From Program Verification to Program Synthesis

Saurabh Srivastava
University of Maryland, College Park
saurabhs@cs.umd.edu

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

Abstract

This paper describes a novel technique for the synthesis of imperative programs. Automated program synthesis has the potential to make programming and the design of systems easier by allowing programs to be specified at a higher-level than executable code. In our approach, which we call proof-theoretic synthesis, the user provides an input-output functional specification, a description of the atomic operations in the programming language, and a specification of the synthesized program's looping structure, allowed stack space, and bound on usage of certain operations. Our technique synthesizes a program, if there exists one, that meets the input-output specification and uses only the given resources.

The insight behind our approach is to interpret program synthesis as generalized program verification, which allows us to bring verification tools and techniques to program synthesis. Our synthesis algorithm works by creating a program with unknown statements, guards, inductive invariants, and ranking functions. It then generates constraints that relate the unknowns and enforces three kinds of requirements: partial correctness, loop termination, and well-formedness conditions on program guards. We formalize the requirements that program verification tools must meet to solve these constraint and use tools from prior work as our synthesizers.

We demonstrate the feasibility of the proposed approach by synthesizing programs in three different domains: arithmetic, sorting, and dynamic programming. Using verification tools that we previously built in the VS^3 project we are able to synthesize programs for complicated arithmetic algorithms including Strassen's matrix multiplication and Bresenham's line drawing; several sorting algorithms; and several dynamic programming algorithms. For these programs, the median time for synthesis is 14 seconds, and the ratio of synthesis to verification time ranges between 1× to 92× (with an median of 7×), illustrating the potential of the approach.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program Synthesis; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory.

Keywords Proof-theoretic program synthesis, verification.

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01...$10.00

1. Introduction

Automated program synthesis, despite holding the promise of significantly easing the task of programming, has received little attention due to its difficulty. Being able to mechanically construct programs has wide-ranging implications. Mechanical synthesis yields programs that are correct-by-construction. It relieves the tedium and error associated with programming low-level details, can aid in automated debugging and in general leaves the human programmer free to deal with the high-level design of the system. Additionally, synthesis could discover new non-trivial programs that are difficult for programmers to build.

In this paper, we present an approach to program synthesis that takes the correct-by-construction philosophy of program design [14, 18, 38] and shows how it can be automated. Program verification tools routinely synthesize program proofs in the form of inductive invariants for partial correctness and ranking functions for termination. We encode the synthesis problem as a verification problem by encoding program guards and statements as logical facts that need to be discovered. This allows us to use certain verification tools for synthesis. The verification tool infers the invariants and ranking functions as usual, but in addition infers the program statements, yielding automated program synthesis. We call our approach *proof-theoretic synthesis* because the proof is synthesized alongside the program.

We define the synthesis task as requirements on the output program: functional requirements, requirements on the form of program expressions and guards, and requirements on the resources used. The key to our synthesis algorithm is the reduction from the synthesis task to three sets of constraints. The first set are safety conditions that ensure the partial correctness of the loops in the program. The second set are well-formedness conditions on the program guards and statements, such that the output from the verification tool (facts corresponding to program guards and statements) correspond to valid guards and statements in an imperative language. The third set are progress conditions that ensure that the program terminates. To our knowledge, our approach is the first that automatically synthesizes programs and their proofs, while previous approaches have either used given proofs to extract programs [27] or made no attempt to generate the proof. Some approaches, while not generating proofs, do ensure correctness for a limited class of finitizable programs [29].

To illustrate our approach, we next show how to synthesize Bresenham's line drawing algorithm. This example is an ideal candidate for automated synthesis because, while the program's requirements are simple to specify, the actual program is quite involved.

1.1 Motivating Example

As a motivating example, we consider a well-known algorithm from the graphics community called Bresenham's line drawing algorithm, shown in Figure 1(a). The algorithm computes (and

```
(a) Bresenhams(int $X,Y$) {
      $v_1$:=$2Y-X$; $y$:=0; $x$:=0;
      while ($x \leq X$)
        | $out[x]$:=$y$;
        | if ($v_1 < 0$)
        |    $v_1$:=$v_1$+$2Y$;
        | else
        |    $v_1$:=$v_1$+2($Y$-$X$); $y$++;
        | $x$++;
      return $out$;
    }
```

```
(b) Bresenhams(int $X,Y$) {
      []true $\rightarrow$ $v_1'$=$2Y-X$ $\wedge$ $y'$=0 $\wedge$ $x'$=0
      while ($x \leq X$)
        | []$v_1 < 0 \rightarrow$ $out'$=upd($out$,$x$,$y$) $\wedge$ $v_1'$=$v_1$+$2Y$ $\wedge$ $y'$=$y$ $\wedge$ $x'$=$x$+1
        | []$v_1 \geq 0 \rightarrow$ $out'$=upd($out$,$x$,$y$) $\wedge$ $v_1'$=$v_1$+2($Y$-$X$) $\wedge$ $y'$=$y$+1 $\wedge$ $x'$=$x$+1
      return $out$;
    }
```

(c) Invariant τ : $0 < Y \leq X \wedge v_1 = 2(x+1)Y-(2y+1)X \wedge$ $2(Y-X) \leq v_1 \leq 2Y \wedge$ $\forall k : 0 \leq k < x \Rightarrow 2|out[k]-(Y/X)k| \leq 1$

Ranking function φ : $X - x$

Figure 1. (a) Bresenham's line drawing algorithm (b) The invariant and ranking function that prove partial correctness and termination, respectively. (c) The algorithm written in transition system form, with statements as equality predicates, guarded appropriately.

writes to the output array out) the discrete best-fit line from $(0, 0)$ to (X, Y), where the point (X, Y) is in the NE half-quadrant, i.e., $0 < Y \leq X$. The best-fit line is one that does not deviate more than half a pixel away from the real line, i.e., $|y - (Y/X)x| \leq 1/2$. For efficiency, the algorithm computes the pixel values (x, y) of this best-fit line using only linear operations, but the computation is non-trivial and the correctness of the algorithm is also not evident.

The specification for this program is succinctly written in terms of its precondition $\tau_{\texttt{pre}}$ and postcondition $\tau_{\texttt{post}}$:

$$\begin{array}{rl} \tau_{\texttt{pre}}: & 0 < Y \leq X \\ \tau_{\texttt{post}}: & \forall k : 0 \leq k \leq X \Rightarrow 2|out[k]-(Y/X)k| \leq 1 \end{array}$$

Notice that in the postcondition, we have written the assertion outside the loop body for clarity of presentation, but it can easily be rewritten, as a quantifier-free assertion, inside. Bresenham proposed the program shown in Figure 1(a) to implement this specification. The question we answer is whether it is possible to synthesize the program given just the specification and a description of the available resources (control flow, stack space and operations). Let us stepwise develop the idea behind synthesis starting from the verification problem for the given program.

Observe that we can write program statements as equality predicates and acyclic fragments as transition systems. For example, we can write $x := e$ as $x' = e$, where x' is a renaming of x to its output value. We will write statements as equalities between the output (primed) versions of the variables and the expression (over the unprimed versions of the variables). Also, guards that direct control flow in an imperative program can now be seen as guards for statement facts in a transition system. Figure 1(b) shows our example written in transition system form. To prove partial correctness, one can write down the inductive invariant for the loop and verify that the verification condition for the program is in fact valid. The verification condition consists of four implications for the four paths corresponding to the entry, exit, and one each for the branches in the loop. Using standard verification condition generation, with the precondition $\tau_{\texttt{pre}}$ and postcondition $\tau_{\texttt{post}}$, and writing the renamed version of invariant τ as τ', these are

$$\begin{array}{rcl} \tau_{\texttt{pre}} \wedge s_{\texttt{entry}} & \Rightarrow & \tau' \\ \tau \wedge \neg g_{\texttt{loop}} & \Rightarrow & \tau_{\texttt{post}} \\ \tau \wedge g_{\texttt{loop}} \wedge g_{\texttt{body1}} \wedge s_{\texttt{body1}} & \Rightarrow & \tau' \\ \tau \wedge g_{\texttt{loop}} \wedge g_{\texttt{body2}} \wedge s_{\texttt{body2}} & \Rightarrow & \tau' \end{array} \quad (1)$$

where we use symbols for the various parts of the program:

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

With a little bit of work, one can *validate* that the invariant τ shown in Figure 1(c) satisfies Eq. (1). Checking the validity of given invariants can be automated using SMT solvers [10]. In fact, powerful program verification tools now exist that can generate fixed-point solutions—inductive invariants such as τ—automatically using constraint-based techniques [6, 21, 32], abstract interpretation [9] or model checking [3]. There are also tools that can prove termination [7]—by inferring ranking functions such as φ—and together with the safety proof provide a proof for total correctness.

The insight behind our paper is to ask the question, if we can infer τ in Eq. (1), then is it possible to *infer* the guards g_i's and the statements s_i's at the same time? We have found that we can indeed infer guards and statements as well, by suitably encoding programs as transition systems, asserting appropriate constraints, and then leveraging program verification techniques to do a systematic (lattice-theoretic) search for unknowns in the constraints. Here the unknowns now represent both the invariants and the statements and guards. It turns out that a direct solution to the unknown guards and statements may be uninteresting, i.e., it may not correspond to real programs. But we illustrate that we can impose additional *well-formedness* constraints on the unknown guards and statements such that any solution to this new set of constraints corresponds to a valid, real program. Additionally, even if we synthesize valid programs, it may be that the programs are non-terminating. Therefore we need to impose additional *progress* constraints that ensure that the synthesized programs are ones that we can actually run. We now illustrate the need for these well-formedness and progress constraints over our example.

Suppose that the statements $s_{\texttt{entry}}$, $s_{\texttt{body1}}$ and $s_{\texttt{body2}}$, are unknown. A trivial satisfying solution to Eq. (1) may set all these unknowns to `false`. If we use a typical program verification tool that computes least fixed-points starting from $\bot$, then indeed, it will output this solution. On the other hand, let us make the conditional guards $g_{\texttt{body1}}$ and $g_{\texttt{body2}}$ unknown. Again, $g_{\texttt{body1}} = g_{\texttt{body2}} = \texttt{false}$ is a satisfying solution. We get uninteresting solutions because the unknowns are not constrained enough to ensure valid statements and control-flow. Statement blocks are modeled as $\bigwedge_i x_i' = e_i$, with one equality for each output variable x_i' and expressions e_i are over input variables. Therefore, `false` does not correspond to any valid block. Similarly $g_{\texttt{body1}} = g_{\texttt{body2}} = \texttt{false}$ does not correspond to any valid conditional with two branches. For example, consider `if` (g) S_1 `else` S_2 with two branches. Note how S_1 and S_2 are guarded by g and $\neg g$, respectively, and $g \vee \neg g$ holds. For every valid conditional, the disjunction of the guards is always a tautology. In verification, the program syntax and semantics ensure the *well-formedness* of acyclic fragments. In synthesis, we will need to explicitly constrain well-formedness of acyclic fragments (Section 3.4).

Next, suppose that the loop guard $g_{\texttt{loop}}$ is unknown. In this case if we attempt to solve for the unknowns τ and $g_{\texttt{loop}}$, then one valid solution assigns $\tau = g_{\texttt{loop}} = \texttt{true}$, which corresponds to an non-terminating loop. In verification, we were only concerned with partial correctness and assumed that the program was terminating. In synthesis, we will need to explicitly *encode progress* by inferring appropriate ranking functions, like φ in Figure 1(c), to prevent the synthesizer from generating non-terminating programs (Section 3.5).

Note that our aim is not to solve the completely general synthesis problem for a given *functional specification*. Guards and statements are unknowns but they take values from given domains, specified by the user as *domain constraints*, so that a lattice-theoretic search can be performed by existing program verification tools. Also notice that we did not attempt to change the number of invariants or the invariant position in the constraints. This means that we assume a given looping or *flowgraph structure*, e.g., one loop for our example. Lastly, as opposed to verification, the set of program variables is not known, and therefore we need a specification of the *stack space* available and also a bound on the type of *computations* allowed.

We use the specifications to construct an *expansion* that is a program with unknown symbols and construct safety conditions over the unknowns. We then impose the additional well-formedness and progress constraints. We call the new constraints *synthesis conditions* and hope to find solutions to them using program verification tools. The constraints generated are non-standard, and therefore to solve them we need verification tools that satisfy certain properties. Verification tools we developed in previous work [32, 21] indeed have those properties. We use them to efficiently solve the synthesis conditions to synthesize programs, with a very acceptable slowdown over verification.

