_{post}\}$$

The corresponding verification for is:

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

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 \( \mathcal{g} \rightarrow g \) 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:

\[
\{ \phi_{pre} \} \{ \mathcal{g}_i \rightarrow s_i \} \{ \phi_{post}' \}
\]

The corresponding verification for is:

\[
\bigwedge_i (\phi_{pre} \land g_i \land s_i \Rightarrow \phi_{post}')
\]

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

- 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.
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.
Transition system language ($\text{TSL}$)

\[
p ::= \text{choose} \left\{ \left[ \right] g_i \rightarrow s_i \right\}_i \\
| \text{while}^{\tau,\varphi} (g) \{ p \} \\
| p; p
\]
Synthesis

Expanding the flowgraph

Expand function

\[
\text{Expand}_D^n, D_{\text{prf}}^n (\circ) = \text{choose} \ \{ \emptyset \ g_i \rightarrow s_i \}_{i=1 \ldots n}
\]

\[
\text{Expand}_D^n, D_{\text{prf}}^n (\ast (T)) = \text{while}_{\tau, \varphi} (g) \ \{ \text{Expand}_D^n, D_{\text{prf}}^n (T) \}
\]

\[
\text{Expand}_D^n, D_{\text{prf}}^n (T_1; T_2) = \text{Expand}_D^n, D_{\text{prf}}^n (T_1) ; \text{Expand}_D^n, D_{\text{prf}}^n (T_2)
\]

where all \( g_i, s_i, g, \tau \) and \( \varphi \) are new generated unknowns.

\[
s \in \bigwedge_i x_i = e_i
\]

where \( x_i \in V, e_i \in D_{\text{exp}}|_V \)

\[
\tau \in D_{\text{prf}}|_V
\]

\[
g \in D_{\text{grd}}|_V
\]

and \( V = \vec{v}_{\text{in}} \cup \vec{v}_{\text{out}} \cup T \cup L \) where

- \( T \) is subject to \( R_{\text{stack}} \)
- \( e_i \) is subject to \( R_{\text{comp}} \)
- \( L \) is the set of iteration counters and ranking function tracker variables
Expanding the flowgraph

Expand function

\[ \text{Expand}^{n, D_{prf}} (\circ) = \text{choose } \{ [] g_i \rightarrow s_i \}_{i=1 \ldots n} \]

\[ \text{Expand}^{n, D_{prf}} (\ast (T)) = \text{while}^{\tau, \varphi} (g) \left\{ \text{Expand}^{n, D_{prf}} (T) \right\} \]

\[ \text{Expand}^{n, D_{prf}} (T_1; T_2) = \text{Expand}^{n, D_{prf}} (T_1) ; \text{Expand}^{n, D_{prf}} (T_2) \]

where all \( g_i, s_i, g, \tau \) and \( \varphi \) are new generated unknowns and

\[ s \in \bigwedge_i x_i = e_i \quad \text{where } x_i \in V, \; e_i \in D_{exp} \mid V \]

\[ \tau \in D_{prf} \mid V \quad \text{and } g \in D_{grd} \mid V \]

and \( V = v_{in} \cup v_{out} \cup T \cup L \) 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

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

For \( n = 1 \) and FOL over quadratic expressions as \( D_{\text{prf}} \) we get:

\[
\text{exp}_{\text{sqrt}} = \text{Expand}_{D, R}^{n, \text{D}_{\text{prf}}} (R_{\text{flow}}) = \\
\text{choose} \ \{ [] g_1 \rightarrow s_1 \} ; \\
\text{while}_{\tau, \varphi} (g_0) \ { \text{choose} \ \{ [] g_2 \rightarrow s_2 \} ; \ } ; \\
\text{choose} \ \{ [] g_3 \rightarrow s_3 \} ; \\
\text{where} \ \vec{v}_{\text{in}} = \vec{v}_{\text{out}} = \{x\}, \ T = \{v\}, \ L = \{i, r\}.
\]
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 \( \sqrt{\cdot} \))
- \( D_{grd} \) limited to quantifier-free first-order logic (FOL) over LA
- \( R_{flow} = (\circ; \ast(\circ); \circ), \) \( R_{stack} = \{(\text{int},1)\}, \) \( R_{comp} = \emptyset \)

