Synthesis of Programs in Computational Logic

David Basin[1], Yves Deville[2], Pierre Flener[3], Andreas Hamfelt[4], and Jørgen Fischer Nilsson[5]

[1] Department of Computer Science
ETH Zurich
Zürich Switzerland
basin@inf.ethz.ch

[2] Department of Computing Science and Engineering
Université catholique de Louvain,
Pl. Ste Barbe 2, B-1348 Louvain-la-Neuve, Belgium
yde@info.ucl.ac.be

[3] Computing Science Division,
Department of Information Technology
Uppsala University, Box 337, S-751 05 Uppsala, Sweden
Pierre.Flener@it.uu.se

[4] Computer Science Division,
Department of Information Science
Uppsala University, Box 513, S-751 20 Uppsala, Sweden
Andreas.Hamfelt@dis.uu.se

[5] Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Lyngby, Denmark
jfn@it.dtu.dk

Abstract. Since the early days of programming and automated reasoning, researchers have developed methods for systematically constructing programs from their specifications. Especially the last decade has seen a flurry of activities including the advent of specialized conferences, such as LOPSTR, covering the synthesis of programs in computational logic. In this paper we analyze and compare three state-of-the-art methods for synthesizing recursive programs in computational logic. The three approaches are constructive/deductive synthesis, schema-guided synthesis, and inductive synthesis. Our comparison is carried out in a systematic way where, for each approach, we describe the key ideas and synthesize a common running example. In doing so, we explore the synergies between the approaches, which we believe are necessary in order to achieve progress over the next decade in this field.

1 Introduction

Program synthesis is concerned with the following question: Given a not necessarily executable specification, how can an executable program satisfying the specification be developed? The notions of "specification" and "executable" are here interpreted broadly. The objective of program synthesis is to develop methods and tools to mechanize or automate (part of) this process.

In the last 30 years, program synthesis has been an active research area; see e.g. [14, 4, 40, 13, 26, 29] for a description of major achievements. The starting point of program synthesis is usually a formal specification, that is an expression in some formal language (a language having a syntax, a semantics, and usually a proof theory). Program synthesis thus has many relationships with *formal specification* [69]. As the end product is a verified correct program, program synthesis is also related to *formal methods* in the development of computer systems [22], and to *automated software engineering*. All of these disciplines share the goal of improving the quality of software.

Program Synthesis in Computational Logic. It is generally recognized that a good starting point for program synthesis is to use declarative formalisms such as functional programming or computational logic, where one specifies *what* a program should do instead of *how*. We focus here on the synthesis of recursive programs in *computational logic*, which provides an expressive and uniform framework for program synthesis. On the one hand, the specification, the resulting program, and their relationship, can all be expressed in the same logic. On the other hand, logic specifications can describe complete specifications as well as incomplete ones, such as examples or properties of the relation that is to be computed. Since all this information can be expressed in the same language, it can be treated uniformly in a synthesis process.

There exist many different approaches to program synthesis in computational logic and different ways of viewing and categorizing them. For example, one can distinguish constructive from deductive synthesis. In *constructive* synthesis, a conjecture based on the specification is constructively proved, and from this proof a program is extracted. In the *deductive* approach, a program is deduced directly from the specification by suitably transforming it. As will be shown in this paper, these two approaches can profitably be viewed together and expressed in a uniform framework. In a different approach, called *schema-based* synthesis, the idea is to use program schemas, that is some abstraction of a class of actual programs, to guide and enhance the synthesis process. Another approach is *inductive* synthesis, where a program is induced from an incomplete specification.

Objectives. Our intent in this paper is to analyze and compare three state-of-the-art methods for synthesizing recursive programs in computational logic. The chosen approaches are constructive/deductive synthesis, schema-guided synthesis, and inductive synthesis. We perform our comparison in a systematic way: we first identify common, generic features of all approaches and afterwards we use a common example to explain these features for each approach. This analysis forms the basis for an in-depth comparison. We show, for example, that from an appropriately abstract viewpoint, there are a number of synergies between the approaches that can be exploited. For example, by identifying rules with schemas, all three methods have a common, underlying synthesis mechanism and it becomes easier to see how the methods can be fruitfully combined, or differentiated. Overall, we hope that our comparison will deepen the communities understanding of the approaches — their relationships, synergies, where they excel, and why — and thereby contribute to achieving progress in this field.

We see this paper as complementary to surveys of program synthesis in computational logic (or more precisely in logic programming), in particular [26, 29]. Rather than a making a broad survey, we focus on the analysis and in-depth comparison of the different approaches and we also consider schema-guided synthesis. Due to lack of space and to comply with our objectives, some technical details are omitted. Here, the reader may rely on his or her intuitive understanding of relevant concepts or follow pointers to references in the literature.

ORGANIZATION. Section 2 presents the different elements that will be used to present and compare the chosen synthesis approaches. These elements include general features of program synthesis approaches as well as the example that will be used for their comparison. Sections 3 through 5 describe the three chosen approaches: constructive/deductive synthesis, schema-guided synthesis, and inductive synthesis. To facilitate a systematic analysis and comparison of the methods, each section has a similar structure. Section 6 compares the three approaches. Finally, Section 7 draws conclusions and presents perspectives for future developments.

2 Elements of Comparison

In the subsequent sections, we will present three synthesis approaches. For each approach, one representative method is described. However, before describing them, we first present their general features. These features are developed in the context of each particular method and serve both to facilitate our analysis and systematize our comparison. We also introduce our example.

2.1 General Features

SPECIFICATION. The starting point for program synthesis is a specification expressed in some language. For each synthesis method, we must fix the specification language and the form of the specification (e.g., a formula or a set of examples).

MECHANISM. Program synthesis methods are based on calculi and procedures prescribing how program are synthesized from specifications. Although the underlying mechanisms of the various systems differ, there are, in some cases, similar underlying concepts.

HEURISTICS. Program synthesis is search intensive and heuristics are required in practice to guide the synthesis process. Are the heuristics specific to a synthesis method or are there common heuristics? How effective are the heuristics in the different methods and to what extent do different methods structure and restrict the search space?

Background Knowledge. Usually, non-trivial specifications refer to background knowledge that formalizes information about the properties of objects used in the specification, e.g., theories about the relevant data types.

Human Interaction. Human interaction involves two different issues. First, how much can a human be automatically assisted? Second, what is the nature of human-computer interaction in synthesis? How can the human step in and, for example, give key steps rather than leave the matter to blind search? Allowing input at critical points requires appropriate system support.

Tool Support. What kind of tool support is needed for turning a synthesis method into a viable system?

Scalability. Scalability is a major concern in program synthesis. Synthesis systems should not only be able to synthesize small simple programs, but they should also be able to tackle large or complex programs that solve real-life problems.

2.2 The Chosen Example

The same example will be used throughout the paper to facilitate a comparison of the different methods. We have chosen a problem simple enough to present in full, but complex enough to illustrate the main issues associated with each approach.

Specification 21 *Let L be a list, I a natural number, and E a term. The relation $atpos(L, I, E)$ holds iff E is the element of L at position I. By convention, the first element of a list is at position 0. The atpos relation can be formally specified as follows:*

$$atpos(L, I, E) \leftrightarrow \exists P, S \,.\; append(P, E \cdot S, L) \wedge length(P, I)$$

where append and length have their usual meaning, and are assumed to be defined in the background theory.

In the formula above, and in the rest of the paper, free variables are assumed to be universally quantified over the entire formula. As list notation, we use *nil* to represent the empty list, and $H \cdot T$ for the list with head H and tail T.

3 Constructive and Deductive Synthesis

We will now look at two approaches to synthesizing programs that are often grouped together: constructive and deductive synthesis. We shall highlight their similarities by viewing both from the same perspective: In both cases, deduction can be used to synthesize programs by solving for unknowns during the application of rules.

3.1 Background

For historical reasons, and because the ideas are simplest to present there, we begin by considering synthesis of functional programs in constructive type theory.

Constructive type theories are logics used for reasoning about functional programs. The simplest example is the simply typed λ-calculus [5, 48], which we briefly review here. Programs in the simply typed λ-calculus are terms in the λ-calculus, which are built from variables, application, and abstraction. Types are built from a set of base types, closed under the function space constructor $\rightarrow$. One reasons about judgments that assert that a term t has a type T, relative to a sequence of bindings Γ, of the form $x_1 : A_1, \ldots, x_n : A_n$, which associate variables to types. The valid judgments are inductively defined by the following rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\,\mathit{hyp} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x.\, M) : (A \rightarrow B)}\,\mathit{abst}$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}\,\mathit{appl}$$

These rules comprise a deduction system for proving that a program t has a type T. Under the *propositions-as-types* interpretation, this type may also be understood as a logical proposition (reading '$\rightarrow$' as intuitionistic implication) that specifies t's properties. Of course, the specification language is quite weak, so it is difficult to specify many interesting properties. In stronger type theories, such as [24, 56], types correspond to propositions in richer logics and one can, for example, specify sorting as

$$\vdash t : (\forall x : \mathit{int\ list}\,.\, \exists y : \mathit{int\ list}\,.\, \mathit{perm}(x, y) \wedge \mathit{ord}(y))\,. \tag{1}$$

This asserts that the program t is a function that, on input x, returns an ordered permutation y.