The guards, statements and proof terms for the example in this section come from the domain of arithmetic. Therefore, a program verification tool for arithmetic would be appropriate. For programs whose guards and statements are more easily expressed in other domains, a corresponding verification tool for that domain should be used. In fact, we have employed tools for the domains of arithmetic and predicate abstraction for proof-theoretic synthesis with great success. Our objective is to reuse existing verification technology—that started with invariant validation and progressed to invariant inference—and push it further to *program synthesis*.

1.2 Contributions

This paper makes the following contributions:

- We present a novel way of specifying a synthesis task as a triple consisting of the functional specification, the domains of expressions and guards that appear in the synthesized program, and resource constraints that the program is allowed to use (Section 2).
- We view program synthesis as generalized program verification. We formally define constraints, called synthesis conditions, that can be solved using verification tools (Section 3).
- We present requirements that program verification tools must meet in order to be used for synthesis of program statements and guards (Section 4).
- We build synthesizers using verification tools and present synthesis results for the three domains of arithmetic, sorting and dynamic programming (Section 5).

2. The Synthesis Scaffold and Task

We now elaborate on the specifications that a proof-theoretic approach to synthesis requires and how these also allow the user to specify the space of interesting programs.

We describe the synthesis problem using a *scaffold* of the form

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

The three components are as follows:

1. Functional Specification The first component $\mathcal{F}$ of a scaffold describes the desired precondition and postcondition of the synthesized program. Let $\vec{v_{\mathtt{in}}}$ and $\vec{v_{\mathtt{out}}}$ be the vectors containing the input and output variables, respectively. Then a functional specification $\mathcal{F} = (F_{\mathtt{pre}}(\vec{v_{\mathtt{in}}}), F_{\mathtt{post}}(\vec{v_{\mathtt{in}}}, \vec{v_{\mathtt{out}}}))$ is a tuple containing the formulae that hold at the entry and exit program locations. For example, for the program in Figure 1, $F_{\mathtt{pre}}(X, Y) \doteq (0 < Y \leq X$ and $F_{\mathtt{post}}(X, Y, out) \doteq \forall k : 0 \leq k \leq X \Rightarrow 2(Y/X)k - 1 \leq 2out[k] \leq 2(Y/X)k + 1$.

2. Domain Constraints The second component $\mathcal{D}$ of the scaffold describes the domains for expressions and guards in the synthesized program. The domain specification $\mathcal{D} = (D_{\mathtt{exp}}, D_{\mathtt{grd}})$ is a tuple that constrains the respective components:

2a. *Program Expressions:* The expressions manipulated by the program come from the domain $D_{\mathtt{exp}}$.

2b. *Program Guards:* The logical guards (boolean expressions) used to direct control flow in the program come from the domain $D_{\mathtt{grd}}$.

For example, for the program in Figure 1, the domains $D_{\mathtt{exp}}, D_{\mathtt{grd}}$ are both linear arithmetic.

3. Resource Constraints The third component $\mathcal{R}$ of the scaffold describes the resources that the synthesized program can use. The resource specification $\mathcal{R} = (R_{\mathtt{flow}}, R_{\mathtt{stack}}, R_{\mathtt{comp}})$ is a triple of resource templates that the user must specify for the flowgraph, stack and computation, respectively:

3a. *Flowgraph Template* We restrict attention to structured programs (those that are goto-less, or whose flowgraphs are reducible [22]). The structured nature of such flowgraphs allows us to describe them using simple strings. The user specifies a string $R_{\mathtt{flow}}$ from the following grammar:

$$T \ ::= \ \circ \ \mid \ *(T) \ \mid \ T;T \tag{2}$$

where $\circ$ denotes an acyclic fragment of the flow graph, $*(T)$ denotes a loop containing the body T and $T;T$ denotes the sequential composition of two flow graphs. For example, for the program in Figure 1, $R_{\mathtt{flow}} = \circ;*(\circ)$.

3b. *Stack Template* A map $R_{\mathtt{stack}} : \mathtt{type} \rightarrow \mathtt{int}$ indicating the number of extra temporary variables of each type available to the program. For example, for the program in Figure 1, $R_{\mathtt{stack}} = (\mathtt{int}, 1)$.

3c. *Computation Template* At times it may be important to put an upper bound on the number of times an operation is performed inside a procedure. A map $R_{\mathtt{comp}} : \mathtt{op} \rightarrow \mathtt{int}$ of operations $\mathtt{op}$ to the upper bound specifies this constraint. For example, for the program in Figure 1, $R_{\mathtt{comp}} = \emptyset$ which indicates that there are no constraints on computation.

On the one hand, the resource templates make synthesis tractable by enabling a systematic lattice-theoretic search, while on the other they allow the user to specify the space of interesting programs and can be used as a feature. For instance, the user may wish to reduce memory consumption at the expense of a more complex flowgraph and still meet the functional specification. If the user does not care, then the resource templates can be considered optional and left unspecified. In this case, the synthesizer can iteratively enumerate possibilities for each resource and attempt synthesis with increasing resources.

2.1 Picking a proof domain and a solver for the domain

Our synthesis approach is proof-theoretic and we synthesize the proof terms, i.e., invariants and ranking functions, alongside the program. These proof terms will take values from a suitably chosen *proof domain* $D_{\texttt{prf}}$. Notice that $D_{\texttt{prf}}$ will be at least as expressive as $D_{\texttt{grd}}$ and $D_{\texttt{exp}}$. The user chooses an appropriate proof domain and also picks a solver capable of handling that domain. We will use program verification tools as solvers and typically, the user will pick the most powerful verification tool available for the chosen proof domain.

2.2 Synthesis Task

Given a scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$, we call an executable program *valid* with respect to the scaffold if it meets the following conditions.

- When called with inputs $\vec{v_{\texttt{in}}}$ that satisfy $F_{\texttt{pre}}(\vec{v_{\texttt{in}}})$ the program terminates, and the resulting outputs $\vec{v_{\texttt{out}}}$ satisfy $F_{\texttt{post}}(\vec{v_{\texttt{in}}}, \vec{v_{\texttt{out}}})$. There are associated invariants and ranking functions that provide a proof of this fact.
- There is a program loop (with an associated loop guard g) corresponding to each loop annotation (specified by "$*$") in the flowgraph template $R_{\texttt{flow}}$. The program contains statements from the following imperative language IML for each acyclic fragment (specified by "$\circ$").

  $$S \;\; ::= \;\; \texttt{skip} \;\mid\; S;S \;\mid\; x := e \;\mid\; \texttt{if}\; g \;\texttt{then}\; S \;\texttt{else}\; S$$

  Where x denotes a variable, e denotes some expression, and g denotes some predicate. (Memory reads and writes are modeled using memory variables and select/update expressions.) The domain of expressions and guards is as specified by the scaffold, i.e., $e \in D_{\texttt{exp}}$ and $g \in D_{\texttt{grd}}$.
- The program only uses as many local variables as specified by $R_{\texttt{stack}}$ in addition to the input and output variables $\vec{v_{\texttt{in}}}, \vec{v_{\texttt{out}}}$.
- Each elementary operation only appears as many times as specified in $R_{\texttt{comp}}$.

EXAMPLE 1 (Square Root). *Let us consider a scaffold with functional specification* $\mathcal{F} = (x \geq 1, (i-1)^2 \leq x < i^2)$, *which states that the program computes the integral square root of the input* x *, i.e.,* $i - 1 = \lfloor \sqrt{x} \rfloor$. *Also, let the domain constraints* $D_{\texttt{exp}}, D_{\texttt{grd}}$ *be limited to linear arithmetic expressions, which means that the program cannot use any native square root or squaring operations. Lastly, let the* $R_{\texttt{flow}}$, $R_{\texttt{stack}}$ *and* $R_{\texttt{comp}}$ *be* $\circ;*(\circ);\circ$, $\{(\texttt{int}, 1)\}$ *and* $\emptyset$, *respectively. A program that is valid with respect to this scaffold is the following:*

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

Invariant τ:
$v{=}i^2 \wedge x \geq (i{-}1)^2 \wedge i \geq 1$

Ranking function φ:
$x - (i{-}1)^2$

where v, i *are the additional stack variable and loop iteration counter (and reused in the output), respectively. Also, the loop is annotated with the invariant* τ *and ranking function* φ *as shown, and which prove partial correctness and termination, respectively.*

In the next two sections, we formally describe the steps of our synthesis algorithm. We first generate *synthesis conditions* (Section 3), which are constraints over unknowns for statements, guards, loop invariants and ranking functions. We then observe that they resemble verification conditions, and we can employ verification tools, if they have certain properties, to solve them (Section 4).

3. Synthesis Conditions

In this section, we define and construct *synthesis conditions* for an input scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$. Using the resource specification $\mathcal{R}$, we first generate a program with unknowns corresponding to the fragments we wish to synthesize. Synthesis conditions then specify constraints on these unknowns and ensure partial correctness, loop termination and well-formedness of control-flow. We begin our discussion by motivating the representation we use for acyclic fragments in the synthesized program.

3.1 Using Transition Systems to Represent Acyclic Code

Suppose we want to infer a set of (straight-line) statements that transform a precondition $\phi_{\texttt{pre}}$ to a postcondition $\phi_{\texttt{post}}$, where the relevant program variables are x and y. One approach might be to generate statements that assign unknown expressions e_x and e_y to x and y, respectively:

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

Then we can use Hoare's axiom for assignment to generate the verification condition $\phi_{\texttt{pre}} \Rightarrow (\phi_{\texttt{post}}[y \mapsto e_y])[x \mapsto e_x]$. However, this verification condition is hard to automatically reason about because it contains substitution into unknowns. Even worse, we have restricted the search space by requiring the assignment to y to follow the assignment to x, and by specifying exactly two assignments.

Instead we will represent the computation as a transition system which provides a much cleaner mechanism for reasoning when program statements are unknown. A *transition* in a transition system is a (possibly parallel) mapping of the input variables to the output variables. Variables have an input version and an output version (indicated by primed names), which allows them to change state. For our example, we can write a single transition:

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

Here $\phi'_{\texttt{post}}$ is the postcondition, written in terms of the output variables, and e_x, e_y are expressions over the input variables. The verification condition corresponding to this tuple is $\phi_{\texttt{pre}} \wedge x' = e_x \wedge y' = e_y \Rightarrow \phi'_{\texttt{post}}$. Note that every state update (assignment) can always be written as a transition.

We can extend this approach to arbitrary acyclic program fragments. A *guarded transition* (written $[]g \rightarrow s$) contains a statement s that is executed only if the quantifier-free guard g holds. A *transition system* consists of a set $\{[]g_i \rightarrow s_i\}_i$ of guarded transitions. It is easy to see that a transition system can represent any arbitrary acyclic program fragment by suitably enumerating the paths through the acyclic fragment. The verification condition for $\{\phi_{\texttt{pre}}\}\{[]g_i \rightarrow s_i\}_i\{\phi'_{\texttt{post}}\}$ is simply $\bigwedge_i (\phi_{\texttt{pre}} \wedge g_i \wedge s_i \Rightarrow \phi'_{\texttt{post}})$.

In addition to the simplicity afforded by the lack of any ordering, the constraints from transition systems are attractive for synthesis as the program statements s_i and guards g_i are facts just like the pre- and postconditions $\phi_{\texttt{pre}}$ and $\phi'_{\texttt{post}}$. Given the lack of differentiation, any (or all) can be unknowns in these *synthesis conditions*. This distinguishes them from verification conditions which can only have unknown invariants, or often the invariants must be known as well.