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

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

where \( \vec{v}_{in} = \vec{v}_{out} = \{ x \} \), \( T = \{ v \} \), \( L = \{ i, r \} \).
To encode a formula for the validity of a Hoare triple, we define

$$PathC : \phi \times TSL \times \phi \rightarrow \phi$$

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

$$PathC(\phi_{pre}, choose \{ [] g_i \rightarrow s_i \}_{i}, \phi_{post}) =$$

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

$$PathC(\phi_{pre}, while^{\tau,\varphi} (g) \{ \vec{p}_{i} \}, \phi_{post}) =$$

$$\phi_{pre} \Rightarrow \tau' \land PathC(\tau \land g, \vec{p}_{l}, \tau) \land (\tau \land \neg g \Rightarrow \phi'_{post})$$

Encoding sequences of statements a bit more difficult because of variable renaming (primed versions of $\tau$ and $\phi_{post}$).
To encode a formula for the validity of a Hoare triple, we define

$$PathC : \phi \times TSL \times \phi \rightarrow \phi$$

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

$$PathC(\phi_{pre}, \textbf{choose} \ \{[] \ g_i \rightarrow s_i\}; , \phi_{post}) =$$

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

$$PathC(\phi_{pre}, \textbf{while}^{\tau, \varphi} (g) \ \{\vec{p_l}\} , \phi_{post}) =$$

$$\phi_{pre} \Rightarrow \tau' \wedge PathC(\tau \wedge g, \vec{p_l}, \tau) \wedge (\tau \wedge \neg g \Rightarrow \phi'_{post})$$

Encoding sequences of statements a bit more difficult because of variable renaming (primed versions of $\tau$ and $\phi_{post}$).
To encode a formula for the validity of a Hoare triple, we define

$$PathC : \phi \times TSL \times \phi \rightarrow \phi$$

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

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

$$PathC(\phi_{pre}, \textbf{while}^{\tau,\varphi} (g) \{ \vec{p}_l \}, \phi_{post}) = \phi_{pre} \Rightarrow \tau' \land PathC(\tau \land g, \vec{p}_l, \tau) \land (\tau \land \neg g \Rightarrow \phi'_{post})$$

Encoding sequences of statements a bit more difficult because of variable renaming (primed versions of $\tau$ and $\phi_{post}$).
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.

\[
PathC(\phi_{\text{pre}}, \textbf{while}^{\tau, \varphi} (g) \{ \vec{p}_i \} ; \vec{p}, \phi_{\text{post}}) =
\]
\[
(\phi_{\text{pre}} \Rightarrow \tau') \land PathC(\tau \land g, \vec{p}_i, \tau) \land PathC(\tau \land \neg g, \vec{p}, \phi_{\text{post}})
\]

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

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

\[
SafetyCond(exp, \mathcal{F}) = PathC(F_{\text{pre}}, exp, F_{\text{post}})
\]
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.

\[
PathC(\phi_{\text{pre}}, \text{while}^{\tau, \varphi}(g) \{\vec{p}_l\}; \vec{p}, \phi_{\text{post}}) = \]
\[
(\phi_{\text{pre}} \Rightarrow \tau') \land PathC(\tau \land g, \vec{p}_l, \tau) \land PathC(\tau \land \neg g, \vec{p}, \phi_{\text{post}})
\]

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

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

\[
\text{SafetyCond}(\exp, \mathcal{F}) = PathC(F_{\text{pre}}, \exp, F_{\text{post}})
\]
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.

\[
\text{PathC}(\phi_{\text{pre}}, \textbf{while}^{\tau, \varphi}(g) \{\vec{p}_l\} ; \vec{p}, \phi_{\text{post}}) = \\
(\phi_{\text{pre}} \Rightarrow \tau') \land \text{PathC}(\tau \land g, \vec{p}_l, \tau) \land \text{PathC}(\tau \land \neg g, \vec{p}, \phi_{\text{post}})
\]