The given deduction system can be used for *program verification*: given a program t and a specification T, prove $\vdash t : T$. For example, for p and q types, we can verify that the program $\lambda x.\, \lambda y.\, x$ satisfies the specification $p \rightarrow (q \rightarrow p)$:

$$\cfrac{\cfrac{\cfrac{x : p \in x : p,\, y : q}{x : p,\, y : q \vdash x : p}\,\mathit{hyp}}{x : p \vdash \lambda y.\, x : q \rightarrow p}\,\mathit{abst}}{\vdash \lambda x.\, \lambda y.\, x : p \rightarrow (q \rightarrow p)}\,\mathit{abst} \tag{2}$$

Perhaps less obviously, the same rules can be used for *program synthesis*: given a specification T, construct a program t such that $\vdash t : T$. This can be done by

1. Reversing the direction in which rules are applied and proofs are constructed. That is, build the proof in a goal-directed, "refinement style" way by starting with the goal and working towards the axioms.

2. Leaving the program t as an unknown, or *metavariable*, which is solved during proof.

Let's try this out in the example above. Using capital letters to indicate metavariables, we begin with

$$\vdash R : p \to (q \to p)\,.$$

Resolving this with the (conclusion of the) *abst* rule yields the new goal

$$x : p \vdash R_1(x) : (q \to p)\,,$$

where R is unified with $\lambda x.\, R_1(x)$. Applying *abst* again results in

$$x : p,\, y : q \vdash R_2(x,y) : p\,,$$

where $R_1(x) = \lambda y.\, R_2(x,y)$. Finally, applying *hyp* unifies the assumption $x : p$ with $R_2(x,y) : p$, instantiating $R_2(x,y)$ to x and completing the proof. Composing the substitutions yields the previously verified program $t = \lambda x.\, \lambda y.\, x$.

The account above is complicated by the fact that the abstraction operator λ binds variables and, to work properly, higher-order unification is required when applying rules. The rules constitute clauses in a higher-order (meta-)language and proofs are constructed by higher-order resolution. A higher-order logic programming language or logical framework based on higher-order resolution like λ-Prolog [27], ELF [61], or Isabelle [59] would support this kind of proof.

There are two conclusions we would like to draw. First, verification and synthesis are closely related activities. In fact, when rules are applied using (higher-order) resolution, they are essentially identical. The only difference is whether unification is between ground or non-ground terms, i.e., whether or not an answer substitution is built. This conclusion should not be surprising to those working in logic programming: the same sequence of resolution steps can be used to establish a ground query $p(t)$ or a non-ground one $p(X)$, generating the substitution $X = t$.

Second, constructive synthesis is of a deductive nature and the line between the two can be fine. As the analogy with Prolog shows, proofs construct objects. In type theory, the objects are programs. Indeed, the idea of proofs synthesizing programs, sometimes called *proofs-as-programs*, can be decomposed into

$$\textit{proofs-as-programs} = \textit{proofs-as-objects} + \textit{objects-as-programs}.$$

In our example, unification, not the constructivity of the logic, is responsible for constructing an object. Constructivity does not play a role in the *synthesis* of objects, but rather in their *execution* and *meaning*. That is, because the logic is constructive, the synthesized terms can be executed and their evaluation behavior agrees with the semantics of the type theory. In contrast, [49], for example, presents a classical type theory where programs correspond to (non-computable) oracles that cannot be executed. There one might say that the line is crossed from constructive (and deductive) program synthesis to deductive object synthesis.

The use of unification is at the heart of deductive and constructive synthesis. Unification is driven by resolution, to synthesize, or solve for, programs during proofs. This idea goes back to work in the 1960s on using first-order resolution to construct terms that represent plans or, more generally, programs [19, 42]. In the logical framework community, the use of higher-order metalogics to represent rules and the use of higher-order unification to apply them is now standard, e.g., [2, 8, 9, 23]. For example, the Isabelle distribution [59] comes with encodings of a number of type theories, where programs can be synthesized as described here.

The vast majority of approaches for synthesizing logic programs are based on first-order reasoning, e.g., equivalence preserving transformations. There have been many proposed methods and [26] contains a good survey. They differ in the form of their axioms (Horn clauses, *iff*-definitions, etc.), exact notion of equivalence used (and there are many, see e.g., [55]), and ease of automation. Many of these, for example unfold-fold based transformations [60], can be recast as synthesis by resolution using rules like those presented here [7, 10].

3.2 Overview

Specifications. In type theory, programs and specifications belong to different languages. When synthesizing logic programs, the specification language is typically the language of a first-order theory and the programming language is some suitable, executable subset thereof. By sharing the same language, logic programs are well suited for deductive synthesis where specifications are manipulated, using equivalence preserving transformations, until a formula with some desired form or property is reached.

Mechanism. The mechanism for synthesizing logic programs during proofs is essentially the same as what we have just seen for type theory. However, what is proved (i.e., the form of the theorem to be proven), and the proof rules used to establish it, are of course different. Namely, we will prove theorems about equivalences between specifications and programs and we will prove these theorems using rules suitable for establishing such equivalences.

For our example, we will employ the following rules:

$$\frac{}{A \leftrightarrow A}\ {\leftrightarrow}\text{-}\mathit{refl} \qquad\qquad \frac{A_1 \leftrightarrow B_1 \quad A_2 \leftrightarrow B_2}{(A_1 \vee A_2) \leftrightarrow (B_1 \vee B_2)}\ \vee\text{-}\mathit{split}$$

In addition, for building recursive programs that recurse over lists we employ the rule schema

$$\frac{A_1 \quad A_2 \quad A_3}{\forall L, \overline{X}\,.\, P(L, \overline{X}) \leftrightarrow Q(L, \overline{X})}\ \mathit{ind}\,,$$

where L is a variable ranging over lists, $\overline{X}$ denotes sequences of zero or more variables of any type, and the assumptions A_i are:

$$\begin{aligned} A_1 \equiv \forall L, \overline{X}\,.\, Q(L, \overline{X}) \leftrightarrow\ & (L = \mathit{nil} \wedge B(\overline{X})) \\ & \vee \exists H, T\,.\, L = H \cdot T \wedge S(H, T, \overline{X}) \end{aligned}$$

$$
\begin{aligned}
A_2 &\equiv \forall \overline{X} \, . \, P(nil, \overline{X}) \leftrightarrow B(\overline{X}) \\
A_3 &\equiv \forall T \, . \, (\forall \overline{X} \, . \, P(T, \overline{X}) \leftrightarrow Q(T, \overline{X})) \to \forall H, \overline{X} \, . \, P(H \cdot T, \overline{X}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \leftrightarrow S(H, T, \overline{X})
\end{aligned}
$$

This rule, which can be derived by induction on the list L, states the equivalence between predicates P and Q (which are metavariables). For the purpose of synthesis, we can take A_1 as the definition of Q, and A_2 and A_3 constrain (and will be used to define) Q's base and recursive cases. In A_3, we are allowed to use the existence of Q, when defining Q, but only on smaller arguments.

We will show below how, by applying these rules (using higher-order resolution), we can construct R while proving its equivalence to *atpos*.

Heuristics and Human Interaction. Proof rules, like those given above, can be applied interactively, semi-interactively, or even automatically. The use of a tactic based theorem prover [41], which allows users to write programs that construct proofs, leaves open the degree of automation.

[50, 51], for example, show how to completely automate the construction of such synthesis proofs in a tactic based setting. In this work, the most important tactic implements the rippling heuristic of [17, 12]. This heuristic automates the application of rewrite or equivalence preserving transformation rules in a way that minimizes differences between terms or formulas. Rippling is typically used in inductive theorem proving to enable the use of the induction hypothesis in simplifying the induction conclusion and it can be used in a similar way during program synthesis where rules that introduce recursion (like *ind*) produce induction-like proof obligations. Rippling has been used to automate completely the synthesis of a number of non-trivial logic programs. However, it should be noted that some interaction with the user is often desirable since the application of proof rules, in particular rules that build recursive programs, determines the efficiency of the synthesized program.