Synthesis conditions can thus be viewed as generalizations of verification conditions. Program verification tools routinely infer fixed-point solutions (invariants) that satisfy the verification conditions with known statements and guards. With our formulation of statements and guards as just additional facts in the constraints, it is possible to use (sufficiently general) verification tools to infer invariants *and* program statements and guards. *Synthesis conditions serve an analogous purpose to synthesis as verification conditions do to verification. If a program is correct (verifiable), then its verification condition is valid. Similarly, if a valid program exists for a scaffold, then its synthesis condition has a satisfying solution.*

3.2 Expanding a flowgraph

We synthesize code fragments for each acyclic fragment and loop annotation in the flowgraph template as follows:

- *Acyclic fragments:* For each acyclic fragment annotation "$\circ$", we infer a transition system $\{g_i \rightarrow s_i\}_i$, i.e., a set of assignments s_i, stated as conjunctions of equality predicates, guarded by quantifier-free first-order-logic (FOL) guards g_i such that the disjunction of the guards is a tautology. Suitably constructed equality predicates and quantifier-free FOL guards are later translated to executable code—assignment statements and conditional guards, respectively—in the language IML.
- *Loops:* For each loop annotation "$*$" we infer three elements. The first is the *inductive loop invariant* τ, which establishes partial correctness of each loop iteration. The second is the *ranking function* φ, which proves the termination of the loop. Both the invariant and ranking function take values from the proof domain, i.e., $\tau, \varphi \in D_{\mathtt{prf}}$. Third, we infer a quantifier-free FOL loop guard g.

Formally, the output of expanding flowgraphs will be a program in the transition system language TSL (note the correspondence to the flowgraph grammar from Eq. 2):

$$p \ ::= \ \mathtt{choose}\ \{[]g_i \rightarrow s_i\}_i \ \ | \ \ \mathtt{while}^{\tau,\varphi}(g)\ \mathtt{do}\ \{p\} \ \ | \ \ p;p$$

Here each s_i is a conjunction of equality predicates, i.e., $\bigwedge_j\ (x_j = e_j)$. We will use $\vec{p}$ to denote a sequence of program statements in TSL. Note that we model memory read and updates using select/update predicates. Therefore, in $x = e$ the variable x could be a memory variable and e could be a memory select or update expression.

Given a string for a flowgraph template, we define an expansion function $\mathtt{Expand} : \mathtt{int} \times D_{\mathtt{prf}} \times \mathcal{R} \times \mathcal{D} \times R_{\mathtt{flow}} \rightarrow$ TSL that introduces fresh unknowns for missing guards, statements and invariants that are to be synthesized. $\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(R_{\mathtt{flow}})$ expands a flowgraph $R_{\mathtt{flow}}$ and is parametrized by an integer n that indicates the number of transition each acyclic fragment will be expanded to, the proof domain and the resource and domain constraints. The expansion outputs a program in the language TSL.

$$\begin{array}{rcll}
\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(\circ) & = & \mathtt{choose}\ \{[]g_i \rightarrow s_i\}_{i=1..n} & g_i, s_i : \textit{fresh unknowns} \\
\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(*(T)) & = & \mathtt{while}^{\tau,\varphi}\ (g)\ \{ & \tau, \varphi, g : \textit{fresh unknowns} \\
 & & \quad \mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T); & \\
 & & \} & \\
\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T_1;T_2) & = & \mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T_1);\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T_2) &
\end{array}$$

Each unknown g, s, τ generated during the expansion has the following domain inclusion constraints.

$$\begin{array}{rcl}
\tau & \in & D_{\mathtt{prf}}|_V \\
g & \in & D_{\mathtt{grd}}|_V \\
s & \in & \bigwedge_i x_i = e_i \quad \text{where } x_i \in V, e_i \in D_{\mathtt{exp}}|_V
\end{array}$$

Here $V = \vec{v_{\mathtt{in}}} \cup \vec{v_{\mathtt{out}}} \cup T \cup L$ is the set of variables: the input $\vec{v_{\mathtt{in}}}$ and output $\vec{v_{\mathtt{out}}}$ variables, the set of temporaries (local variables) T as specified by $R_{\mathtt{stack}}$, and the set of iteration counters and ranking function tracker variables is L (which we elaborate on later), one for each loop in the expansion. The restriction of the domains by the variable set V indicates that we are interested in the fragment of the domain over the variables in V. Also, the set of operations in e_i is bounded by $R_{\mathtt{comp}}$.

The expansion has some similarities to the notion of a user-specified *sketch* in previous approaches [31, 29]. However, the unknowns in the expansion here are more expressive than the integer unknowns considered earlier, and this allows us to perform a lattice search as opposed to the combinatorial approaches proposed earlier. Notice that the unknowns τ, g, s, φ we introduce can all be interpreted as boolean formulae (τ, g naturally; s using our transition modeling; and φ as $\varphi > c$, for some constant c), and consequently ordered in a lattice.

EXAMPLE 2. *Let us revisit the integral square root computation from Example 1. Expanding the flowgraph template* $\circ;*(\circ);\circ$ *with* $n = 1$ *yields* exp_{sqrt}*:*

$$\begin{array}{l}
\mathtt{choose}\ \{[]g_1 \rightarrow s_1\}\ ; \\
\mathtt{while}^{\tau,\varphi}\ (g_0)\ \{ \\
\quad \mathtt{choose}\ \{[]g_2 \rightarrow s_2\}\ ; \\
\}; \\
\mathtt{choose}\ \{[]g_3 \rightarrow s_3\}
\end{array}
\qquad
\begin{array}{rcl}
\tau & \in & D_{\mathtt{prf}}|_V \\
g_1, g_2, g_3 & \in & D_{\mathtt{grd}}|_V \\
s_1, s_2, s_3 & \in & \bigwedge_i x_i = e_i \\
 & & x_i \in V, e_i \in D_{\mathtt{exp}}|_V
\end{array}$$

where $V = \{x, i, r, v\}$*. The variables* i *and* r *are the loop iteration counter and ranking function tracker variable, respectively, and* v *is the additional local variable. Also, the chosen domains for proofs* $D_{\mathtt{prf}}$*, guards* $D_{\mathtt{grd}}$ *and expressions* $D_{\mathtt{exp}}$ *are FOL facts over quadratic expressions, FOL facts over linear arithmetic and linear arithmetic, respectively.*

Notice that the expansion encodes everything specified by the domain and resource constraints and the chosen proof domain. The only remaining specification is $\mathcal{F}$, which we will use in the next section to construct safety conditions over the expanded scaffold.

3.3 Encoding Partial Correctness: Safety Conditions

Now that we have the expanded scaffold we need to collect the constraints (safety conditions) for partial correctness implied by the simple paths in the expansion. *Simple paths* (straight-line sequence of statements) start at a loop header $F_{\mathtt{pre}}$ and end at a loop header or program exit. The loop headers, program entry and program exit are annotated with invariants, precondition $F_{\mathtt{pre}}$ and postcondition $F_{\mathtt{post}}$, respectively.

Let ϕ denote formulae, that represent pre and postconditions and constraints. Then we define $\mathtt{PathC} : \phi \times \text{TSL} \times \phi \rightarrow \phi$ as a function that takes a precondition, a sequence of statements and a postcondition and outputs safety constraints that encode the validity of the Hoare triple. Let us first describe the simple cases of constraints from a single acyclic fragment and loop:

$$\begin{array}{l}
\mathtt{PathC}(\phi_{\mathtt{pre}}, (\mathtt{choose}\ \{[]g_i \rightarrow s_i\}_i\ ), \phi_{\mathtt{post}}) = \\
\quad \bigwedge_i (\phi_{\mathtt{pre}} \wedge g_i \wedge s_i \Rightarrow \phi_{\mathtt{post}}{}') \\
\mathtt{PathC}(\phi_{\mathtt{pre}}, (\mathtt{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}), \phi_{\mathtt{post}}) = \\
\quad \phi_{\mathtt{pre}} \Rightarrow \tau' \wedge \mathtt{PathC}(\tau \wedge g, \vec{p_l}, \tau) \wedge (\tau \wedge \neg g \Rightarrow \phi_{\mathtt{post}}{}')
\end{array}$$

Here $\phi_{\mathtt{post}}{}'$ and τ' are the postcondition $\phi_{\mathtt{post}}$ and invariant τ but with all variables renamed to their output (primed) versions. Since the constraints need to refer to *output* postconditions and invariants the rule for a sequence of statements is a bit complicated. For simplicity of presentation, we assume that acyclic annotations do not appear in succession. This assumption holds without loss of generality because it is always possible to collapse consecutive acyclic fragments, e.g., two consecutive acyclic fragments with n transitions each can be collapsed into a single acyclic fragment with n^2 transitions. For efficiency, it is prudent to not make this assumption in practice, and the construction here generalizes easily. For a sequence of statements in TSL, under the above assumptions, there are three cases to consider. First, a loop followed by statements $\vec{p}$. Second, an acyclic fragment followed by just a loop. Third, an acyclic fragment, followed by a loop, followed by statements $\vec{p}$. Each of these generates the following, respective, constraints:

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

The safety condition for a scaffold with functional specification $\mathcal{F} = (F_{\texttt{pre}}, F_{\texttt{post}})$, flowgraph template $R_{\texttt{flow}}$, and expansion $exp = \texttt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\texttt{prf}}}(R_{\texttt{flow}})$ is given by:

$$\texttt{SafetyCond}(exp, \mathcal{F}) = \texttt{PathC}(F_{\texttt{pre}}, exp, F_{\texttt{post}}) \qquad (3)$$

EXAMPLE 3. *Consider the expanded scaffold (from Example 2) and the functional specification $\mathcal{F}$ (from Example 1) for integral square root. The loop divides the program into three simple paths, which results in $\textit{SafetyCond}(exp_{sqrt}, \mathcal{F})$:*

$$\begin{array}{rcll} x \geq 1 \wedge g_1 \wedge s_1 & \Rightarrow & \tau' & \wedge \\ \tau \wedge g_0 \wedge g_2 \wedge s_2 & \Rightarrow & \tau' & \wedge \\ \tau \wedge \neg g_0 \wedge g_3 \wedge s_3 & \Rightarrow & (i'-1)^2 \leq x' \wedge x' < i'^2 & \end{array}$$

Notice that g_i, s_i, τ are all unknown placeholder symbols.

3.4 Encoding Valid Control: Well-formedness Conditions

We next construct constraints to ensure the well-formedness of `choose` statements. In the preceding development, we treated each path through the `choose` statement as independent. In any executable program control will always flow through at least one branch/transition of the statement, and each transition will contain well-formed assignment statements. We first describe a constraint that encodes this directly and then discuss an alternative way of ensuring well-formedness of transition guards.

Non-iterative upper bounded search During the expansion of a scaffold, we can choose n to be greater than the number of transitions expected in any acyclic fragment. Any excess transitions will have their guards instantiated to `false`. For any statement `choose` $\{[]g_i \to s_i\}$ in the expansion, we impose the well-formedness constraint:

$$\begin{array}{rll} \texttt{WellFormTS}(\{[]g_i \to s_i\}_i) \doteq & \left(\bigwedge_i \texttt{valid}(s_i)\right) & \textit{Valid transition} \\ & \wedge \left(\bigvee_i g_i\right) & \textit{Covers space} \end{array} \qquad (4)$$

Here the predicate $\texttt{valid}(s_i)$ ensures *one and only one equality assignment* to each variable in s_i. This condition ensures that each s_i corresponds to a well-formed transition that can be translated to executable statements. The second term constrains the combination of the guards to be a tautology. Note that this is important to ensure that each transition system is well-formed and can be converted to a valid executable conditional. For example, consider the executable conditional `if` (G) `then` $x := E_1$ `else` $x := E_2$. The corresponding transition system is $\{[]g_1 \to (x' = E_1), []g_2 \to (x' = E_2)\}$, where $g_1 = G$ and $g_2 = \neg G$ and $g_1 \vee g_2$ holds. In *every* well-formed executable conditional the disjunction of the guards will be a tautology. The second term imposes this constraint.

Notice that this construction does not constrain the guards to be disjoint (mutually exclusive). Disjointedness is not required for correctness [11] because if multiple guards are triggered then arbitrarily choosing the body for any one suffices. Therefore, without loss of generality, the branches can be arbitrarily ordered (thus ensuring mutual exclusivity) in the output to get a valid imperative program.

Iterative lower bounded search Notice that Eq. (4) is non-standard, i.e., it is not an implication constraint like typical verification conditions, and we will elaborate on this in Section 4. For the case when a program verification tool is unable to handle such non-standard constraints, we need a technique for ensuring well-formedness of transitions without asserting Eq. (4).

We first assume that $\texttt{valid}(s_i)$ holds, and we will show in Section 4.3 the conditions under which it does. Then all we need to ensure well-formedness is that $\vee_i g_i$ is a tautology. Since the transitions of a `choose` statement represent independent execution paths, we can perform an iterative search for the guards g_i. We start by finding *any* satisfying guard (and corresponding transition)—which can even be `false`. We then iteratively ask for another guard (and transition) such that the space defined by the new guard is *not* entirely contained in the space defined by the disjunction of the guards already generated. If we ensure that at each step the newly discovered guard covers some more space that was not covered by earlier guards, then eventually the disjunction of all will be a tautology.