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

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

\[
\text{SafetyCond}(\exp, \mathcal{F}) = \text{PathC}(\mathcal{F}_{\text{pre}}, \exp, \mathcal{F}_{\text{post}})
\]
Synthesis

Safety conditions

Example

\[ F = \left( x \geq 1, (i - 1)^2 \leq x < i^2 \right) \]

\[ \exp_{\text{sqrt}} = \]

\[ \text{choose} \ \{ [] g_1 \rightarrow s_1 \}; \]

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

\[ \text{choose} \ \{ [] g_3 \rightarrow s_3 \}; \]

\[ \text{SafetyCond}(\exp_{\text{sqrt}}, F) = \]

\[ (x \geq 1 \land g_1 \land s_1 \Rightarrow \tau') \land \]

\[ (\tau \land g_0 \land g_2 \land s_2 \Rightarrow \tau') \land \]

\[ (\tau \land \neg g_0 \land g_3 \land s_3 \Rightarrow (i' - 1)^2 \leq x' < i'^2) \]

where \( g_i, s_i \) and \( \tau \) are all unknowns.
Synthesis

Safety conditions

Example

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

\( \exp_{\text{sqrt}} = \)

\[
\begin{align*}
& \text{choose } \{[] g_1 \rightarrow s_1 \}; \\
& \text{while}^{\tau, \varphi} (g_0) \{ \text{choose } \{[] g_2 \rightarrow s_2 \}; \} ; \\
& \text{choose } \{[] g_3 \rightarrow s_3 \};
\end{align*}
\]

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

where \(g_i, s_i\) and \(\tau\) are all unknowns.
Synthesis

Well-formedness conditions

\[
WellFormTS(\{[] g_i \to s_i\}_i) = \left(\bigwedge_i \text{valid}(s_i)\right) \land \left(\bigvee_i g_i\right)
\]

where

- \(\text{valid}(s_i)\) 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

\[
WellFormCond(exp) = \bigwedge_{(\text{choose} \; \{[] g_i \to s_i\}_i) \in \text{cond}(exp)} WellFormTS(\{[] g_i \to s_i\}_i)
\]

where \(\text{cond}(exp)\) is the set of all \text{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

\[
\text{WellFormTS}(\{\lbrack g_i \rightarrow s_i \rbrack \}_i) = (\bigwedge_i \text{valid}(s_i)) \land (\bigvee_i g_i)
\]

where

- \(\text{valid}(s_i)\) 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

\[
\text{WellFormCond}(\text{exp}) = \bigwedge_{(\text{choose } \{\lbrack g_i \rightarrow s_i \rbrack \}_i) \in \text{cond}(\text{exp})} \text{WellFormTS}(\{\lbrack g_i \rightarrow s_i \rbrack \}_i)
\]

where \(\text{cond}(\text{exp})\) is the set of all \text{choose} statements in the expanded scaffold \(\text{exp}\).

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

\[
WellFormTS(\{[] g_i \rightarrow s_i \}_i) = \left( \bigwedge_i valid(s_i) \right) \land \\
\left( \bigvee_i g_i \right)
\]

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

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

where \( \text{cond}(exp) \) is the set of all \textbf{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

Example

\[ \text{exp}_{\text{sqrt}} = \]
\[
\begin{align*}
\text{choose} & \quad \{ [] g_1 \rightarrow s_1 \} ; \\
\text{while} & \quad \tau, \varphi \quad (g_0) \quad \{ \text{choose} \quad \{ [] g_2 \rightarrow s_2 \} ; \quad \} ; \\
\text{choose} & \quad \{ [] g_3 \rightarrow s_3 \} ; 
\end{align*}
\]