Background Knowledge. The approach we present here for synthesizing logic programs involves two kinds of rules. The first kind are rules, like $\leftrightarrow$–*refl* and $\vee$–*split*, which are derived rules of first-order logic. These derived rules are not, strictly speaking, necessary (provided we are working in a complete axiomatization of first-order logic), but their addition makes it easier to construct synthesis proofs by reasoning about equivalences. The second kind of rules are theory specific rules, e.g., rules about inductively defined data types like numbers and lists. The rule *ind* given above is an example of such a rule. It is derivable in a theory that axiomatizes lists and formalizes induction over lists.

Tool Support. For synthesizing the *atpos* example, we have used the Isabelle system. Isabelle's basic mechanism for proof construction is top-down proof by higher-order resolution, which is precisely what we require. Moreover, as a logical framework, Isabelle supports the derivation of new rules, so we can formally derive, and thus insure the correctness of, the specialized rules needed for synthesis; in our example, we derive the rules just presented in a standard first-order theory of lists. Finally, tactics can be used to partially, or entirely, automate proof construction. The Isabelle distribution comes with simplifiers and decision procedures that we used to semi-automate synthesis.

SCALABILITY. The search space in most approaches to deductive synthesis is quite large. In practice, building non-trivial programs requires an environment that supports heuristics for automating simple proof steps, e.g., by the application of tactics. It is also important that the user can safely augment a synthesis system with derived rules. As we will later observe, schemas, for schema guided synthesis, can be seen as derived rules specialized for synthesizing programs of a particular form, and their integration with deductive synthesis approaches can help with large scale developments. Examples of this are provided in [1].

3.3 Example

Let us illustrate our synthesis method on the *atpos* example. We wish to construct a logic program equivalent to the specification 21. As with synthesis in the type theory, we use a metavariable, R, to stand in for the desired program. Hence we start with

$$\vdash \forall L, I, E \,.\, atpos(L, I, E) \leftrightarrow R(L, I, E)\,. \tag{3}$$

Working backwards, resolving (using higher-order unification) this conclusion with the conclusion of the *ind* rule yields the three subgoals

$$\begin{array}{l} \forall L, I, E \,.\, R(L, I, E) \leftrightarrow \;(L = nil \wedge B(I, E)) \\ \qquad\qquad\qquad\qquad\qquad \vee \exists H, T \,.\, L = H \cdot T \wedge S(H, T, I, E) \\ \forall I, E \,.\, atpos(nil, I, E) \leftrightarrow B(I, E) \\ \forall T \,.\, (\forall I, E \,.\, atpos(T, I, E) \leftrightarrow \\ \qquad R(T, I, E) \rightarrow \forall H, I, E \,.\, atpos(H \cdot T, I, E) \leftrightarrow S(H, T, I, E) \end{array}$$

and Q is unified with R.

The first subgoal constitutes a program template, which will later be filled out by solving the other subgoals. In the second subgoal, expanding the definition of *atpos* results in

$$\vdash \forall I, E \,.\, (\exists P, S \,.\, append(P, E \cdot S, nil) \wedge length(P, I)) \leftrightarrow B(I, E)\,.$$

Let I and E be arbitrary. To show

$$\vdash (\exists P, S \,.\, append(P, E \cdot S, nil) \wedge length(P, I)) \leftrightarrow B(I, E)\,,$$

observe that there are no values for P or S for which $append(P, E \cdot S, nil)$ is true. Hence this subgoal is equivalent to

$$\vdash false \leftrightarrow B(I, E)\,.$$

We can complete the proof with $\leftrightarrow$*–refl*, which unifies $B(I, E)$ with *false*.

For the third subgoal, we assume the existence of an arbitrary list T and the antecedent of the implication (which amounts to an induction hypothesis) and must prove the consequent (the induction conclusion). Hence, expanding the definition of *atpos*, we assume

$$\forall I, E \,.\, (\exists P, S \,.\, \mathit{append}(P, E \cdot S, T) \wedge \mathit{length}(P, I)) \leftrightarrow R(T, I, E)$$

and we must prove, for some arbitrary H, I, and E,

$$\vdash (\exists P, S \,.\, \mathit{append}(P, E \cdot S, H \cdot T) \wedge \mathit{length}(P, I)) \leftrightarrow S(H, T, I, E)\,.$$

Now, since P ranges over lists, for any formula $\phi(l)$, $\exists P.\phi(P)$ is equivalent (by case analysis) to $\phi(\mathit{nil}) \vee \exists H, T \,.\, \phi(H \cdot T)$. Hence, the above is equivalent to

$$\begin{aligned}\vdash\ & ((\exists S \,.\, \mathit{append}(\mathit{nil}, E \cdot S, H \cdot T) \wedge \mathit{length}(\mathit{nil}, I)) \\ & \vee (\exists H', T', S.\mathit{append}(H' \cdot T', E \cdot S, H \cdot T) \wedge \mathit{length}(H' \cdot T', I))) \\ & \quad \leftrightarrow S(H, T, I, E)\,.\end{aligned}$$

We proceed by decomposing the disjunction on the left-hand side by resolving with $\vee$*–split*. Doing so builds a disjunction for S, by instantiating $S(H, T, I, E)$ with $S_1(H, T, I, E) \vee S_2(H, T, I, E)$, and yields the two subgoals:

$$\begin{aligned}&\vdash \exists S \,.\, \mathit{append}(\mathit{nil}, E \cdot S, H \cdot T) \wedge \mathit{length}(\mathit{nil}, I) \leftrightarrow S_1(H, T, I, E) \\ &\vdash \exists H', T', S \,.\ \mathit{append}(H' \cdot T', E \cdot S, H \cdot T) \\ &\qquad\qquad \wedge \mathit{length}(H' \cdot T', I) \leftrightarrow S_2(H, T, I, E)\end{aligned}$$

For the first, the left-hand side is true whenever $\exists S \,.\, E = H \wedge S = T \wedge I = 0$. Hence, setting S to T, this subgoal is equivalent to

$$\vdash (E = H \wedge I = 0) \leftrightarrow S_1(H, T, I, E)\,.$$

We can again discharge this using $\leftrightarrow$*–refl*, which unifies $S_1(H, T, I, E)$ with $E = H \wedge I = 0$. Now, under the standard definition of *append* and *length*, the second subgoal is equivalent to

$$\begin{aligned}\vdash\ & (\exists I'.s(I') = I \wedge (\exists T', S.\mathit{append}(T', E \cdot S, T) \wedge \mathit{length}(T', I'))) \\ & \leftrightarrow S_2(H, T, I, E)\end{aligned}$$

where $s(I')$ represents the successor of I'. We can now simplify this using the antecedent (induction hypothesis), which yields

$$(\exists I'.s(I') = I \wedge R(T, I', E)) \leftrightarrow S_2(H, T, I, E)\,.$$

We complete the proof with $\leftrightarrow$*–refl*, unifying $S_2(H, T, I, E)$ with $\exists I'.s(I') = I \wedge R(T, I', E)$.

We are done! If we apply the accumulated substitutions to the remaining assumption A_1 we have

$$\begin{aligned}\forall L, I, E \,.\ & R(L, I, E) \\ & \leftrightarrow (L = \mathit{nil} \wedge \mathit{false}) \\ & \quad \vee \exists H, T \,.\, L = H \cdot T \wedge ((E = H \wedge I = 0) \\ & \qquad\qquad \vee \exists I' \,.\, s(I') = I \wedge R(T, I', E))\,.\end{aligned}$$

and we have proved the equivalence of (3) under this definition, i.e., $atpos(L, I, E)$ is equivalent to the synthesized instance of $R(L, I, E)$.

The alert reader may have wondered why we did not complete the proof earlier by resolving with $\leftrightarrow$*–refl*. In this example, our goal was to transform *atpos* so that the result falls within a particular subset of first-order formulae, sometimes called *pure logic programs* [16] or *logic descriptions* [25], that define logic programs. These formulae can be easily translated to Horn clauses or run directly in a language like Gödel [47]. In this case, we get the clauses:

$$\begin{aligned} atpos(nil, I, E) &\leftarrow false \\ atpos(H \cdot T, I, E) &\leftarrow E = H, I = 0 \\ atpos(H \cdot T, I, E) &\leftarrow s(I') = I, atpos(T, I', E) \end{aligned}$$

which can be simplified to

$$\begin{aligned} atpos(E \cdot \_, 0, E) &\leftarrow \\ atpos(\_ \cdot T, s(I'), E) &\leftarrow atpos(T, I', E) \end{aligned}$$

3.4 Analysis

Overall, when cast in this way, the deductive synthesis of logic programs is quite similar to the previous constructive/deductive synthesis of functional programs. In both cases, we leave the program as an unknown, and solve for it, by unification, during proof. Of course, the metatheoretic properties of the programs produced are quite different. In the case of logic program synthesis, the rules, as they are given, do not enforce that the object constructed has any special syntactic properties (e.g., is a pure logic program); we only know that it is an equivalent formula. Moreover, we do not *a priori* know anything about its termination behavior (although it is not difficult to show that the induction rule builds predicates that terminate when the first argument is ground).