More formally, suppose n such calls result in the transition system $\{[]g_i \to s_i\}_{i=1..n}$, and $\vee_{i=1..n} g_i$ is not already a tautology. Then for the $n+1^{st}$ transition, we assert the constraint $\neg(g_{n+1} \Rightarrow (\vee_{i=1..n} g_i))$. This constraint ensures that g_{n+1} will cover some space not covered by $\vee_{i=1..n} g_i$. We repeat until $\vee_i g_i$ holds. This iterative search for the transitions also eliminates the need to guess the value of n.

Well-formedness of an Expanded Scaffold We constrain the well-formedness of each transition system in the expanded scaffold $exp = \texttt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\texttt{prf}}}(R_{\texttt{flow}})$ using Eq. (4).

$$\texttt{WellFormCond}(exp) = \bigwedge_{\texttt{choose}\ \{[]g_i \to s_i\}_i\ \in \texttt{cond}(exp)} \texttt{WellFormTS}(\{[]g_i \to s_i\}_i) \qquad (5)$$

where $\texttt{cond}(exp)$ recursively examines the expanded scaffold exp and returns the set of all `choose` statements in it.

EXAMPLE 4. *For the expanded scaffold in Example 2, since each acyclic fragment only contains one guarded transition, the well-formedness constraints are simple and state that each of $g_1, g_2, g_3 = \texttt{true}$ and $\texttt{valid}(s_1) \wedge \texttt{valid}(s_2) \wedge \texttt{valid}(s_3)$ holds.*

3.5 Encoding Progress: Ranking functions

Until now our encoding has focused on safety conditions that, by themselves, only ensure partial correctness but not termination. Next, we add progress constraints to ensure that the synthesized programs terminate.

To encode progress for a loop $l = \texttt{while}^{\tau,\varphi_l}(g)\ \texttt{do}\ \{\vec{p}\}$, we assert the existence of a *ranking function* as an unknown (numerical) expression φ_l that is lower bounded and decreases with each iteration of the loop. Because φ_l is an *unknown expression* it is difficult to encode directly that it decreases. Therefore, we introduce a tracking variable r_l, such that $r_l = \varphi_l$. We use r_l to remember the value of the ranking function at the head of the loop, and because it is a proof variable no assignments to it can appear in the body of the loop. On the other hand, φ_l changes due to the loop body, and at the end of the iteration we can then check if the new value is strictly less than the old value, i.e., $r_l > \varphi_l$. Without loss of generality, we pick a lower bound of 0 for the tracking variable and conservatively assume that the termination argument is implied by the loop invariant τ, i.e, $\tau \Rightarrow r_l \geq 0$.

Now that we have asserted the lower bound, what remains is to assert that φ_l decreases in each iteration. Assume, for the time being, that the body does not contain any nested loops. Then we can capture the effect of the loop body using `PathC` as defined earlier, with precondition $\tau \wedge g$ and postcondition $r_l > \varphi$. Then, the progress constraint for loop l without any inner loop is:

$$\texttt{prog}(l) \doteq (r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge \texttt{PathC}(\tau \wedge g, \vec{p}, r_l > \varphi_l))$$

Using the above definition of progress we define the progress constraint for the entire expanded scaffold $exp = \texttt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\texttt{prf}}}(R_{\texttt{flow}})$:

$$\texttt{RankCond}(exp) = \bigwedge_{l \in \texttt{loops}(exp)} \texttt{prog}(l) \qquad (6)$$

where $\texttt{loops}(exp)$ recursively examines the expanded scaffold exp and returns the set of all loops in it.

EXAMPLE 5. *In the expanded scaffold of Example 2 there is only one loop, whose ranking function we denote by φ_l and with tracker r_l. Then we generate the following progress constraint:*

$$r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge (\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow r'_l > \varphi'_l)$$

To relax the assumption we made earlier about no nesting of loops, we need a simple modification to the progress constraint $\texttt{prog}(l)$. Instead of considering the effect of the entire body $\vec{p}$ (which now contains inner loops), we instead consider the fragment $\texttt{end}(l)$ *after* the last inner loop in $\vec{p}$. Also, let $\tau_{\texttt{end}}$ denote the invariant for the last inner loop. Then, the progress constraint for loop l is:

$$\texttt{prog}(l) \doteq r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge \texttt{PathC}(\tau_{\texttt{end}}, \texttt{end}(l), r_l > \varphi_l)$$

Notice that because the loop invariants are not decided a priori, i.e., we are *not* doing program extraction, we may assert that the invariants should be strong enough to satisfy the progress constraints. Specifically, we have imposed the requirement that the intermediate loop invariants carry enough information such that it suffices to consider only the last loop invariant $\tau_{\texttt{end}}$ in the assertion.

3.6 Entire Synthesis Condition

Finally, we combine the constraints from the preceding sections to yield the entire synthesis condition for an expanded scaffold $exp = \texttt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\texttt{prf}}}(R_{\texttt{flow}})$. The constraint $\texttt{SafetyCond}(exp, \mathcal{F})$ (Eq. 3) ensures partial correctness of the program with respect to the functional specification. The constraint $\texttt{WellFormCond}(exp)$ (Eq. 5) restricts the space to programs with valid control-flow. The constraint $\texttt{RankCond}(exp)$ (Eq. 6) restricts the space to terminating programs. The entire synthesis condition is given by

$$sc = \texttt{SafetyCond}(exp, \mathcal{F}) \wedge \texttt{WellFormCond}(exp) \wedge \texttt{RankCond}(exp)$$

Notice that we have omitted the implicit quantifiers for the sake of clarity. The actual form is $\exists U \forall V : sc$. The set V denotes the program variables, $\vec{v_{\texttt{in}}} \cup \vec{v_{\texttt{out}}} \cup T \cup L$ where T is the set of temporaries (additional local variables) as specified by the scaffold and L is the set of iteration counters and ranking function trackers. Also, U is the set of all unknowns of various types instantiated during the expansion of scaffold. This includes unknowns for the invariants τ, the guards g and the statements s.

EXAMPLE 6. *Accumulating the partial correctness, well-formedness of branching and progress constraints we get the following synthesis condition (where we have removed the trivial guards g_1, g_2, g_3 as discussed in Example 4):*

$$\begin{array}{rcll} x \geq 1 \wedge s_1 & \Rightarrow & \tau' & \wedge \\ \tau \wedge g_0 \wedge s_2 & \Rightarrow & \tau' & \wedge \\ \tau \wedge \neg g_0 \wedge s_3 & \Rightarrow & (i'-1)^2 \leq x' \wedge x' < i'^2 & \wedge \\ \multicolumn{3}{l}{\texttt{valid}(s_1) \wedge \texttt{valid}(s_2) \wedge \texttt{valid}(s_3)} & \wedge \\ \multicolumn{4}{l}{r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge (\tau \wedge g_0 \wedge s_2 \Rightarrow r'_l > \varphi'_l)} \end{array}$$

Here is a valid solution to the above constraints:

$$\begin{array}{rl} \tau: & v = i^2 \wedge x \geq (i-1)^2 \wedge i \geq 1 \\ g_0: & v \leq x \\ \varphi_l: & x - (i-1)^2 \\ s_1: & v' = 1 \wedge i' = 1 \wedge x' = x \wedge r'_l = r_l \\ s_2: & v' = v + 2i + 1 \wedge i' = i + 1 \wedge x' = x \wedge r'_l = r_l \\ s_3: & v' = v \wedge i' = i \wedge x' = x \wedge r'_l = r_l \end{array} \qquad (7)$$

Notice how each of the unknowns satisfy their domain constraints, i.e., τ is from FOL over quadratic relations, φ_l is a quadratic expression, s_1, s_2, s_2 are conjunctions of linear equalities and g_0 is from quantifier-free FOL over linear relations. In the next section we show how such solutions can be computed using existing tools.

```
Input: Scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$, maximum transitions $n$,
       proof domain $D_{\texttt{prf}}$
Output: Executable program or FAIL
begin
    $exp$ := $\texttt{Expand}^{n,D_{\texttt{prf}}}_{\mathcal{D},\mathcal{R}}(R_{\texttt{flow}})$;
    $sc$ := $\texttt{SafetyCond}(exp, \mathcal{F}) \wedge$
            $\texttt{WellFormCond}(exp) \wedge$
            $\texttt{RankCond}(exp)$;
    $\pi$ := $\texttt{Solver}(sc)$;
    if $(\texttt{unsat}(\pi))$ then
        return FAIL;
    return $\texttt{Exe}^{\pi}(exp)$;
end
```

Algorithm 1: The entire synthesis algorithm.

Under the assumption [13] that every loop with a pre- and post-condition has an inductive proof of correctness, and every terminating loop has a ranking function, and that the domains chosen are expressive enough, we can prove that the synthesis conditions, for the case of non-iterative upper bounded well-formedness, model the program faithfully:

THEOREM 1 (Soundness and Completeness). *The synthesis conditions corresponding to a scaffold are satisfiable iff there exists a program (with a maximum of n transitions in each acyclic fragment where n is the parameter to the expansion) that is valid with respect to the scaffold.*

Additionally, for the alternative approach to discovering guards (Section 3.4), we can prove soundness and relative completeness:

THEOREM 2 (Soundness and Relative Completeness). (a) Soundness: *If there exists a program that is valid with respect to the scaffold then at each step of the iteration the synthesis conditions generated are satisfiable.* (b) Relative completeness: *If the iterative search for guards terminates then it finds a program that is valid with respect to the scaffold.*

4. Solving Synthesis Conditions

In this section we describe how the synthesis conditions for an expanded scaffold can be solved using fixed-point computation tools (program verifiers).

Suppose we have a procedure $\texttt{Solver}(sc)$ that can generate solutions to a synthesis condition sc. Algorithm 1 is our synthesis algorithm, which expands the given scaffold to exp, constructs synthesis conditions sc, uses $\texttt{Solver}(sc)$ to generate a solution π to the unknowns that appear in the constraints and finally generates concrete programs (whose acyclic fragments are from the language IML from Section 2) using the postprocessor $\texttt{Exe}^{\pi}(exp)$.

The concretization function $\texttt{Exe}^{\pi}(exp)$ takes the solution π computed by $\texttt{Solver}(sc)$ and the expanded scaffold exp, and outputs a program whose acyclic fragments are from the language IML. The function defines a concretization for each statement in TSL and annotates each loop with its loop invariant and ranking function:

$$\begin{array}{rcl} \texttt{Exe}^{\pi}(p;\vec{p}) & = & \texttt{Exe}^{\pi}(p);\texttt{Exe}^{\pi}(\vec{p}) \\ \texttt{Exe}^{\pi}(\texttt{while}^{\tau,\varphi}(g)\ \texttt{do}\ \{\vec{p}\}) & = & \\ \multicolumn{3}{r}{\texttt{while}^{\pi(\tau),\pi(\varphi)}(\pi(g))\ \{\ \texttt{Exe}^{\pi}(\vec{p})\ \}} \\ \texttt{Exe}^{\pi}(\texttt{choose}\ \{[]g \rightarrow s\}) & = & \\ \multicolumn{3}{c}{\texttt{if}\ (\pi(g))\ \{\texttt{Stmt}(\pi(s))\}\ \texttt{else}\ \{\texttt{skip}\}} \\ \texttt{Exe}^{\pi}(\texttt{choose}\ \{[]g_i \rightarrow s_i\}_{i=1..n}) & = & \qquad \text{(where } n > 1) \\ \multicolumn{3}{c}{\texttt{if}\ (\pi(g_1))\ \{\texttt{Stmt}(\pi(s_1))\}} \\ \multicolumn{3}{c}{\texttt{else}\ \{\texttt{Exe}^{\pi}(\texttt{choose}\ \{[]g_i \rightarrow s_i\}_{i=2..n})\}} \end{array}$$

where π maps each s to a conjunction of equalities and the concretization function $\texttt{Stmt}(s)$ expands the equality predicates to their corresponding state updates:

$$\texttt{Stmt}(\bigwedge_{i=1..n} x_i = e_i) \doteq \begin{array}{l}(t_1 := e_1; ..; t_n := e_n);\\ \qquad\qquad (x_1 := t_1; ..; x_n := t_n)\end{array}$$

The above is a simple translation that uses additional fresh temporary variables $t_1 \,..\, t_n$ to simulate parallel assignment. Alternatively, one can use data dependency analysis to generate code that uses fewer temporary variables.