WellFormCond(\text{exp}_{\text{sqrt}}) = valid(s_1) \land valid(s_2) \land valid(s_3) \land g_1 \land g_2 \land g_3
Example

\[ \exp_{\sqrt{\cdot}} = \]

\[
\text{choose } \{ \square g_1 \rightarrow s_1 \}; \]

\[
\text{while } \tau; \varphi (g_0) \{ \text{choose } \{ \square g_2 \rightarrow s_2 \}; \} ; \]

\[
\text{choose } \{ \square g_3 \rightarrow s_3 \};
\]

\[
\text{WellFormCond}(\exp_{\sqrt{\cdot}}) = \text{valid}(s_1) \land \text{valid}(s_2) \land \text{valid}(s_3) \land \]

\[
g_1 \land g_2 \land g_3
\]
Synthesis

Progress conditions

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

where

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

\[ \text{RankCond}(\text{exp}) = \bigwedge \text{prog}(\textbf{while}^{\tau,\varphi} (g) \{ \vec{p} \}) \]
\[ (\textbf{while}^{\tau,\varphi} (g) \{ \vec{p} \}) \in \text{loops}(\text{exp}) \]

where \( \text{loops}(\text{exp}) \) is the set of all \textbf{while} statements in the expanded scaffold \( \text{exp} \).
Synthesis

Progress conditions

\[ prog(\text{while}^{\tau, \phi} (g) \{ \vec{p} \}) \equiv (r = \phi) \land (\tau \Rightarrow r \geq 0) \land \]

\[ PathC(\tau_{\text{end}} \land g, \text{end}(\vec{p}), r > \phi) \]

where

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

\[ \text{RankCond}(\text{exp}) = \bigwedge \text{prog}(\text{while}^{\tau, \phi}(g) \{ \vec{p} \}) \]

\[ (\text{while}^{\tau, \phi}(g) \{ \vec{p} \}) \in \text{loops}(\text{exp}) \]

where \( \text{loops}(\text{exp}) \) is the set of all \textbf{while} statements in the expanded scaffold \text{exp}.
Example

\[ \exp_{\text{sqrt}} = \]
\[
\begin{align*}
\text{choose} & \{ [] g_1 \rightarrow s_1 \}; \\
\text{while} & \tau; \varphi \ (g_0) \{ \text{choose} \ {[]} g_2 \rightarrow s_2 \}; \\
\text{choose} & \ {[]} g_3 \rightarrow s_3 \};
\end{align*}
\]

\[
\text{RankCond}(\exp_{\text{sqrt}}) = (r = \varphi) \land (\tau \Rightarrow r \geq 0) \land \\
(\tau \land g_0 \land g_2 \land s_2 \Rightarrow r' > \varphi')
\]
Example

\[ \exp_{\sqrt{\cdot}} = \]
\[
\text{choose} \ \{ \lhd g_1 \rightarrow s_1 \}; \\
\text{while}^{r;\varphi} (g_0) \{ \text{choose} \ \{ \lhd g_2 \rightarrow s_2 \}; \}; \\
\text{choose} \ \{ \lhd g_3 \rightarrow s_3 \}; \\
\]

\[
\text{RankCond}(\exp_{\sqrt{\cdot}}) = (r = \varphi) \land (r \geq 0) \land \\
(\tau \land g_0 \land g_2 \land s_2 \Rightarrow r' > \varphi')
\]
Synthesis

Entire synthesis algorithm

- **Input:**
  - Scaffold \( \langle F, D, R \rangle \)
  - Maximum number of transitions \( n \)
  - Proof domain \( D_{prf} \)

- **Output:** Executable program or FAIL

\[
\begin{align*}
\text{exp} & := \text{Expand}^{n,D_{prf}}_{D,R} (R_{flow}) ; \\
\text{sc} & := \text{SafetyCond}(\text{exp}, F) \land \text{WellFormCond}(\text{exp}) \land \text{RankCond}(\text{exp}) ; \\
\pi & := \text{Solver}(\text{sc}) ; \\
\text{if} \ (\text{unsat}(\pi)) \\
\text{return} \ FAIL ; \\
\text{return} \ \text{Exe}^{\pi}(\text{exp}) ;
\end{align*}
\]
Synthesis
Concretization algorithm

\[ \text{Exe}^\pi(p; \vec{p}) = \text{Exe}^\pi(p) ; \text{Exe}^\pi(\vec{p}) \]

\[ \text{Exe}^\pi(\text{while}^{\tau,\varphi}(g) \{ \vec{p} \}) = \text{while}^{\pi(\tau),\pi(\varphi)}(\pi(g)) \{ \text{Exe}^\pi(\vec{p}) \} \]