This kind of development, as with most approaches to logic program synthesis, is best described as deductive synthesis. They are constructive only in the weak sense that, at the metalevel (or metalogic, if one is carrying out the proof in a logical framework), one is essentially proving a theorem of the form

$$\exists R \,.\, \forall L, I, E \,. atpos(L, I, E) \leftrightarrow R(L, I, E)$$

and building a witness (in this case, a predicate definition) for R. (For more on this notion of constructivity and the proof theory behind it, see [11].) Many proposed methods for the constructive synthesis of logic programs can also be explained in this way. For example, the Whelk Calculus of [71], which is motivated by experiments in synthesizing relations in a constructive type theory, can be recast as this kind of synthesis [6].

4 Schema-Guided Synthesis

We here outline Flener, Lau, Ornaghi, and Richardson's definition, representation, and semantics of program schemas: see [33] for details.

4.1 Background

Intuitively, a program schema is an abstraction of a class of actual programs, in the sense that it represents their data-flow and control-flow, but neither contains all their actual computations nor all their actual data structures. Program schemas have been shown to be useful in a variety of applications. In synthesis, the main idea is to simplify the proof obligations by taking the difficult ones offline, so that they are proven once and for all at schema design time. Also, the reuse of existing programs is made the main synthesis mechanism.

A symbol occurring in a theory T is *open* [52] in T if it is neither defined in T, nor a predefined symbol. A non-open symbol in T is *closed* in T. A theory with at least one open symbol is an *open* theory; otherwise it is *closed*. This terminology applies to formal specifications and logic programs. An (open) program for a relation r is *steadfast* [25, 53] with respect to its specification if it is correct with respect to its specification whenever composed with programs that are correct with respect to the specifications of its (open) relations other than r.

Among the many possible forms of programs, there are the *divide-and-conquer programs* with one recursive call: if a distinguished formal parameter, called the *induction parameter*, say X, has a minimal value, then one can directly solve for a corresponding other formal parameter, called the *result parameter*, say Y; otherwise, X is decomposed into a "smaller" value T (under some well-founded relation $\prec$) by splitting off a quantity H, so that a sub-result V corresponding to T can be computed by a recursive call, and an overall result Y can be composed from H and V. A third formal parameter, called the *passive parameter*, say Z, participates unchanged in these operations. Formally, this *problem-independent* dataflow and control-flow can be captured in the following open program for r:

$$\begin{aligned} r(X,Y,Z) &\leftarrow min(X,Z), solve(X,Y,Z) \\ r(X,Y,Z) &\leftarrow \neg min(X,Z), dec(X,Z,H,T), \\ &\qquad r(T,V,Z), comp(H,Z,V,Y) \end{aligned} \tag{DC}$$

The relations min, $solve$, dec, $comp$ are open. When I is the induction parameter, L the result, and E the passive parameter, so that $atpos(L,I,E) \leftrightarrow r(I,L,E)$, a closed program for $atpos$ is the instance of DC under the program substitution

$$\begin{array}{ll} min(X,Z) \leftarrow X = 0 & solve(X,Y,Z) \leftarrow Y = Z \cdot S \\ dec(X,Z,H,T) \leftarrow X = s(T) & comp(H,Z,V,Y) \leftarrow Y = F \cdot V \end{array} \tag{ϕ_1}$$

This substitution captures the *problem-dependent* computations of that program.

But programs by themselves are syntactic entities, hence some programs are undesired instances of open programs. For instance, the generate-and-test program $r(X,Y,Z) \leftarrow g(X,Y,Z),\ t(Y,Z)$ is an instance of DC under the substitution

$$\begin{array}{rr} min(X,Z) \leftarrow true & solve(X,Y,Z) \leftarrow g(X,Y,Z),\ t(Y,Z) \\ dec(X,Z,H,T) \leftarrow true & comp(H,Z,V,Y) \leftarrow true \end{array}$$

An open program such as DC thus has no fixed meaning. The knowledge captured by an open program is not completely formalized, and the domain knowledge and underlying language are still implicit. In order for such open programs to be useful for guiding synthesis, such undesired instances need to be prevented and some semantic considerations need to be explicitly added.

A *program schema* [33] has a name, a set of formal sort and relation parameters, a signature with sorted relation and function declarations, a set of axioms defining the declared symbols, a set of constraints restricting the actual parameters, an open program T called the *template*, and specifications S of the relations in T, such that T is steadfast with respect to S in that axiomatization.

The schema $\mathcal{DC}$ can be abduced, as in [32], from our informal account of how divide-and-conquer programs work. The parameters SX, SY, SZ, SH are sorts; they are used in the signatures of the other parameters, which are relations. There are no axioms because the signature declares no other symbols than the parameters. The template is the open program DC, which defines the relation r and has min, $solve$, dec, $comp$ as open relations. The closed relation r is specified by S_r, and the open relations have S_{min}, S_{solve}, S_{dec}, S_{comp} as specifications. The conditional specification S_r exhibits i_r, o_r as the input/output conditions of r, while S_{dec} exhibits i_{dec}, o_{dec} as the input/output conditions of dec. The input/output conditions of the remaining open relations are also expressed in terms of the parameters i_r, i_{dec}, o_r, o_{dec}. The constraints restrict dec to succeed at least once if its input condition holds, and then to yield a value that satisfies the input condition of r (so that a recursive call to r is "legal") and that is smaller than X according to $\prec$, which must be a well-founded relation (so that recursion terminates). The open program DC is steadfast with respect to S_r, within the given axiomatization.

In the schema $\mathcal{REUSE}$, the parameters SX, SY, SZ are sorts; they are used in the signatures of the other parameters, which are relations. There are no axioms because the signature declares no other symbols than the parameters. The template is the open program $\{r(X,Y,Z) \leftarrow q(X,Y,Z)\}$, which defines the relation r and has q as the open relation. The relation r is specified by S_r, and the relation q has the same input/output conditions as r. There are no constraints on the parameters. This schema provides for the reuse of a program for q when starting from a specification for r. The open program *Reuse* is steadfast with respect to S_r, within the given axiomatization.

4.2 Overview

Let us now examine the specifications, mechanism, heuristics, background knowledge, human interaction, tool support, and scalability of schema-guided synthesis.

Schema $\mathcal{DC}(\mathsf{SX}, \mathsf{SY}, \mathsf{SZ}, \mathsf{SH}, \prec, i_r, o_r, i_{dec}, o_{dec})$

$$
\begin{array}{lll}
\text{SORTS:} & \mathsf{SX}, \mathsf{SY}, \mathsf{SZ}, \mathsf{SH} & \\
\text{RELATIONS:} & i_r, i_{dec} : (\mathsf{SX}, \mathsf{SZ}) \qquad \prec \ : (\mathsf{SX}, \mathsf{SX}) & \\
 & o_r \quad : (\mathsf{SX}, \mathsf{SY}, \mathsf{SZ}) \qquad o_{dec} : (\mathsf{SX}, \mathsf{SZ}, \mathsf{SH}, \mathsf{SX}) & \\
\text{AXIOMS:} & \text{(none)} & \\
\text{CONSTRS:} & i_{dec}(X,Z) \rightarrow \exists H : \mathsf{SH} \,.\, \exists T : \mathsf{SX} \,.\, o_{dec}(X,Z,H,T) & (C_1) \\
 & i_{dec}(X,Z) \wedge o_{dec}(X,Z,H,T) \rightarrow i_r(T,Z) \wedge T \prec X & (C_2) \\
 & \mathit{wellFounded}(\prec) & (C_3) \\
\text{SPECIFS:} & i_r(X,Z) \rightarrow (\ r(X,Y,Z) \leftrightarrow o_r(X,Y,Z)\ ) & (S_r) \\
 & i_r(X,Z) \rightarrow (\ \mathit{min}(X,Z) \leftrightarrow \neg i_{dec}(X,Z)\ ) & (S_{min}) \\
 & i_r(X,Z) \wedge \neg i_{dec}(X,Z) \rightarrow (\ \mathit{solve}(X,Y,Z) \leftrightarrow o_r(X,Y,Z)\ ) & (S_{solve}) \\
 & i_{dec}(X,Z) \rightarrow (\ \mathit{dec}(X,Z,H,T) \leftrightarrow o_{dec}(X,Z,H,T)\ ) & (S_{dec}) \\
 & o_{dec}(X,Z,H,T) \wedge o_r(T,V,Z) \rightarrow & \\
 & \qquad\qquad (\ \mathit{comp}(H,Z,V,Y) \leftrightarrow o_r(X,Y,Z)\ ) & (S_{comp}) \\
\text{TEMPLATE:} & r(X,Y,Z) \leftarrow \mathit{min}(X,Z), \mathit{solve}(X,Y,Z) & \\
 & r(X,Y,Z) \leftarrow \neg \mathit{min}(X,Z), \mathit{dec}(X,Z,H,T), & (DC) \\
 & \qquad\qquad r(T,V,Z), \mathit{comp}(H,Z,V,Y) &
\end{array}
$$