4.1 Basic Requirement for $\texttt{Solver}(sc)$

Our objective is to use verification tools to implement $\texttt{Solver}(sc)$, but we realize that not all tools are powerful enough. For use as a solver for synthesis conditions, verification tools require certain properties.

Let us first recall [32] the notion of the polarity of unknowns in a formula. Let ϕ be a FOL formula with unknowns whose occurrences are unique. Notice that all the constraints we generate have unique occurrences as we rename appropriately. We can categorize unknowns as either *positive* or *negative* such that strengthening (weakening) the positive (negative) unknowns makes ϕ stronger. Structurally, the nesting depth under negation—in the boolean formula written using the basic operators $(\vee, \wedge, \neg, \exists, \forall)$—defines whether an unknown is positive (even depth) or negative (odd depth). For example, the formula $(a \vee \neg b) \wedge \neg(\neg c \vee d)$ has positive unknowns $\{a, c\}$ and negative unknowns $\{b, d\}$.

In program verification we infer loop invariants given verification conditions with known program statements. Let us reconsider the verification condition in Eq. (1) with known program statements and guards. Notice that the constraints can be categorized into three forms, $\tau \wedge f_1 \Rightarrow \tau'$, $\tau \wedge f_2 \Rightarrow f_3$ and $f_4 \Rightarrow \tau'$, where f_i's denote known formulae. Also, observe that these three are the only forms in which constraints in verification conditions can occur. From these, we can see that the verification conditions contain at most one positive and one negative unknown (using the disjunctive translation of implication), depending on whether the corresponding path ends or starts at an invariant. Program verification tools implementing typical fixed-point computation algorithms are specialized to work solely with constraints with one positive and one negative unknown because there is no need to be more general.

In fact, traditional iterative fixed-point computation is even more specialized in that it requires support for either just one positive unknown or just one negative unknown. Traditional verifiers work either in a forward (computing least fixed-point) or backwards (computing greatest-fixed point) direction starting with the approximation $\bot$ or $\top$, respectively, and iteratively refining it.

A backwards iterative data flow analyzer always instantiates the positive unknown to the current approximation and uses the resulting constraint (with only one negative unknown) to improve the approximation. For example, suppose the current approximation to the invariant τ is f_5, then a backwards analyzer may instantiate τ' in the constraint $\tau \wedge f_1 \Rightarrow \tau'$ to get the formula $\tau \wedge f_1 \Rightarrow f_5'$ (with one negative unknown τ). It will then use the formula to improve the approximation by computing a new value for τ that makes this formula satisfiable.

Similarly, a typical forwards iterative data flow analyzer instantiates the negative unknown to the current approximation and uses the resulting constraint (with only one positive unknown) to improve the approximation. For example, suppose the current approximation to the invariant τ is f_6, then a forwards analyzer may instantiate τ in the constraint $\tau \wedge f_1 \Rightarrow \tau'$ to get the formula $f_6 \wedge f_1 \Rightarrow \tau'$ (with one positive unknown τ). It will then use the formula to improve the approximation by computing a new value for τ' that makes this formula satisfiable.

In contrast, let us consider the components (from Section 3) of the synthesis condition. The component $\texttt{SafetyCond}(exp)$ (Eq. (3)), in addition to the unknowns due to the invariants τ, contains unknowns for the program guards g and program statements s. These unknowns appear exclusively as negative unknowns, and there can be multiple such unknowns in each constraint. For example, in Eq. (1), the guards and statement unknowns appear as negative unknowns. On the other hand, the component $\texttt{WellFormCond}(exp)$ (Eq. (5)) contains the well-formedness condition on the guards $\vee_i g_i$ that is a constraint with multiple positive unknowns. Therefore we need a verifier that satisfies the following.

REQUIREMENT 1. *Support for multiple positive and multiple negative unknowns.*

Notice this requirement is more general than that supported by typical verifiers we discussed above.

Now consider, an example safety constraint such as $\tau \wedge g \wedge s \Rightarrow \tau'$ with unknowns τ, g and s, that can be rewritten as $\tau \Rightarrow \tau' \vee \neg g \vee \neg s$. Also, let us rewrite an example well-formedness constraint $\vee g_i$ as $\texttt{true} \Rightarrow \vee g_i$. This view presents an alternative explanation for Requirement 1 in that we need a tool that can infer the right case split, which in most cases would not be unique and would require maintaining multiple orthogonal solutions. Intuitively, this is related to a tool's ability to infer disjunctive facts.

In the above we implicitly assumed the invariant to be a conjunction of predicates. In the general case, we may wish to infer more expressive (disjunctive) invariants, e.g., of the form $u_1 \Rightarrow u_2$ or $\forall k : u_3 \Rightarrow u_4$, where u_i's are unknowns. In this case, multiple negative and positive unknowns appear even in the verification condition and therefore the verification tool must satisfy Requirement 1, which matches the intuition that disjunctive inference is required.

4.2 Constraint-based Verifiers as $\texttt{Solver}(sc)$

Constraint-based fixed-point computation is a relatively recent approach to program verification that has been successfully used for difficult analyses [32]. In previous work, we designed efficient constraint-based verification tools for two popular domains, predicate abstraction [32, 20] and linear arithmetic [21]. The tools for both domains satisfy Requirement 1.

Constraint-based verification tools reduce a verification condition vc (with invariant unknowns) to a boolean constraint $\psi(vc)$ such that a satisfying solution to the boolean constraint corresponds to valid invariants. The property they ensure is the following:

PROPERTY 1. *The boolean constraint $\psi(vc)$ is satisfiable iff there exists a fixed-point solution for the unknowns corresponding to the invariants.*

The reduction can also be applied to synthesis condition sc to get boolean constraints $\psi(sc)$ and a similar property holds. That is, the boolean constraint $\psi(sc)$ is satisfiable iff there exist statements, guards and invariants that satisfy the synthesis condition.

4.3 Iterative Verifiers as $\texttt{Solver}(sc)$

Let us now consider the case where the verification tool cannot handle non-standard constraints, such as Eq. (4). This is the case for typical iterative program verification tools that compute increasingly better approximations to invariants. We show that despite this lack of expressivity it is still possible to solve synthesis conditions as long as the tool satisfies an additional requirement.

The only non-implication constraint in the synthesis condition sc is $\texttt{WellFormCond}(sc)$. In Section 3.4, we discussed how an iterative lower-bounded search can discover the transitions $\{[]g_i \rightarrow s_i\}_i$ without asserting Eq. (5). There we had left the question of ensuring $\texttt{valid}(s_i)$ unanswered. Consider now the case where a valid solution g_i, s_i exists (i.e., s_i is not `false` or that $\texttt{valid}(s_i)$ holds) that satisfies the constraint set. As an instance, in Example 6, we have a synthesis condition for which a valid solution exists as shown by Eq. (7). Notice that this solution is *strictly* weaker than another solution that assigns identical values to other unknowns but assigns `false` to any of s_2, s_2 or s_3. In fact, we can observe that if the tool only generates maximally weak solutions then between these two solutions (which are comparable as we saw), it will always pick the one in which it does not assign `false` to statement unknowns. Therefore, it will always generate s_i such that $\texttt{valid}(s_i)$ holds unless no such s_i exists. Therefore, if the program verification tool satisfies the following requirement, then we can omit Eq. (5) from the synthesis condition and still solve it using the tool.

REQUIREMENT 2. *Solutions are maximally weak.*

This requirement corresponds to the tool's ability to compute weakest preconditions. The typical approach to weakest preconditions (greatest fixed-point) computation propagates facts backwards, but this is considered difficult and therefore not many tools exist that do this. However, although traditional iterative data flow verifiers fail to meet Requirements 1 and 2, there do exist some iterative tools [32] that compute maximally weak solutions and therefore satisfy the requirements.

We have argued that maximally weak solutions *for statement unknowns* s_i ensure $\texttt{valid}(s_i)$, but this comes at the cost of degraded performance because maximally weak solutions are generated for guard and invariant unknowns too. We require maximally weak solutions only for the statement unknowns, while for synthesis we are interested in *any* solution to the guard and invariant unknowns that satisfy the synthesis condition. In our trials, the constraint-based scheme [21, 32] (which computes any fixed-point in the lattice rather than the greatest fixed-point) outperformed the iterative scheme [32]. In fact, our tool based on iterative approximations does not terminate for most benchmarks, and we therefore perform the experiments using our constraint-based tool.

5. Experimental Case Studies

To evaluate our approach, we synthesized examples in three categories: First, easy to specify but tricky to program *arithmetic* programs; second, *sorting* programs, which all have the same specification but yield different sorting strategies depending on the resource constraints; third, *dynamic programming* programs for which naive solutions yield exponential runtimes, but which can be computed in polytime by suitable memoization.

5.1 Implementation

We augmented our constraint-based verification tools from prior work to build more powerful verifiers that we use as solvers for synthesis conditions. In this section, we summarize the capabilities of these tools and our extensions to them. The description here is necessarily brief due to lack of space, and more details can be found in the companion technical report [35]. We also describe a technique we use to simplify user input by expanding flowgraphs to be more expressive as required sometimes.

Verification Tools Our synthesis technique relies on an underlying program verification tool. For this work, we used tools that are part of the $\texttt{VS}^3$ project [33]. We used two tools: an arithmetic verification tool [21], which we call $\texttt{VS}^3_{\texttt{LIA}}$ here; and a predicate abstraction verification tool [32, 34, 20], which we call $\texttt{VS}^3_{\texttt{PA}}$ here.

Capabilities Both verification tools $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$ are based on the idea of reducing the problem of invariant generation to satisfiability solving. Each tool takes as input a C program (annotated with assertions, typically the postcondition; and assumptions, typically the precondition) and hints about the form of the invariants. The tool then generates invariants that suffice to prove the assertions. If the tool fails to generate the invariants, then either the assertions in the program do not hold or no invariants exist that are instantiations of the given template form.

- $\texttt{VS}^3_{\texttt{LIA}}$ works over the theory of linear arithmetic and discovers (quantifier-free) invariants in DNF form with linear inequalities over program variables as the atomic facts. As hints, it expects three integer parameters, $\texttt{max}_{dsj}$, $\texttt{max}_{cnj}$, $\texttt{max}_{bv}$. The parameters $\texttt{max}_{dsj}$ and $\texttt{max}_{cnj}$ limit the maximum number of disjuncts and conjuncts (in each disjunct) in the invariants. Additionally, $\texttt{max}_{bv}$ is a integer pair (b_1, b_2) for the size in bits of invariant coefficients b_1 and intermediate computations b_2. None of our synthesized benchmarks required disjunctive invariants ($\texttt{max}_{dsj} = 1$), and we choose appropriate values for $\texttt{max}_{cnj}$ and $\texttt{max}_{bv}$ for different benchmarks. For example, suppose x, y, z are the program variables and $x \leq y \wedge z = x + 1$ is the unknown program invariant. The user will specify some value v_1 for $\texttt{max}_{cnj}$ and (v_2, v_3) for $\texttt{max}_{bv}$. $\texttt{VS}^3_{\texttt{LIA}}$ will setup and solve constraints for a conjunctive invariant with atoms of the form $c_0 + c_1x + c_2y + c_3z \geq 0$. It will search for solutions to each coefficient c_i assumed to have v_2 bits and use v_3 bits for any intermediate values in the constraints. In this example, any $v_1 \geq 3$ and any $v_2 \geq 2$ (one bit for the sign) will work, and we can choose a large enough v_3 to avoid overflows.
- $\texttt{VS}^3_{\texttt{PA}}$ works over a combination of the theories of equality with uninterpreted functions, arrays, and linear arithmetic and discovers (possibly) quantified invariants. $\texttt{VS}^3_{\texttt{PA}}$ expects the boolean structure of the invariants, i.e., quantification and disjunctions, to be made explicit as a template. As hints, it expects a boolean template T (with holes for conjunctive facts) and a set of predicates P. $\texttt{VS}^3_{\texttt{PA}}$ infers the subset of predicates from P—the atoms of the conjunct—that populate the holes in T. For example, if $x \geq 0 \wedge \forall k : 0 \leq k \leq x \Rightarrow A[k] \leq A[k+1]$ is the invariant, then $\texttt{VS}^3_{\texttt{PA}}$ would discover it when run with any T that is at least $([-] \wedge \forall k : [-] \Rightarrow [-])$ and any P that is a superset of $\{x \geq 0, k \geq 0, x \geq k, A[k+1] \geq A[k]\}$, where $[-]$ denotes a conjunctive hole.