\[ \text{Exe}^\pi(\text{choose} \{ [] g \rightarrow s \}) = \text{if} (\pi(g)) \{ \text{Stmt}(\pi(s)) \} \]

\[ \text{Exe}^\pi(\text{choose} \{ [] g_i \rightarrow s_i \}_{i=1...n}) = \text{if} (\pi(g)) \{ \text{Stmt}(\pi(s)) \} \]

\[ \text{else} \{ \text{Exe}^\pi(\text{choose} \{ [] g_i \rightarrow s_i \}_{i=2...n}) \} \]

\[ \text{Stmt} \left( \bigwedge_{i=1...n} x_i = e_i \right) = t_1 := e_1 ; \ldots ; t_n := e_n ; \]

\[ x_1 := t_1 ; \ldots ; x_n := t_n ; \]
Synthesis
Concretization algorithm

\[ \text{Exe}^\pi(p; \vec{p}) = \text{Exe}^\pi(p) ; \text{Exe}^\pi(\vec{p}) \]

\[ \text{Exe}^\pi(\text{while}^{\tau, \varphi}(g) \{ \vec{p} \}) = \text{while}^{\pi(\tau), \pi(\varphi)}(\pi(g)) \{ \text{Exe}^\pi(\vec{p}) \} \]

\[ \text{Exe}^\pi(\text{choose} \{ [] g \rightarrow s \}) = \text{if} (\pi(g)) \{ \text{Stmt}(\pi(s)) \} \]
\[ \text{else} \{ \text{skip} \} \]

\[ \text{Exe}^\pi(\text{choose} \{ [] g_i \rightarrow s_i \}_{i=1\ldots n}) = \text{if} (\pi(g)) \{ \text{Stmt}(\pi(s)) \} \]
\[ \text{else} \{ \text{Exe}^\pi(\text{choose} \{ [] g_i \rightarrow s_i \}_{i=2\ldots n}) \} \]

\[ \text{Stmt}\left(\bigwedge_{i=1\ldots n} x_i = e_i\right) = t_1 := e_1 ; \ldots ; t_n := e_n ; \]
\[ x_1 := t_1 ; \ldots ; x_n := t_n ; \]
Synthesis

Example

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

\[
\tau : (v = i^2) \land (x \geq (i - 1)^2) \land (i \geq 1)
\]
\[
g_0 : v \leq x
\]
\[
\varphi : x - (i - 1)^2
\]
\[
s_1 : (v' = 1) \land (i' = 1) \land (x' = x) \land (r' = r)
\]
\[
s_2 : (v' = v + 2i + 1) \land (i' = i + 1) \land (x' = x) \land (r' = r)
\]
\[
s_3 : (v' = v) \land (i' = i) \land (x' = x) \land (r' = r)
\]
Synthesis

Example

\[
\begin{align*}
\left[ (x \geq 1 \land g_1 \land s_1 \Rightarrow \tau') \land (\tau \land g_0 \land g_2 \land s_2 \Rightarrow \tau') \land (\tau \land \neg g_0 \land g_3 \land s_3 \Rightarrow (i' - 1)^2 \leq x' < i'^2) \right] \land \\
\left[ \text{valid}(s_1) \land \text{valid}(s_2) \land \text{valid}(s_3) \right] \land \left[ (r = \varphi) \land (\tau \Rightarrow r \geq 0) \land (\tau \land g_0 \land g_2 \land s_2 \Rightarrow r' > \varphi') \right]
\end{align*}
\]

\[
\tau : (v = i^2) \land (x \geq (i - 1)^2) \land (i \geq 1)
\]

\[
g_0 : v \leq x
\]

\[
\varphi : x - (i - 1)^2
\]

\[
s_1 : (v' = 1) \land (i' = 1) \land (x' = x) \land (r' = r)
\]

\[
s_2 : (v' = v + 2i + 1) \land (i' = i + 1) \land (x' = x) \land (r' = r)
\]

\[
s_3 : (v' = v) \land (i' = i) \land (x' = x) \land (r' = r)
\]
Synthesis

Example

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

\[\tau : (v = i^2) \land (x \geq (i - 1)^2) \land (i \geq 1)\]

\[g_0 : v \leq x\]

\[\varphi : x - (i - 1)^2\]

\[s_1 : (v' = 1) \land (i' = 1) \land (x' = x) \land (r' = r)\]

\[s_2 : (v' = v + 2i + 1) \land (i' = i + 1) \land (x' = x) \land (r' = r)\]

\[s_3 : (v' = v) \land (i' = i) \land (x' = x) \land (r' = r)\]
Synthesis

Requirements for solvers

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

Tools

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. $[-] \land \forall k : [-] \Rightarrow [-]$