Schema $\mathcal{REUSE}(\mathsf{SX}, \mathsf{SY}, \mathsf{SZ}, i_r, o_r)$

$$
\begin{array}{lll}
\text{SORTS:} & \mathsf{SX}, \mathsf{SY}, \mathsf{SZ} & \\
\text{RELATIONS:} & i_r : (\mathsf{SX}, \mathsf{SZ}) \qquad o_r : (\mathsf{SX}, \mathsf{SY}, \mathsf{SZ}) & \\
\text{AXIOMS:} & \text{(none)} & \\
\text{CONSTRAINTS:} & \text{(none)} & \\
\text{SPECIFICATIONS:} & i_r(X,Z) \rightarrow (\ r(X,Y,Z) \leftrightarrow o_r(X,Y,Z)\ ) & (S_r) \\
 & i_r(X,Z) \rightarrow (\ q(X,Y,Z) \leftrightarrow o_r(X,Y,Z)\ ) & (S_q) \\
\text{TEMPLATE:} & r(X,Y,Z) \leftarrow q(X,Y,Z) & (\mathit{Reuse})
\end{array}
$$

Specifications. Among the many possible forms of specifications, there are the classical *conditional specifications*: under some input condition i_r on inputs X, Z, a program for relation r succeeds iff some output condition o_r on X, Z and output Y holds. Formally, this gives rise to the following open specification of r:

$$
\begin{array}{ll}
\forall X : \mathsf{SX} \,.\, \forall Y : \mathsf{SY} \,.\, \forall Z : \mathsf{SZ} \,. & (\mathit{Cond}) \\
i_r(X,Z) \rightarrow (\ r(X,Y,Z) \leftrightarrow o_r(X,Y,Z)\ ) &
\end{array}
$$

The open symbols are the relations i_r, o_r and the sorts SX, SY, SZ. Other forms of specification can also be handled.

Mechanism. *Schema-guided synthesis* from a specification S_0 is a tree construction process consisting of 5 steps, where the initial tree has just one node, namely S_0:

1. Choose a specification S_i that has not been handled yet.
2. Choose a program schema with parameters P, axioms A, constraints C, template T, and specifications S.

3. Infer a substitution θ_1 under which S_i is an instance of the specification (available in S) of the defined relation in template T. This instantiates some (if not all) of the parameters P.
4. Choose a substitution θ_2 that instantiates the remaining (if any) parameters in P, such that the constraints C hold (i.e., such that $\theta_1 \cup \theta_2 \vdash C$) and such that one can reuse existing programs P_Q for some (if not all) of the now fully instantiated specifications $S \cup \theta_1 \cup \theta_2$ of the open relations in template T. Simplify the remaining (if any) specifications in $S \cup \theta_1 \cup \theta_2$, yielding S_G.
5. Add $T \cup P_Q$ — called the *reused program* — to the node with S_i and add the elements of S_G to the unhandled specifications, as children of S_i.

These steps are iterated until all specifications have been handled; the overall result program P_0 for S_0 is then assembled by conjoining, at each node, the reused programs. If any of these steps fails, synthesis backtracks to its last choice point. Schema-guided program synthesis is thus a recursive specification (problem) decomposition process followed by a recursive program (solution) composition process.

The $\mathcal{REUSE}$ schema can be chosen at Step 2; it forces the reuse at Step 4 of a program for q, because q is its only open relation. *Every* schema leads to some reuse at Step 4; for instance, $\mathcal{DC}$ results in the reuse of a program for *dec*.

HEURISTICS. Many choice points reside in schema-guided synthesis, so heuristics are needed to make good decisions, possibly by looking ahead into the synthesis.

Some heuristics can be applied when designing a schema. For instance, a *synthesis strategy* is the choice at Step 4 of the open relations for which programs are reused. All templates envisaged by us so far have only a few meaningful strategies, hence it is best to hardwire these. For instance, template DC has only two interesting strategies: when starting with *dec*, the divide-and-conquer schema is as above; when starting with *comp*, it would have to be reexpressed in terms of the input/output conditions of r and *comp*, giving rise to another schema, with the same template.

Other heuristics can be expressed as applicability conditions. For instance, the question arises of *what* program schema to apply at Step 2. An implicit heuristic can be achieved by ordering the schemas; putting $\mathcal{REUSE}$ first would enforce our emphasis on reuse. There also is the question of *how* to apply a chosen program schema at Step 3. For instance, with $\mathcal{DC}$, one of the formal parameters in the given specification S_r has to be the induction parameter, and another the result parameter. This can be done based on the sort information in S_r: only a parameter of an inductively defined sort can be the induction parameter. One can also augment specifications with mode information, because parameters declared to be ground at call-time are particularly good induction parameters [25].

BACKGROUND KNOWLEDGE. Step 2 assumes a base of program schemas, capturing a range of program classes. Also, Step 4 relies on a base of reusable programs. For instance, for the $\mathcal{DC}$ schema, a base of specifications and programs for *dec* programs and $\prec$ well-founded relations needs to be available.

HUMAN INTERACTION. Schema-guided synthesis can be fully automated, as demonstrated with CYPRESS [65], KIDS [66], DESIGNWARE [67], and PLANWARE [15]. However, interactive synthesis is preferable, with the human programmer taking the creative, high-level, heuristic design decisions, and the synthesizer doing the more clerical work. The design issues are intelligible to humans because the very objective of program schemas is to capture recognized, useful, human-designed programming strategies and program classes.

TOOL SUPPORT. An implementation of schema-guided synthesis can be made on top of any existing proof planner, exploiting the fact that program schemas can be seen as proof methods [35]. This provides support for the necessary higher-order matching and discharging of proof obligations.

SCALABILITY. The search space of schema-guided synthesis is much smaller than for deductive synthesis. First, schema-guided synthesis by definition bottoms out in reuse, both of the template itself and of existing programs. One can significantly reduce the number of reuse queries by applying heuristics detecting that an *ad hoc* program can be trivially built from the specification. Second, the proof obligations of Steps 3 and 4 are quite lightweight. Schema-guided synthesis thus scales up to real-life synthesis tasks, especially if coupled with a powerful program optimization workbench and sufficient domain knowledge. For instance, Smith [67] has successfully deployed his tools on real-life problems, such as transportation scheduling.

4.3 Example

Let us synthesize a program from the following specification, open in sort ST:

$$\begin{array}{l} \forall L : list(\mathsf{ST}) \,.\, \forall I : nat \,.\, \forall E : \mathsf{ST} \,.\, true \rightarrow \\ (\, atpos(L, I, E) \\ \quad \leftrightarrow \exists P, S : list(\mathsf{ST}) \,.\, append(P, E \cdot S, L) \wedge length(P, I)) \end{array} \qquad (S_{atpos})$$

The first iteration of synthesis proceeds as follows. At Step 1, the specification S_{atpos} is chosen because it is the only unhandled specification. At Step 2, suppose schema $\mathcal{DC}$ is chosen, after a failed attempt to apply schema $\mathcal{REUSE}$. At Step 3, the specification S_{atpos} is inferred to be an instance of S_r, when $atpos(L, I, E)$ is seen as $r(I, L, E)$, under the substitution

$$\begin{array}{rl} \langle \mathsf{SX}, \mathsf{SY}, \mathsf{SZ} \rangle & = \langle nat, list(\mathsf{ST}), \mathsf{ST} \rangle \\ i_r(X, Z) & \leftrightarrow true \\ o_r(X, Y, Z) & \leftrightarrow \exists P, S : list(\mathsf{ST}) \,.\; append(P, Z \cdot S, Y) \\ & \qquad \wedge length(P, X) \end{array} \qquad (\phi_2)$$

So far, 5 of the 9 parameters of $\mathcal{DC}$ have been instantiated. At Step 4, suppose the following substitution is chosen:

$$\begin{array}{rr} \mathsf{SH} = nat & A \prec B \leftrightarrow B = s(A) \\ i_{dec}(X, Z) \leftrightarrow \neg X = 0 & o_{dec}(X, Z, H, T) \leftrightarrow X = s(T) \end{array}$$