Extensions These tools are powerful and can infer expressive invariants such as those requiring quantification and disjunction, but for some of the benchmarks, the reasoning required was beyond even their capabilities. We therefore extended the base verifiers with the following features.

- *Quadratic expressions for arithmetic* For handling quadratic expressions in the proofs, we implemented a sound but incomplete technique that renames quadratic expressions to fresh variables and then uses linear arithmetic reasoning, already built into $\texttt{VS}^3_{\texttt{LIA}}$. This suffices for most of our benchmarks, except when linear relations need to be lifted to quadratic relations, e.g., $a \geq b \geq 0 \Rightarrow a^2 \geq b^2 \geq 0$. This happens in one isolated step in the integral square root binary search case, which we circumvent by explicitly encoding an assumption. We call this augmented solver $\texttt{VS}^3_{\texttt{QA}}$.
- *Axiomatization* Proposals exist for extending verification tools with axioms for theories they do not natively support, e.g., the theory of reachability for lists [26]. We take such axiomatization a step further and allow the user to specify axioms over uninterpreted symbols that define computations. We implement this in $\texttt{VS}^3_{\texttt{PA}}$ to specify the meaning of dynamic programming programs, e.g., the definition of Fibonacci. We call this augmented solver $\texttt{VS}^3_{\texttt{AX}}$.

(a)

```
Strassens(int $a_{ij}, b_{ij}$) {
  $v_1$:=$(a_{11}$+$a_{22})(b_{11}$+$b_{22})$
  $v_2$:=$(a_{21}$+$a_{22})b_{11}$
  $v_3$:=$a_{11}(b_{12}$-$b_{22})$
  $v_4$:=$a_{22}(b_{21}$-$b_{11})$
  $v_5$:=$(a_{11}$+$a_{12})b_{22}$
  $v_6$:=$(a_{21}$-$a_{11})(b_{11}$+$b_{12})$
  $v_7$:=$(a_{12}$-$a_{22})(b_{21}$+$b_{22})$
  $c_{11}$:=$v_1$+$v_4$-$v_5$+$v_7$
  $c_{12}$:=$v_3$+$v_5$
  $c_{21}$:=$v_2$+$v_4$
  $c_{22}$:=$v_1$+$v_3$-$v_2$+$v_6$
  return $c_{ij}$;
}
```

(c)

```
Fib(int $n$) {
  $v_1$:=0;$v_1$:=1;$i_1$:=0;
  while$^{\tau,\varphi}$($i_1 \leq n$)
    | $v_1$:=$v_1$+$v_2$;swap($v_1, v_2$);
    | $i_1$++;
  return $v_1$;
}
```

Ranking function φ:
$x - s$

Invariant τ:
$v_1 = \texttt{Fib}(i_1) \wedge v_2 = \texttt{Fib}(i_1+1)$

(b)

```
SelSort(int $A[], n$) {
  $i_1$:=0;
  while$^{\tau_1,\varphi_1}$($i_1 < n-1$)
    | $v_1$:=$i_1$;
    | $i_2$:=$i_1$+1;
    | while$^{\tau_2,\varphi_2}$($i_2 < n$)
    |   | if ($A[i_2] < A[v_1]$)
    |   |             $v_1$:=$i_2$;
    |   | $i_2$++;
    | swap($A[i_1], A[v_1]$);
    | $i_1$++;
  return $A$;
}
```

Ranking functions:
$\varphi_1 : n - i_1 - 2$
$\varphi_2 : n - i_2 - 1$

Invariant τ_1:
$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n$
$\quad \wedge k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]$

Invariant τ_1:
$i_1 < i_2 \wedge i_1 \leq v_1 < n$
$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n$
$\quad \wedge k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]$
$\forall k : i_1 \leq k < i_2 \wedge k \geq 0$
$\qquad \Rightarrow A[v_1] \leq A[k]$

Figure 2. Illustrative examples from each of the domains. (a) Arithmetic: Strassen's Matrix Multiplication (b) Sorting: Selection Sort (c) Dynamic Programming: Fibonacci. For simplifying the presentation, we omit degenerate conditional branches, i.e. true/false guards, We name the loop iteration counters $L = \{i_1, i_2, ..\}$ and the temporary stack variables $T = \{v_1, v_2, ..\}$.

Note that these extensions are to facilitate verification and not synthesis. The synthesis solver is exactly the same as the verification tool. Without the extensions most of our benchmarks cannot even be verified, and thus their verification can be seen as an independent contribution.

Flowgraphs with Init/Final Phases In practice a fair number of loops have characteristic *initialization* and *finalization* phases that exhibit behavior different from the rest of the loop. In theory, verifiers should be able to infer loop invariants that capture such semantically different phases. However, this requires disjunctive reasoning, which is fairly expensive if at all supported by the verifier. Instead we use an alternate expansion $\overline{\texttt{Expand}}^n(T)$ that introduces acyclic fragments for the initialization and finalization if synthesis without them fails. For instance, for Example 1, the the user only needs to specify the flowgraph $*(\circ)$ instead of the more complicated $\circ;*(\circ);\circ$. Except for the expansion of loops, $\overline{\texttt{Expand}}^n(T)$ expands all other statements exactly like $\texttt{Expand}^n(T)$ does. For loops, it builds an initialization and finalization phase as follows.

$$\begin{array}{lll}\overline{\texttt{Expand}}^n(*(T)) = & \texttt{Expand}^n(\circ); & \rightarrow \textit{Added initialization} \\ & \texttt{while}^{\tau}\ (g)\ \{\overline{\texttt{Expand}}^n(T);\} & \\ & \texttt{Expand}^n(\circ); & \rightarrow \textit{Added finalization}\end{array}$$

5.2 Algorithms that use arithmetic

For this category, we pick $D_{\texttt{prf}}$ to be quadratic arithmetic and use as our solver the $\texttt{VS}^3_{\texttt{QA}}$ tool. We chose a set of arithmetic benchmarks with simple-to-state functional specifications but each containing some tricky insight that human programmers may miss.

Swapping without Temporaries Consider a program that swaps two integer-valued variables *without* using a temporary. The precondition and postcondition to the program are specified as $F_{\texttt{post}} \doteq (x = c_2 \wedge y = c_1)$ and $F_{\texttt{pre}} \doteq (x = c_1 \wedge y = c_2)$, respectively. We specify an acyclic flowgraph template $R_{\texttt{flow}} \doteq \circ$ and a computation template $R_{\texttt{comp}} \doteq \emptyset$ that imposes no constraints. To ensure that no temporaries are used we specify $R_{\texttt{stack}} \doteq \emptyset$. The synthesizer generates various versions of the program, e.g., one being

$$\texttt{Swap(int}\ x, y\texttt{)}\{x := x + y; y := x - y; x := x - y;\}$$

Strassen's 2×2 ***Matrix Multiplication*** Consider Strassen's matrix multiplication, which computes the product of two $n \times n$ matrices in $\Theta(n^{2.81})$ time instead of $\Theta(n^3)$. The key to this algorithm is an acyclic fragment that computes the product of two 2×2 input matrices $\{a_{ij}, b_{ij}\}_{i,j=1,2}$ using 7 multiplications instead of the expected 8. Used recursively, this results in asymptotic savings. We do not attempt to synthesize the full matrix multiplication procedure because it contains no significant insight. Instead, we synthesize the crucial acyclic fragment, which is shown in Figure 2(a). Here the precondition $F_{\texttt{pre}}$ is true and the postcondition $F_{\texttt{post}}$ is the conjunction of four equalities as (over the outputs $\{c_{ij}\}_{i,j=1,2}$):

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

The synthesizer also generates many alternate versions that are functionally equivalent to Figure 2(a).

As a side note, we also attempted synthesis using 6 multiplications, which failed. This suggests that possibly no asymptotically faster solution exists using simple quadratic computations—theoretical results up to $n^{2.376}$ are known, but use products that cannot be easily be captured in the simple domains considered here.

Integral Square Root Consider computing the integral square root $\lfloor\sqrt{x}\rfloor$ of a positive number x using only linear or quadratic operations. The precondition is $F_{\texttt{pre}} \doteq x \geq 1$ and the postcondition, involving the output i, is $F_{\texttt{post}} \doteq (i-1)^2 \leq x < i^2$. We provide a single loop flowgraph template $R_{\texttt{flow}} \doteq *(\circ)$ and an empty computation template $R_{\texttt{comp}} \doteq \emptyset$. The synthesizer generates different programs depending on the domain constraints and the stack template:

- $R_{\texttt{stack}} \doteq \{(\texttt{int}, 0)\}$ and we allow quadratic expressions in $D_{\texttt{exp}}, D_{\texttt{grd}}$. The synthesized program does a sequential search downwards starting from $i = x$ by continuously recomputing and checking $(i-1)^2$ against x.
- $R_{\texttt{stack}} \doteq \{(\texttt{int}, 1)\}$ and we only allow linear expressions in $D_{\texttt{exp}}, D_{\texttt{grd}}$. The synthesized program does a sequential search but uses the additional local variable rather surprisingly to track the value of $(i-1)^2$ using only linear updates. The synthesized program is Example 1, from earlier.
- $R_{\texttt{stack}} \doteq \{(\texttt{int}, 2)\}$ and we allow quadratic expressions in $D_{\texttt{exp}}, D_{\texttt{grd}}$. The synthesized program does a binary search for the value of i and uses the two additional local variables to hold the low and high end of the binary search space. To restrict reasoning to linear arithmetic we model $m = \lfloor(s_1 + s_2)/2\rfloor$ as the assumption $s_1 \leq m \leq s_2$. Additionally, because of the incompleteness in the handling of quadratic expressions, our solver cannot derive $(s_2 + 1)^2 \leq s_1^2$ from $s_2 + 1 \leq s_1$. Thus we provide the quadratic inequality as another assumption.

Bresenham's Line Drawing Algorithm Consider Bresenham's line drawing algorithm, as we discussed in Section 1.1. For efficiency, the algorithm only uses linear updates, which are non-trivial to verify [16] or even understand (let alone discover from scratch).

We specify the precondition $F_{\texttt{pre}} \doteq 0 < Y \leq X$. The postcondition can be written as a quantified assertion outside the loop or as a quantifier-free assertion inside the loop, as mentioned in Section 1.1. We choose to annotate the flowgraph with the simpler quantifier-free $2|y - (Y/X)x| \leq 1$ at the loop header and additionally specify that the loop iterates over $x = 0..X$. This simplification in the implementation allowed us to use $\mathtt{VS}^3_{\mathtt{QA}}$, which only supports quantifier-free facts.

We specify a single loop flowgraph $R_{\mathtt{flow}} \doteq *(\circ)$ and empty stack and computation templates $R_{\mathtt{stack}} \doteq \emptyset$, $R_{\mathtt{comp}} \doteq \emptyset$. The synthesizer generates multiple versions, one of which is exactly as shown in Figure 1(b) and can be translated to the program in Figure 1(a).

5.3 Sorting Algorithms

For this category, we pick $D_{\mathtt{prf}}$ to be the theory supported by $\mathtt{VS}^3_{\mathtt{PA}}$, which we use as our solver. The current version of our tool works with a user-supplied set of predicates. We are working on predicate inference techniques—in the style of CEGAR-based model checkers [23]—but for now, we give the tool a candidate set of predicates.

The sortedness specification consists of the precondition $F_{\mathtt{pre}} \doteq \mathtt{true}$ and the postcondition $F_{\mathtt{post}} \doteq \forall k : 0 \le k < n \Rightarrow A[k] \le A[k+1]$. The full functional specification would also ensure that the output array is a permutation of the input, but verifying—and thus, synthesizing—the full specification is outside the capabilities of most tools today.

We therefore use a mechanism to limit the space of programs to desirable sorting algorithms, while still only using $F_{\mathtt{post}}$. We limit $D_{\mathtt{exp}}$ to those that only involve operations that maintain elements—for example, *swapping* elements or *moving* elements to unoccupied locations. Using this mechanism, we ensure that invalid algorithms (that replicate or lose array elements) are not considered.