- VS$^3_{AX} = VS^3_{PA} +$ user-specified axioms over uninterpreted symbols
Experimental case studies

Tools

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. $[\neg] \land \forall k : [\neg] \Rightarrow [\neg]$

- $VS^3_{AX} = VS^3_{PA} +$ user-specified axioms over uninterpreted symbols
Experimental case studies

Tools

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. $[-] \land \forall k : [-] \Rightarrow [-]$

- VS$^3_{AX} = VS^3_{PA} +$ user-specified axioms over uninterpreted symbols
Experimental case studies

Tools

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. $[-] \land \forall k : [-] \Rightarrow [-]$

- **VS$^3_{AX} = VS^3_{PA} +$ user-specified axioms over uninterpreted symbols**
Experimental case studies

Flowgraphs with initialization and finalization

We instead treat loops \((\ast(T))\) in \textit{Expand} as \(\circ;\ast(T);\circ\) to make things easier for the verification tools.
Experimental case studies

Swapping of values

Example

\[ F_{pre} \equiv (x = c_1) \land (y = c_2) \]
\[ F_{post} \equiv (x = c_2) \land (y = c_1) \]
\[ R_{flow} \equiv \circ \]
\[ R_{comp} \equiv \emptyset \]
\[ R_{stack} \equiv \emptyset \]

Synthesizer generates various versions, including

\[
\text{Swap} (\text{int } x, \text{ int } y) \\
\begin{align*}
  x &:= x + y; \\
  y &:= x - y; \\
  x &:= x - y;
\end{align*}
\]
Experimental case studies
Swapping of values

Example

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

Synthesizer generates various versions, including

\[
\text{Swap}(\text{int } x, \text{ int } y) \\
| x := x + y ; \\
| y := x - y ; \\
| x := x - y ;
\]
Experimental case studies
Integral square root

Example

- $\mathcal{F} = \left( x \geq 1, (i - 1)^2 \leq x < i^2 \right)$
- $R_{\text{flow}} \equiv \ast (\circ)$ and $R_{\text{comp}} \equiv \emptyset$
- $R_{\text{stack}} \equiv \{ (\text{int}, 1) \} + \text{quadratic expressions in } D_{\text{exp}}, D_{\text{grd}} = \text{sequential search}$
- $R_{\text{stack}} = \{ (\text{int}, 2) \} + \text{linear expressions in } D_{\text{exp}}, D_{\text{grd}} = \text{sequential search}$
- $v := 1; i := 1$
- $\text{while}^\top \gamma (v \leq x)$
  - $v := v + 2i + 1; i++$
- $\text{return } i - 1$
- $R_{\text{stack}} = \{ (\text{int}, 3) \} + \text{quadratic + extra assumptions}$
  - $\ast \text{ binary search (temporaries hold search range)}$
Experimental case studies

Integral square root

Example

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

- \( R_{\text{stack}} \vdash \{(\text{int}, 1)\} + \text{quadratic expressions in } D_{\text{exp}}, D_{\text{grd}} = \text{sequential search} \)
- \( R_{\text{stack}} \vdash \{(\text{int}, 2)\} + \text{linear expressions in } D_{\text{exp}}, D_{\text{grd}} = \text{sequential search} \)
- \( R_{\text{stack}} \vdash \{(\text{int}, 3)\} + \text{quadratic + extra assumptions} \)
  - binary search (temporaries hold search range)
Experimental case studies