This instantiates the remaining 4 parameters of $\mathcal{DC}$ in a way that the constraints C_1, C_2, C_3 hold and that the program $P_{dec} = \{dec(X,Z,H,T) \leftarrow X = s(T)\}$ can be reused to meet the now fully instantiated specification S_{dec}. The specifications of the remaining open relations in template DC are now also fully instantiated:

$$
\begin{array}{ll}
true \rightarrow (\ min(X,Z) \leftrightarrow \neg\neg X = 0\ ) & (S_{min}) \\
true \wedge \neg\neg X = 0 \rightarrow & \\
(solve(X,Y,Z) \leftrightarrow \exists P,S \,.\, append(P, Z \cdot S, Y) \wedge length(P,X)) & (S_{solve}) \\
X = s(T) \wedge \exists P,S \,.\, append(P, Z \cdot S, V) \wedge length(P,T) \rightarrow & \\
(\ comp(H,Z,V,Y) \leftrightarrow \exists P',S' \,.\ append(P', Z \cdot S', Y) & (S_{comp}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge length(P',X)\ ) &
\end{array}
$$

They can be simplified into the following specifications:

$$
\begin{array}{ll}
min(X,Z) \leftrightarrow X = 0 & (S'_{min}) \\
X = 0 \rightarrow (\ solve(X,Y,Z) \leftrightarrow \exists S : list(\mathsf{ST}) \,.\, Y = Z \cdot S\ ) & (S'_{solve}) \\
X = s(T) \wedge \exists P,S \,.\ \ append(P, Z \cdot S, V) \wedge length(P,T) \rightarrow & (S'_{comp}) \\
\qquad\qquad\qquad (comp(H,Z,V,Y) \leftrightarrow \exists F : \mathsf{ST} \,.\, Y = F \cdot V) &
\end{array}
$$

At Step 5, the program $DC \cup P_{dec}$ becomes the reused program for S_{atpos}, while S'_{min}, S'_{solve}, and S'_{comp} are added to the now empty list of unhandled specifications.

The next iterations of synthesis proceed as follows. When S'_{min}, S'_{solve}, and S'_{comp} are chosen, suppose applications of some suitable variants of $\mathcal{REUSE}$ succeed through the *ad hoc* building of the programs $P_{min} = \{min(X,Z) \leftarrow X = 0\}$, $P_{solve} = \{solve(X,Y,Z) \leftarrow Y = Z \cdot S\}$, and $P_{comp} = \{comp(H,Z,V,Y) \leftarrow Y = F \cdot V\}$. Since no new specifications were created, the synthesis is completed and has discovered the substitution ϕ_1. For call-mode $atpos(+,-,+)$, say, the corresponding logic program

$$
\begin{array}{l}
atpos(L,I,E) \leftarrow I = 0,\ L = E \cdot S \\
atpos(L,I,E) \leftarrow \neg I = 0,\ I = s(T),\ atpos(V,T,E),\ L = F \cdot V
\end{array}
$$

can be implemented [25], say by the Mercury compiler [68], into the following steadfast program:

$$
\begin{array}{r}
atpos(E \cdot S, 0, E) \leftarrow \\
atpos(F \cdot V, s(T), E) \leftarrow atpos(V,T,E)
\end{array}
$$

The *comp* operator had to be moved in front of the recursive call to achieve this. (Prolog cannot do this, so mode-specific implementation is left as a manual task to the Prolog programmer.)

This example illustrated a relatively simple use of the $\mathcal{DC}$ schema. In [31], a quicksort program is synthesized, using a variant of the divide-and-conquer schema $\mathcal{DC}$ with two recursive calls.

4.4 Analysis

Schema-guided synthesis captures recognized, useful, human-designed programming strategies and program classes in program schemas. In doing so, it takes the hardest proof obligations offline, preventing their repeated proof across various syntheses and making reuse of existing programs the central mechanism for synthesizing programs. In the presence of powerful program optimization tools and sufficient domain knowledge, it thus naturally scales up, without any limitations on specification forms or program forms, due to the modular nature of the various forms of background knowledge. Heuristic guidance issues are still best tackled by humans, so schema-guided synthesis is best carried out interactively.

A unified view of schema-guided synthesis and proof planning has been proposed [35], revealing potential new aspects of program schemas, such as applicability conditions capturing heuristics, as well as the possibility of formulating program schemas as proof methods and thereby reusing an existing proof planner as a homogeneous implementation platform for both the schema applications and the proof obligations of schema-guided synthesis.

Our future work includes redoing the constraint abduction process for more general divide-and-conquer templates, where some $nonMinimal(X, Z)$ is not necessarily $\neg min(X, Z)$, and crafting the corresponding strategies, in order to allow the synthesis of a larger class of programs. Other design methodologies need to be captured in logic programming schemas; for instance, a global search schema has been proposed for the synthesis of constraint logic programs [37].

5 Inductive Synthesis

Following a brief introduction to inductive generalization, we present a particular approach to induction of recursive logic program called compositional inductive synthesis, which is described in detail in [46].

5.1 Background

The inductive approach to program synthesis originates in inductive logic. Inductive logic is concerned with the construction of logical theories T explaining available observations or events. This means that, given evidence in the form of atomic formulas $a_1, a_2, \ldots, a_s$, the logical induction approach is to devise an appropriate logical theory T so that

$$T \vdash a_1 \wedge a_2 \wedge \ldots \wedge a_s.$$

A major concern is to constrain T so as to rule out trivial solutions, such as T being inconsistent (thus supporting any evidence), or T being identical to the conjunction of available evidence. In the more traditional application of logical theories of induction in artificial intelligence, the quest is for a theory T taking the form of general rules, e.g., scientific rules, supporting the given evidence. In the context of induction of logic programs addressed here, the "observations" are intended sample program input-output results in the form of atomic formulas, and the theory T is to be a definite clause logic program. Thus the consistency of T is guaranteed, but computational properties such as termination and computational tractability of the synthesized program have to be separately considered.

So the goal of inductive logic programming (ILP) is to obtain a collection of clauses with universally quantified variables, which subsumes the given finite list of intended program results. The main approach to achieve this goal is syntactic generalization of the given examples. Consider atoms $p(a, a{\cdot}b{\cdot}nil)$ and $p(b, b{\cdot}nil)$. These two unit clauses generalize to the clause program $p(X, X{\cdot}Y) \leftarrow$. This rests on the existence of a dual of the most general unifier of two atoms known as the least general generalization (LGG) [63, 62]. In this simple case, the LGG yields the intended program as a unit clause witness, $p(X, X \cdot Y) \vdash p(a, a \cdot b \cdot nil) \wedge p(b, b \cdot nil)$.

The syntactical generalization of terms has been extended to a notion of generalized subsumption of clauses [18, 63] and further to a method known as inverse resolution, see e.g., [58]. This method has proven useful for concept formation, deductive databases and data mining. However, it is too weak for induction of recursive logic programs. Consider examples of list concatenation, e.g., $p(nil, a \cdot nil, a \cdot nil)$ and $p(a \cdot nil, b \cdot nil, a \cdot b \cdot nil)$. The least general generalization yields the clause $p(X, Y \cdot nil, a \cdot Z) \leftarrow$, which fails to capture the recursive definition of concatenation. Providing more examples eventually leads to an overly general clause: the universal predicate $p(X, Y, Z)$, which subsumes all concatenation examples though it blatantly fails to capture concatenation of lists. A general remedy for over-generalization is to include negative examples, which are understood as examples in the complement set of the intended result set of atoms. In general, the key problem in synthesizing such programs is the invention and introduction of appropriate recursive forms of clauses.

Compositional inductive synthesis employs a compositional logical language for computing relations in analogy to functional programming languages intended for composing and computing functions. The method does not apply the above generalization mechanisms. A program takes the form of a variable-free predicate expression φ encompassing elementary predicates and operators for combining relations and producing new resulting relations.

Let $\varphi \vdash e$ mean that the tuple (of terms) e is deducible from the program predicate expression φ. The computational semantics of the language can then be explained by means of inference rules of the form

$$\frac{\varphi_1 \vdash e_1 \quad \ldots \quad \varphi_n \vdash e_n}{op(\varphi_1, \ldots, \varphi_n) \vdash e},$$

where e depends on op and $e_1, \ldots, e_n$, as explicated in the concrete rules below. Let $\varphi \vdash e_1 + \ldots + e_n$ mean $\varphi \vdash e_i$ for $i = 1..n$, so that $+$ combines result tuples. Thus, $\varphi \vdash e_1 + e_2 + \ldots$ expresses that the tuples e_i of the term form $\langle t_1, t_2, \ldots, t_n \rangle$ are computable from the n-ary predicate expression φ.