Non-recursive sorting algorithms Consider comparison-based sorting programs that are composed of nested loops. We specify a flowgraph template $R_{\mathtt{flow}} \doteq *(*(\circ))$ and a computation template $R_{\mathtt{comp}}$ that limits the operations to swapping of array values.

- $R_{\mathtt{stack}} \doteq \emptyset$: The synthesizer produces two sorting programs that are valid with respect to the scaffold. One corresponds to Bubble Sort and the other is a non-standard version of Insertion Sort. The standard version of Insertion Sort uses a temporary variable to hold the inserted object. Since we do not provide a temporary variable, the synthesized program moves the inserted element by swapping it with its neighbor, while still performing operations similar to Insertion Sort.
- $R_{\mathtt{stack}} \doteq \{(\mathtt{int}, 1)\}$: The synthesizer produces another sorting program that uses the temporary variable to hold an array index. This program corresponds to Selection Sort and is shown in Figure 2(b). Notice the non-trivial invariants and ranking functions that are synthesized alongside for each of the loops.

Recursive divide-and-conquer sorting Consider comparison-based sorting programs that use recursion. We make a few simple modifications to the system to specify recursive programs. First, we introduce a terminal string "$\circledast$" to the flowgraph template language (Eq. 2), representing a recursive call. Let $(F_{\mathtt{pre}}(\vec{v_{\mathtt{in}}}), F_{\mathtt{post}}(\vec{v_{\mathtt{out}}}))$ denote the functional specification. Then we augment the expansion (Section 3.2) to handle the new flowgraph string as follows:

$$\mathtt{Expand}^n(\circledast) \quad = \quad \mathtt{choose}\{[]\mathtt{true} \rightarrow s_{\mathtt{recur}}\}$$

where $s_{\mathtt{recur}} = s_{\mathtt{args}} \wedge (F_{\mathtt{pre}}(\vec{v_{\mathtt{in}}}') \Rightarrow F_{\mathtt{post}}(\vec{v_{\mathtt{out}}}'')) \wedge s_{\mathtt{ret}}$ sets values to the arguments of the recursive call (using $s_{\mathtt{args}}$), assumes the effect of the recursive call (using $F_{\mathtt{pre}}(\vec{v_{\mathtt{in}}}') \Rightarrow F_{\mathtt{post}}(\vec{v_{\mathtt{out}}}'')$, with the input arguments renamed to $\vec{v_{\mathtt{in}}}'$ and the return variables renamed to $\vec{v_{\mathtt{out}}}''$) and lastly, outputs the returned values into program variables (using $s_{\mathtt{ret}}$). The statements $s_{\mathtt{args}}, s_{\mathtt{ret}}$ take the form:

$$\begin{array}{rcll} s_{\mathtt{args}} & = & \bigwedge_i x_i = e_i & \text{where } x_i \in \vec{v_{\mathtt{in}}}', e_i \in D_{\mathtt{exp}}|_{\mathtt{Vars}} \\ s_{\mathtt{ret}} & = & \bigwedge_i x_i = e_i & \text{where } x_i \in \mathtt{Vars}, e_i \in D_{\mathtt{exp}}|_{\vec{v_{\mathtt{out}}}''} \end{array}$$

Here $\mathtt{Vars}$ denote the variables of the procedure (the input, output and local stack variables). We also tweak the statement concretization function (Section 4) to output a recursive call statement $\mathtt{rec}$:

$$\mathtt{Stmt}(F_{\mathtt{pre}}(\vec{v_{\mathtt{in}}}') \Rightarrow F_{\mathtt{post}}(\vec{v_{\mathtt{out}}}'')) = \vec{v_{\mathtt{out}}}'' := \mathtt{rec}(\vec{v_{\mathtt{in}}}')$$

We specify a computation template that allows only swapping or moving of elements. We then try different values of the flowgraph and stack templates:

- $R_{\mathtt{flow}} \doteq \circledast;\circledast;\circ$ (two recursive calls followed by an acyclic fragment) and $R_{\mathtt{stack}} \doteq \emptyset$: The synthesizer produces a program that recursively sorts subparts and then combines the results. This corresponds to Merge Sort.
- $R_{\mathtt{flow}} \doteq \circ;\circledast;\circledast$ (an acyclic fragment followed by two recursive calls) and $R_{\mathtt{stack}} \doteq \{(\mathtt{int}, 1)\}$: The synthesizer produces a program that partitions the elements and then recursively sorts the subparts. This corresponds to Quick Sort.

5.4 Dynamic Programming Algorithms

For this category, we pick $D_{\mathtt{prf}}$ to be the theory supported by $\mathtt{VS}^3_{\mathtt{AX}}$, which we use as our solver. As in the previous section, since our tool does not currently infer predicates, we give it a candidate set. We choose all the textbook dynamic programming examples [8] and attempt to synthesize them from their functional specifications.

The first hurdle (even for verification) for these algorithms is that the meaning of the computation is not easily specified. To address this issue, we need support for axioms, which are typically recursive definitions.

Definitional Axioms The verification tool allows the user to define the meaning of a computation as an uninterpreted symbol, with (recursive) quantified facts defining the semantics of the symbol axiomatically. For example, the semantics of Fibonacci are defined in terms of the symbol $\mathtt{Fib}$ and the three axioms:

$$\begin{array}{l} \mathtt{Fib}(0) = 0 \wedge \mathtt{Fib}(1) = 1 \\ \forall k : k \ge 0 \Rightarrow \mathtt{Fib}(k+2) = \mathtt{Fib}(k+1) + \mathtt{Fib}(k) \end{array}$$

The tool passes the given symbol and its definitional axioms to the underlying theorem prover (Z3 [10]), which assumes the axioms before every theorem proving query. With this interpretation of the symbol $\mathtt{Fib}$ known to the theorem prover, the verifier can now pose theorem proving queries involving the symbol $\mathtt{Fib}$. For instance, the iterative program for Fibonacci maintains an invariant of the form $x = \mathtt{Fib}(i)$, which the verifier can now infer. This allows the tool to verify dynamic programming programs that are typically iterative, relating them to the recursive definitional axioms.

Even with verification in place, automatic synthesis of these programs involves three non-trivial tasks for the synthesizer. First, the synthesizer needs to automatically discover a strategy for translating the recursion (in the functional specification) to *non-recursive iteration* (for the actual computation). The functional specifications do not contain this information, e.g., in the specification for Fibonacci above, the iteration strategy for the computation is not evident. Second, the synthesizer needs to take the (non-directional) equalities in the specifications and *impose directionality* such that elements are computed in the right order. For example, for Fibonacci the synthesizer needs to automatically discover that $\mathtt{Fib}(k)$ and $\mathtt{Fib}(k+1)$ should be computed before $\mathtt{Fib}(k+2)$. Third, the synthesizer needs to discover an *efficient memoization* strategy for only those results needed for future computations, to fit the computation in the space provided—which is one of the benefits of dynamic programming algorithms. For example, Fibonacci can be computed using only two additional memory locations by suitable memoization. Fortunately, just by specifying the resource constraints and using our proof-theoretic approach the synthesizer is able to perform these tasks and synthesize dynamic programming algorithms from their recursive functional specifications.

Also, as in the case of sorting, we want to disallow completely arbitrary computations. In sorting, we could uniformly restrict the expression language to only swap and move operations. For dynamic programming, the specification of the operations is problem-specific. For instance, for shortest path, we only want to allow the path matrix updates that correspond to valid paths, e.g., disallow arbitrary multiplication of path weights. $R_{\mathtt{comp}}$ specifies these constraints by only permitting updates through certain predicates.

Dynamic programming solutions typically have an initialization phase (init-loop) and then a phase (work-loop) that fills the appropriate entries in the table. Therefore, we chose a $R_{\mathtt{flow}}$ with an init-loop ($*(\circ)$) followed by a work-loop.

By specifying a flowgraph template $R_{\mathtt{flow}} \doteq *(\circ);*(\circ)$ and a stack template with no additional variables (except for the case of Fibonacci, where the synthesizer required $R_{\mathtt{stack}} \doteq \{(\mathtt{int}, 2)\}$), we were able to synthesize the following four examples:

Fibonacci Consider computing the nth Fibonacci number from the functional specification as above. Our synthesizer generates a program that memoizes the solutions to the two subproblems $\mathtt{Fib}(i_1)$ and $\mathtt{Fib}(i_1 + 1)$ in the i_1th iteration. It maintains a sliding window for the two subproblems and stores their solutions in the two additional stack variables. The synthesized program along with its invariant and ranking function is shown in Figure 2(c).

Checkerboard Consider computing the least-cost path in a rectangular grid (with costs at each grid location), from the bottom row to the top row. The functional specification states the path cost for a grid location in terms of the path costs for possible previous locations (i.e., below left, below or below right). Our synthesizer generates a program that finds the minimum cost paths.

Longest Common Subsequence (LCS) Consider computing the longest common substring that appears in the same order in two given input strings (as arrays of characters). The recursive functional specification relates the cost of a substring against the cost of substrings with one fewer character. Our synthesizer generates a program for LCS.

Single Source Shortest Path Consider computing the least-cost path from a designated source to all other nodes where the weight of edges is given given as a cost function for each source and destination pair. The recursive functional specification states the cost structure for all nodes in terms of the cost structure of all nodes if one fewer hop is allowed. Our synthesizer generates a program for the single source shortest path problem.

For the following two examples, synthesis failed with the simpler work-loop, but we synthesize the examples by specifying a flowgraph template $*(\circ);*(*(\circ))$ and no additional stack variables:

All-pairs Shortest Path Consider computing all-pairs shortest paths using a recursive functional specification similar to the one we used for single source shortest path. Our synthesizer times out for this example. We therefore attempt synthesis by (i) specifying the acyclic fragments and synthesizing the guards, and (ii) specifying the guards and synthesizing the acyclic fragments. In each case, our synthesizer generates the other component, corresponding to Floyd-Warshall's algorithm.

Matrix Chain Multiply Consider computing the optimal way to multiply a matrix chain. Depending on the bracketing, the total number of multiplications varies. We wish to find the bracketing that minimizes the number of multiplications. E.g., if we use the simple n^3 multiplication for two matrices, then $A_{10\times100}B_{100\times1}C_{1\times50}$ can either takes 1,500 multiplications for $(AB)C$ or 55,000 multiplications for $A(BC)$. The functional specification defines the cost of multiplying a particular chain of matrices in terms of the cost of a chain with one fewer element. Our synthesizer generates a program that computes the optimal matrix bracketing.

Benchmark	$\mathtt{max}_{cnj}$	$\mathtt{max}_{bv}$	Assumes
Swap two	0	2, 6	0
Strassen's	0	2, 6	0
Sqrt (linear search)	4	2, 6	0
Sqrt (binary search)	3	2, 8	2
Bresenham's	6	2, 5	0

Table 1. Parameters used for synthesis using $\mathtt{VS}^3_{\mathtt{QA}}$. For each benchmark, we list the maximum number of conjuncts in any invariant ($\mathtt{max}_{cnj}$) and bit vector sizes ($\mathtt{max}_{bv}$, for constants and for bit-blasting), respectively. The last column lists any assumes we manually specified.

Benchmark	Number of		
	Defn. Axioms	Templates, Predicates	Annot. or Assumes
Bubble Sort	0	3, 17	1
Insertion Sort	0	3, 16	1
Selection Sort	0	3, 20	1
Merge Sort	0	4, 16	3
Quick Sort	0	3, 15	0
Fibonacci	3	1, 12	0
Checkerboard	5	2, 8	0
Longest Common Subseq.	6	2, 13	0
Matrix Chain Multiply	7	2, 18	0
Single-Src Shortest Path	3	3, 16	0
All-pairs Shortest Path	10	4, 19	0

Table 2. Parameters used for synthesis using $\mathtt{VS}^3_{\mathtt{AX}}$. For each benchmark, we list the number of definitional axioms required to specify the meaning of the computation. Additionally, we list the number of invariant templates and size of predicate set given to $\mathtt{VS}^3_{\mathtt{AX}}$. The last column lists any annotations or assumes we manually specified.