Integral square root

Example

\[ F = \left( x \geq 1, (i - 1)^2 \leq x < i^2 \right) \]

\[ R_{flow} = \ast (\circ) \text{ and } R_{comp} = \emptyset \]

\[ R_{stack} = \{ (\text{int}, 1) \} + \text{quadratic expressions in } D_{exp}, D_{grd} = \text{sequential search} \]

\[ R_{stack} = \{ (\text{int}, 2) \} + \text{linear expressions in } D_{exp}, D_{grd} = \text{sequential search} \]

\[ \nu := 1; \ i := 1; \]

\begin{verbatim}
while^{\tau, \varphi} (\nu \leq x) \\
| \quad \nu := \nu + 2i + 1; \quad i++; \\
return \quad i - 1;
\end{verbatim}

\[ R_{stack} = \{ (\text{int}, 3) \} + \text{quadratic + extra assumptions} \]

\[ \text{binary search (temporaries hold search range)} \]
Experimental case studies

Integral square root

Example

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

▸ $R_{\text{flow}} \equiv \ast (\circ)$ and $R_{\text{comp}} \equiv \emptyset$

▸ $R_{\text{stack}} \equiv \{(\text{int}, 1)\} + \text{quadratic expressions in } D_{\text{exp}}, \ D_{\text{grd}} = \text{sequential search}$

▸ $R_{\text{stack}} \equiv \{(\text{int}, 2)\} + \text{linear expressions in } D_{\text{exp}}, \ D_{\text{grd}} = \text{sequential search}$

$v := 1; \ i := 1;$

\[
\textbf{while}^{\mathcal{T}, \mathcal{Q}} (v \leq x) \\
\begin{align*}
| \quad v &:= v + 2i + 1; \quad i++; \\
\textbf{return} & \quad i - 1;
\end{align*}
\]

▸ $R_{\text{stack}} \equiv \{(\text{int}, 3)\} + \text{quadratic + extra assumptions}$
  ▸ 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, $R_{flow} = \ast (\ast (\circ))$
- $R_{stack} = \emptyset$: Bubble Sort and a non-standard version of Insertion Sort.
- $R_{stack} = \{(\text{int}, 1)\}$: Selection Sort.
Experimental case studies
Non-recursive sorting

Example

- $\mathcal{F} = (\text{true}, \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k + 1])$
- $D_{\text{exp}}$ includes swapping of array elements, $R_{\text{comp}}$ allows swapping only, $R_{\text{flow}} \doteq \ast (\ast (\circ))$

- $R_{\text{stack}} \doteq \emptyset$: Bubble Sort and a non-standard version of Insertion Sort.

- $R_{\text{stack}} \doteq \{(\text{int}, 1)\}$: 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, $R_{flow} = \ast (\ast (\circ))$
- $R_{stack} = \emptyset$: Bubble Sort and a non-standard version of Insertion Sort.
- $R_{stack} = \{(\text{int}, 1)\}$: Selection 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 $\odot$

- $R_{\text{stack}} = \emptyset$, $R_{\text{flow}} = \odot; \odot; \odot$: Merge Sort.
- $R_{\text{stack}} = \{(\text{int}, 1)\}$, $R_{\text{flow}} = \emptyset; \odot; \odot$: 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 $\otimes$

- $R_{stack} \doteq \emptyset$, $R_{flow} \doteq \otimes; \otimes;\circ$: Merge Sort.
- $R_{stack} \doteq \{(\text{int}, 1)\}$, $R_{flow} \doteq \circ; \otimes; \otimes$: 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 $\ast$

- $R_{stack} \doteq \emptyset$, $R_{flow} \doteq \ast; \ast; \circ$: Merge Sort.
- $R_{stack} \doteq \{(\text{int}, 1)\}$, $R_{flow} \doteq \circ; \ast; \ast$: 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 $VS_{QA}^3$ (quadratic expression handling) and inefficient $VS_{AX}^3$.
- Need a set of candidate predicates for $VS_{AX}^3$.

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$_{QA}^3$ (quadratic expression handling) and inefficient VS$_{AX}^3$.
- Need a set of candidate predicates for VS$_{AX}^3$

Scalability

- More efficient verifiers are needed.

Relevance

- Multiple solutions differ in performance, readability.
EOF