In the language Combilog employed here, the given elementary predicates are constant formation, identity and list construction defined by the inference rules:

$$\frac{}{const_c \vdash \langle c\rangle} \qquad \frac{}{id \ \vdash \ \langle t,t\rangle} \qquad \frac{}{cons \ \vdash \ \langle h,t,h\cdot t\rangle}$$

In addition to the elementary predicates, there is a collection of operators, which map argument relations to relations. The three fundamental operators are here defined by:

$$\frac{\varphi \vdash \langle t_1, t_2, \ldots, t_n\rangle}{make_{\mu_1,\mu_2,\ldots,\mu_m}(\varphi) \vdash \langle t_{\mu_1}, t_{\mu_2}, \ldots, t_{\mu_m}\rangle}\,(make)$$

$$\frac{\varphi_1 \vdash e + e' \quad \varphi_2 \vdash e + e''}{and(\varphi_1,\varphi_2) \vdash e}\,(and) \qquad \frac{\varphi_1 \vdash e_1 \quad \varphi_2 \vdash e_2}{or(\varphi_1,\varphi_2) \vdash e_1 + e_2}\,(or)$$

The *make* operator is a generalized unary projection operator carrying an auxiliary vector of indices $\mu_1, \ldots, \mu_m$ serving to reorder arguments and introduce don't cares. As described in [46], COMBILOG possesses a compositional semantics in which *and* is set intersection and *or* is set union, which motivates the inference rules for the *and* and *or* operators. These operators reflect, respectively, logical conjunctions in clause bodies and multiple defining clauses.

This operator language becomes as expressive as ordinary clause programs if the language is extended with facilities for naming predicate expressions and using these names recursively in program predicate definitions. However, in the present form the language does not introduce predicate names in a program. Instead, the defined predicates are anonymous and in order to accommodate recursive formulations e.g., for list processing, the iteration operators *foldr* and *foldl* are introduced. These operators are akin to the *fold* operators in functional programming and with theoretical underpinning in the theory of primitive recursive functions as discussed in [45, 46], The associated rules are:

$$\frac{\psi \vdash \langle t_1, t_3\rangle}{foldr(\varphi,\psi) \vdash \langle t_1, nil, t_3\rangle}\,(foldr\ 0)$$

$$\frac{foldr(\varphi,\psi) \vdash \langle t_1, t_2, z\rangle \quad \varphi \vdash \langle h, z, t_3\rangle}{foldr(\varphi,\psi) \vdash \langle t_1, h\cdot t_2, t_3\rangle}\,(foldr > 0)$$

$$\frac{\psi \vdash \langle t_1, t_3\rangle}{foldl(\varphi,\psi) \vdash \langle t_1, nil, t_3\rangle}\,(foldl\ 0)$$

$$\frac{\varphi \vdash \langle h, t_1, z\rangle \quad foldl(\varphi,\psi) \vdash \langle z, t_2, t_3\rangle}{foldl(\varphi,\psi) \vdash \langle t_1, h\cdot t_2, t_3\rangle}\,(foldl > 0)$$

For instance, with *foldr* available, the well-known *append* concatenation predicate is $make_{2,1,3}(foldr(cons, id))$, where the *make* operator swaps the two first arguments.

Below we illustrate the application of the rules using the append program, proving $make_{2,1,3}(foldr(cons, id)) \vdash \langle a \cdot nil, b \cdot nil, a \cdot b \cdot nil \rangle$:

$$\cfrac{\cfrac{\cfrac{id \vdash \langle b \cdot nil, b \cdot nil \rangle}{foldr(cons, id) \vdash \langle b \cdot nil, nil, b \cdot nil \rangle}\,(foldr\ 0) \quad cons \vdash \langle a, b \cdot nil, a \cdot b \cdot nil \rangle}{foldr(cons, id) \vdash \langle b \cdot nil, a \cdot nil, a \cdot b \cdot nil \rangle}\,(foldr > 0)}{make_{2,1,3}(foldr(cons, id)) \vdash \langle a \cdot nil, b \cdot nil, a \cdot b \cdot nil \rangle}\,(make)$$

When the inference rules are used to compute result tuples, these tuples are unknown parameters to be determined in the course of the execution. In contrast, in the compositional inductive synthesis method, the result tuples are given initially, as a contribution to the result, whereas $\varphi_1, \ldots, \varphi_n$ are (partly) unknown program constituents to be determined recursively in the course of the synthesis. These inference rules are used in the way described in Section 3.1 for building proofs in a goal directed manner where the program constructs are unknowns, given as metavariables, and instantiated during proof. This facilitates understanding of the induction process as a stepwise, principled, program composition process.

5.2 Overview

Let us now present compositional inductive synthesis in terms of its generic features.

SPECIFICATIONS. In inductive synthesis, specifications are partial extensional definitions of the programs to be induced, i.e., a set of atoms or tuples constituting sample program results. No other problem specific specifications need be employed.

MECHANISM. The operators are similar to schemas in the schema guided approach to synthesis. In the present method, the program is synthesized in a strict recursive divide-and-conquer process by tentatively selecting an operator and then recursively attempting synthesis of constituent parameter programs.

Our synthesis takes advantage of the metainterpreter outlined below for compositional programs and does not rely on generalization mechanisms. The approach can be characterized as the top-down stepwise composition and specialization of a COMBILOG program intended as a solution in the sense that the program subsumes the program examples. The search involved in choosing between operators is taken care of by the back-tracking mechanism in the synthesizer.

In principle, our synthesis proceeds by introducing meta-variables for the left operand predicate expressions of $\vdash$ in the proof construction, and then successively instantiating these variables in the course of the goal-driven proof construction; in doing so, we also appeal to the rule

$$\frac{\varphi \vdash e_1 \quad \varphi \vdash e_2}{\varphi \vdash e_1 + e_2},$$

which is used for goal splitting on the program examples. Thus the above proof may be conceived of as a trace of a sample inductive synthesis proof.

In our metainterpreter system, the relationship $\varphi \vdash e$ is realized as a binary predicate *syn*, which simultaneously serves as metainterpreter and synthesizer. The key principle of our synthesis method is the inverted use of our metainterpreter so that the first argument program predicate is to be instantiated in the course of synthesizing a program.

Thus the heart of the synthesizer is clauses of the following, general, divide-and-conquer form for the available operators:

$$\begin{aligned} syn(comb(P_1, \ldots, P_m), Ex) \leftarrow\ & apply\_comb(Ex, Ex_1, \ldots, Ex_m) \\ & \wedge\ syn(P_1, Ex_1) \wedge \ldots \wedge syn(P_m, Ex_m). \end{aligned}$$

Programs consisting of an elementary predicate are trivially synthesized without recursive invocation of *syn*. Let us consider the synthesis of a basic predicate expression for the *head* predicate yielding the head of a non-empty list, given say the two examples $\langle a \cdot b \cdot nil, a\rangle$ and $\langle a \cdot nil, a\rangle$. Synthesis of *head* is initiated with a goal clause

$$\leftarrow syn(P, [[a, b], a]) \wedge syn(P,\ [[a], a]).$$

A successful proof instantiates P with the synthesized expression $make_{3,1}(cons)$.

Heuristics. A detailed description of the synthesizer is found in [46]. To prevent the synthesizer from running astray in the infinite space of possible program hypotheses, the search is conducted as an iterative deepening. To avoid unwanted trivial program solutions, further constraints are imposed on the synthesizer. Consider, for instance, synthesis of the *append* predicate. An overly general solution is obtained as the universal predicate, say, with the expression $make_{2,3,4}(const_c)$ corresponding to the clause $p(X_1, X_2, X_3)$. As mentioned, such unwanted solutions might be ruled out by the use of negative examples. However in our synthesizer we have chosen to enforce well-modedness constraints on the synthesized programs thus suppressing the above solution in favor of the recursive

$$P = make_{2,1,3}(foldr(cons, id)),$$

which is obtained as the syntactically smallest solution given the two sample results $\langle nil, nil, nil\rangle$ and $\langle a \cdot nil, b \cdot nil, a \cdot b \cdot nil\rangle$ and the mode pattern $[+, +, -]$, and complying with the usual clauses for *append*. The synthesis proceeds as a goal-driven proof construction of the sample proof shown in the above section.

Background Knowledge. The elementary predicates and the operators determine the admissible forms of programs and thereby constitute a form of background knowledge. No problem-specific background knowledge is provided but a search bias may be imposed by providing additional auxiliary predicates.

Tool Support. For synthesizing the *at_pos* program, a system called CombInduce was used, which is based on the method outlined above and described in detail in [46].

Human Interaction and Scalability. The current experimental system conducts the inductive synthesis automatically. The computational search costs limit the size of inducible programs to around 6 predicates and operators.

However, we envisage integration of the CombiInduce principles into a semi-automatic compositional development system. In this system, the programmer can offer assistance by proposing appropriate auxiliary predicates within the pertinent data type. The imposition of data types will also serve to constrain further the search space of well-moded program candidates. Recursion (fold) over lists will be generalized to other data types later.