5.5 Performance

Parameters for $\mathtt{VS}^3_{\mathtt{QA}}$ ***and*** $\mathtt{VS}^3_{\mathtt{AX}}$ Table 1 lists the parameters required to run $\mathtt{VS}^3_{\mathtt{QA}}$ over our arithmetic benchmarks. The second column lists the maximum number of conjuncts ($\mathtt{max}_{cnj}$) expected in each invariant. The third column lists the bit vector sizes ($\mathtt{max}_{bv}$) used for invariant coefficients and intermediate values. The last column lists the assumptions we manually provided.

We guessed reasonable values for $\mathtt{max}_{cnj}$ for the programs with loops. We started with a small value and iteratively increased it if synthesis failed. For the bit-vector sizes, we choose two bits for the coefficients since we were not expecting large coefficients in the invariants. We then chose reasonable values (5–8) for the bit-vector size for intermediate computations. There was only one case, square root using binary search, where we had to specify a manual assertion, as discussed earlier. In all, little user effort was required.

Table 2 lists the parameters required to run $\mathtt{VS}^3_{\mathtt{AX}}$ over our sorting and dynamic programming benchmarks. The second column lists the number of definitional axioms required for specifying the meaning of the computation and are required even for verification. The third column lists the number of templates and predicates used. The last column lists the number of annotations or assumptions that were manually provided.

The templates contain conjunctive holes and explicit quantification and disjunction. For each benchmark, there is one quantifier-free, disjunction-less, template and the remaining are universally quantified with a form almost identical to the postcondition, and therefore easy to write out. All invariants are conjunctions of the templates with their holes instantiated with a subset (conjunction) of the predicate set. The predicates are atomic relations between linear and array expression over program variables and constants. We specify a suitable predicate set for each benchmark. We start with a candidate set and then iteratively added predicates when syn-

	Benchmark	Verif.	Synthesis	Ratio
Arith. ($\texttt{VS}^3_{\texttt{QA}}$)	Swap two	0.11	0.12	1.09
	Strassen's	0.11	4.98	45.27
	Sqrt (linear search)	0.84	9.96	11.86
	Sqrt (binary search)	0.63	1.83	2.90
	Bresenham's	166.54	9658.52	58.00
Sorting ($\texttt{VS}^3_{\texttt{PA}}$)	Bubble Sort	1.27	3.19	2.51
	Insertion Sort	2.49	5.41	2.17
	Selection Sort	23.77	164.57	6.92
	Merge Sort	18.86	50.00	2.65
	Quick Sort	1.74	160.57	92.28
Dynamic Prog. ($\texttt{VS}^3_{\texttt{AX}}$)	Fibonacci	0.37	5.90	15.95
	Checkerboard	0.39	0.96	2.46
	Longest Common Subseq.	0.53	14.23	26.85
	Matrix Chain Multiply	6.85	88.35	12.90
	Single-Src Shortest Path	46.58	124.01	2.66
	All-pairs Shortest Path[1]	112.28	(i) 226.71 (ii) 750.11	(i) 2.02 (ii) 6.68

Table 3. (a) Arithmetic (b) Sorting (c) Dynamic Programming. For each category, we indicate the tool used both for verification and synthesis. For each benchmark, we indicate the time in seconds to solve the verification conditions and the synthesis conditions, and the slowdown for synthesis compared to verification.

thesis failed. We needed at most 20 predicates, which were easily found for each benchmark. In the interest of space, we omit the exact templates and predicates used for each benchmark, but they can be found in the companion technical report [35]. For each of the sorting programs with a nested loop structure (Bubble, Insertion and Selection Sort), the outer loop invariant is redundant given the inner loop invariant. Therefore, we tag the outer loop—the one annotation each case—and the solver does not to attempt to generate its invariant, improving performance. For Merge Sort, synthesis fails unless we reduce the search space by manually specifying three simple quantifier-free atomic facts of the invariant. In all, little user effort was required.

Runtimes Table 3 presents the performance of a constraint-based synthesizer over arithmetic, sorting and dynamic programming benchmarks using the parameters from Tables 1 and 2. All runtimes are median of three runs, measured in seconds. We measure the time for verification and the time for synthesis using the same tool. The total synthesis time varies between 0.12–9658.52 seconds, depending on the difficulty of the benchmark, with a median runtime of 14.23 seconds. The factor slowdown for synthesis varies between 1.09–92.28 with a median of 6.68.

The benchmarks we used are considered difficult even for verification. Consequently the low average runtimes for proof-theoretic synthesis are encouraging. Also, the slowdown for synthesis compared to verification is acceptable, and shows that we can indeed exploit the advances in verification to our advantage for synthesis.

5.6 Limitations and Future Work

Our synthesis system borrows the limitations of the underlying verifiers. Specifically, we added assumptions for two cases, binary search square root and merge sort, to compensate for the incomplete handling of quadratic expressions by $\texttt{VS}^3_{\texttt{QA}}$ and inefficiency of $\texttt{VS}^3_{\texttt{AX}}$, respectively. Similarly, we have to specify a set of candidate predicates for $\texttt{VS}^3_{\texttt{AX}}$—an overhead that can be alleviated using predicate inference techniques. For our experiments, this meant that at times, our guesses were not sufficient, and synthesis only succeeded after a first few failed attempts which were used to iteratively refine the set of predicates, akin to a manual run of CEGAR [4]. Aside from these avoidable incompleteness issues of verifiers, there are two major concerns for any synthesis system, namely *scalability* and *relevance*.

Scalability The synthesis conditions we generate are tractable for current solvers, even though these benchmarks are considered some of the most difficult even to verify. Yet, the synthesis conditions are not trivial, as illustrated by the need to annotate three of the sorting benchmarks to omit invariants for their outer loops. With the current system we expect to be able to synthesize programs that have more lines of code, but for which the reasoning involved is not as complicated as in our benchmarks. But to synthesize programs that are larger *and* involve more complicated reasoning, more efficient verifiers are needed.

Relevance Any solution to the synthesis conditions is a valid terminating program that satisfies the scaffold specification. But there may be multiple valid ones that may differ, for instance, in cache behavior, runtime performance, or readability. Currently, we do not prioritize the synthesized programs in any manner, but for completeness we let the solver enumerate solutions. To get another solution, we assert the negation of a solution generated in a step and iteratively ask for the next solution. Notice though that the synthesis times reported in Table 3 are for generating the first of those solutions. We envision that in the future, we can either augment synthesis conditions with constraints about relevance or use a postprocessing step to prioritize and pick relevant solutions from those enumerated.

6. Related Work

Proof-theoretic program synthesis Dijkstra [14], Gries [18] and Wirth [38] advocated that programmers write program that are correct-by-construction by manually developing the proof of correctness alongside the program. Because techniques for efficient invariant inference were unavailable in the past, synthesis was considered intractable [12]. Recently, scheme-guided synthesis [17] has been proposed but specialized to the arithmetic domain [5]. Categorizations of approaches as constructive/deductive synthesis, schema-guided synthesis and inductive synthesis are presented in a recent survey [1]. Our approach can be seen as midway between constructive/deductive synthesis and schema-guided synthesis. Some researchers proposed heuristic techniques for automation, but they cater to a very limited schematic of programs are limited in their applicability [15]. In this paper, we have shown that verification has reached a point where *automatic* synthesis is feasible.

Extracting program from proofs The semantics of program loops is related to mathematical induction. There, an inductive proof of the theorem induced by a program specification can be used to extract a program [27]. Using significant human input, theorems proved interactively in the Coq have a computational analog that can be extracted [2]. The difficulty is that the theorem is of the whole program, and proves that an output exists for the specification. Such a theorem is much more difficult than the simple theorem proving queries generated by the verification tool.

Sketching Instead of a declarative specification of the desired computation as we use, combinatorial sketching [29, 30, 31] uses an unoptimized program as the specification. A model checker eliminates invalid candidate programs that the synthesizer generates. Loops are handled in a novel but incomplete manner, by unrolling, or by using a predefined skeleton, or by using domain specific loop finitization tricks [29] that are not applicable when synthesizing true unbounded loops (which our approach synthesizes naturally using safety and liveness constraints). Sketching does not

[1] These timings are for separately (i) synthesizing the loop guards, and for (ii) synthesizing the acyclic fragments.

inherently generate the proof, although postprocessing steps can ensure correctness [30], while our approach produces the program, and the proof.

Model checking-based synthesis of automata Seminal work on model checking [3] proposed synthesizing synchronization skeletons—a problem that has recently seen renewed interest [36, 37]. Synthesis from LTL specification has also been considered [28]. For the case of reactive systems, proposals exist that reduce the synthesis problem to a game between the environment and the synthesizer where the winning strategy corresponds to the synthesized program. Recently, this approach has also been applied to program repair [25, 19], which can be seen as restricted program synthesis. Despite optimizations [24], the practicality of these approaches for complete program synthesis remains unclear.

7. Conclusions

We have presented a principled approach to synthesis that treats synthesis as a generalized verification problem. The novelty of our approach lies in generating synthesis conditions, which are composed of safety conditions, well-formedness conditions, and progress conditions, such that a satisfying solution to the synthesis conditions corresponds to a synthesized program. We have been able to use verification tools to synthesize programs, and, simultaneously, their proof (invariants, ranking functions). We have demonstrated the viability of our approach by synthesizing difficult examples in the three domains of arithmetic, sorting, and dynamic programming, all in very reasonable time. We believe the reason for the practicality of our approach is the interplay between the proof (invariants) and the statements. Specifically, by setting up constraints with both statement and proof unknowns together, statements that do not have a corresponding proof are efficiently eliminated. We believe this is the first proposal that leverages this insight for efficient and automatic program synthesis.

Acknowledgments

This work was supported in parts by the grants CCF-0430118 and CCF-0346982. We would also like to thank Avik Chaudhuri, Elnatan Reisner, and the other anonymous reviewers, for their invaluable feedback.

References

[1] D. Basin, Y. DeVille, P. Flener, A. Hamfelt, and J. F. NIlsson. Synthesis of programs in computational logic. In *LNCS 3049*.

[2] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71. Springer-Verlag, 1982.

[4] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00*, pages 154–169, 2000.

[5] Michael Colón. Schema-guided synthesis of imperative programs by constraint solving. In *LOPSTR*, pages 166–181, 2004.

[6] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV'03*.

[7] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06*, pages 415–426, 2006.

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*.

[9] P. Cousot and R. Cousot. Abstract interpretation a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*.

[10] Leonardo de Moura and Nikolaj Bjørner. Z3, 2008. `http://research.microsoft.com/projects/Z3/`.

[11] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

[12] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Math.*, 8(3):174–186, 1968.

[13] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in CS. 1990.

[14] Edsger Wybe Dijkstra. *A Discipline of Programming*. 1976.

[15] Joe W. Duran. Heuristics for program synthesis using loop invariants. In *ACM '78*, pages 891–900, New York, NY, USA. ACM.

[16] Jean-Christophe Filliâtre. Using SMT solvers for deductive verification of C and Java programs. In *SMT'08*.

[17] Pierre Flener, Kung-Kiu Lau, Mario Ornaghi, and Julian Richardson. An abstract formalization of correct schemas for program synthesis. *J. Symb. Comput.*, 30(1):93–127, 2000.

[18] David Gries. *The Science of Programming*. 1987.

[19] Andreas Griesmayer, Paul Bloem Roderick, and Byron Cook. Repair of boolean programs with an application to C. In *CAV'06*.

[20] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI'09*.

[21] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI '08*, pages 281–292.

[22] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. In *STOC '72*, pages 238–250, New York, NY, USA, 1972. ACM.

[23] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL '04*, 2004.

[24] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *FMCAD '06*, pages 117–124. IEEE Computer Society.

[25] Barbara Jobstmann, Andreas Griesmayer, and Roderick Paul Bloem. Program repair as a game. In *CAV'05*, pages 226 – 238.

[26] Shuvendu Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL '08*.

[27] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[28] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89*, pages 179–190, New York, NY, USA. ACM.

[29] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI '07*, pages 167–178, New York, NY, USA. ACM.

[30] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *PLDI '08*.

[31] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI '05*.

[32] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *PLDI '09*.

[33] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. VS^3. `http://www.cs.umd.edu/~saurabhs/pacs/`.

[34] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. VS3: SMT solvers for program verification. In *CAV '09*.

[35] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Proof-theoretic program synthesis: From program verification to program synthesis. Technical report, Microsoft Research, Redmond, 2009.

[36] Martin Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *TACAS'09*, 2009.

[37] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis. In *POPL'10*, 2010.

[38] Nicholas Wirth. *Systematic Programming: An Introduction*. 1973.