5.3 Example

Since at this stage, the synthesis system supports list as the only data type we represent the number n as a list of length n with constants i, where i can be any constant. Synthesis of the *atpos* program from the single sample $\langle a \cdot b \cdot nil, i \cdot nil, b\rangle$ yields the solution

$$atpos = foldl(make_{4,3,2}(cons), make_{3,1}(cons)))$$

as illustrated by the following trace:

$$\dfrac{\begin{array}{l} make_{4,3,2}(cons) \vdash \\ \langle \_, a \cdot b \cdot nil, b \cdot nil\rangle \end{array} \qquad \dfrac{make_{3,1}(cons) \vdash \langle b \cdot nil, b\rangle}{\begin{array}{l} foldl(make_{4,3,2}(cons), make_{3,1}(cons)) \vdash \\ \langle b \cdot nil, nil, b\rangle \end{array}}\;(foldl\ 0)}{foldl(make_{4,3,2}(cons), make_{3,1}(cons)) \vdash \langle a \cdot b \cdot nil, i \cdot nil, b\rangle}\;(foldl > 0)$$

The synthesized program is the Combilog form of the definite clause program

$$\begin{aligned} atpos(L, I, E) &\leftarrow syn(foldl(tail', head), [L, I, E]) \\ syn(tail', [\_, F \cdot T, T]) &\leftarrow \\ syn(head, [F \cdot T, F]) &\leftarrow \end{aligned}$$

Synthesis with the *foldr* operator is not possible. However, swapping the two subgoals of *foldr* yields the operator *foldrrev* allowing the following variant program to be synthesized

$$atpos = make_{3,2,1}(foldrrev(cons, make_{1,3}(cons))).$$

The relationship between such a pair of variant programs is theoretically established by a duality theorem stated and proved in [44].

In order to facilitate the comparison of the synthesis approaches, let us transform the first Combilog form of the *atpos* definite clause program into a recursive *atpos* program. We first unfold the *atpos* clause:

$$\begin{aligned} atpos(L, nil, E) &\leftarrow head(L, E) \\ atpos(L, X \cdot T, E) &\leftarrow tail'(X, L, Z), syn(foldl(tail', head), [Z, T, E]) \end{aligned}$$

Now, unfolding *head* and *tail*, and folding back the second literal with *atpos*, we obtain the following logic program.

$$atpos(L, nil, E) \leftarrow L = E \cdot T$$
$$atpos(L, X \cdot T, E) \leftarrow L = F \cdot Z, atpos(Z, T, E)$$

5.4 Analysis

Check that meaning is preserved! Designing a metainterpreter for COMBILOG is simplified by the variable-free form of COMBILOG programs, the separation of predicate expressions and terms in separate arguments, and the elimination of introduced predicate names. These simplifications substantially reduce search and allow us to effectively use the metainterpreter as the backbone of our ILP method by reversing the provability metalogic programming *demo* predicate as examined e.g., in [43] and in [21] for ordinary definite clauses.

In [46] we compare with other inductive synthesis systems and report results on successful automatic synthesis of a number of textbook programs including non-naive as well as naive reversal of lists. The latter program makes calls for the auxiliary predicate *append*, which is recursively induced. This predicate invention, which is generally considered problematic in ILP, is handled smoothly in our compositional method since explicit predicate names are not introduced.

The outlined compositional method facilitates a program development methodology where customized domain specific operators are added to the general purpose ones. Moreover, it seems that the compositional method surpasses more traditional ILP methods with respect to predicate invention and termination of induced programs within the considered class of primitive recursive relations delineated by the available recursive operators.

6 Comparison

In this section, the synthesis approaches are compared from different points of view. First, we compare the synthesized *atpos* programs. Afterwards, we contrast the general features of the different approaches. Finally, we conclude by analyzing how schemas are used, implicitly or explicitly, in program synthesis and we suggest that they play a central role in understanding different synthesis methods. In the following, we will refer to *inductive synthesis*, *deductive synthesis*, and *schema-guided synthesis* to denote the particular synthesis methods presented in this paper.

6.1 The *atpos(L,I,E)* Program

All three methods yielded the same program. This was the case even though they differ in which variable they choose as an induction parameter: both *inductive synthesis* and *schema-guided synthesis* choose I as the induction parameter, while *deductive synthesis* chooses L. In the case of *deductive synthesis*, we could just as well have carried out induction on I. However, for *schema-guided synthesis*, switching would require a separate schema with a different template, namely with an additional non-recursive clause for the non-minimal case. The same holds for *inductive synthesis* where a fold combinator over numbers and an associated rule would be required.

In general, the choice of the induction parameter will affect the form of the resulting program and even its complexity [25]. In this regard, *deductive synthesis* offers more flexibility, as one can perform induction over any well-founded relation, and development (hence program construction) proceeds in smaller steps. Of course, in *schema-guided synthesis* and *inductive synthesis*, one can always introduce new schemas, respectively operators, corresponding to new ways of building programs, as the need arises.

6.2 Specification

The forms of the specifications in *deductive synthesis* and *schema-guided synthesis* are similar. Both are first-order formulas asserting a possibly conditional equivalence. In *inductive synthesis*, the specification is a finite set of examples (a subset of the extensional definition of the relation), which is by nature incomplete (when the extensional definition is infinite). Specifications in *inductive synthesis* may also include negative examples or properties [28, 36], but in general they remain incomplete. This incompleteness is a significant difference and, as we will see, it has far-reaching consequences. Indeed, it will play a key role in differentiating *inductive synthesis* from the other two approaches with respect to the other generic features.

For the *deductive synthesis* and *schema-guided synthesis* approaches, in contrast to *inductive synthesis*, it is important for non-trivial applications to be able to construct complex specifications and this requires ways of parameterizing and combining specifications. In our work on *deductive synthesis*, we achieve this, in practice, by using logical frameworks like Isabelle [59], which provide support for structured theory presentations. In *schema-guided synthesis*, [33] express program schemas as extensions of specification frameworks [52], which support parameterized specifications and their composition.

Of course, the use of first-order logic as a specification language has its limitations. For example, in *schema-guided synthesis*, we needed the well-foundedness of a relation $\prec$ as a constraint in the $\mathcal{DC}$ schema. However, a formalization of well-foundedness generally falls outside of first-order logic, unless one formalizes, e.g., set-theory. A work-around is to assume that some fixed collection of relations is declared to be well-founded. The alternative is to use a stronger (higher-order) logic or theory [1] where concepts such as well-foundedness can be defined and well-founded relations can be constructed. Stronger logics, of course, have their own drawbacks; in particular it is more difficult to automate deduction.

6.3 Mechanism

As presented, the mechanisms used in the three methods appear quite dissimilar. *Deductive synthesis* is oriented around derivations, *schema-guided synthesis* was described using an algorithm for applying schemas, and *inductive synthesis* uses a meta-interpreter to build programs. Yet it is possible to recast all three so that the central mechanism is the same: *a top-down application of rules is used to incrementally construct a program, during a derivation, in a correctness-preserving way.* In *deductive synthesis*, derived rules are applied top-down, using higher-order unification to build programs as a "side-effect" of proof construction. Although the mechanism for applying schemas has been presented in an algorithmic fashion, it is possible to recast *schema-guided synthesis* as the application of rules in a deductive system [1]; namely, a schema constitutes a (derivable) rule whose premises are given by the schema's constraints and (the completion of its) template and the conclusion is given by the schema's specifications. Viewed in this way, *schema-guided synthesis*, like *deductive synthesis*, constructs programs, during proofs, by the higher-order application of rules. The main distinction between the two methods boils down to the rules, granularity of steps, and heuristics/interaction for constructing proofs. Finally, in *inductive synthesis*, rules are also given for constructing COMBILOG programs. There, the rules are automatically applied by a Prolog meta-interpreter.

Although they differ in form, the rules employed by the different methods have a similar nature. Not surprisingly, in all cases, mathematical induction plays a key role in program synthesis, as it is necessary for constructing iterative or recursive programs. In *deductive synthesis*, induction principles can be derived from induction principles for data types or even the inductive (least-fixedpoint) semantics of logic programs [1]. The induction principles (perhaps in a reformulated form, e.g., the *ind* rule of Section 3.2) are then explicitly applied and their application constructs a template for a recursive program. In *schema-guided synthesis*, the correctness of schemas for synthesizing recursive programs is also justified by inductive arguments. Indeed, complex schemas can be seen as kinds of complex macro-development steps that precompile many micro steps, including induction. One might say that induction is implicitly applied when using a schema to construct recursive programs. In *inductive synthesis*, programs are iterative, instead of recursive, and programs that iterate over lists (or, more generally, other inductively defined data types) are built using *fold* rules. Again, mathematical induction principles play a role, behind-the-scenes, in justifying the correctness of iteration rules, and rule application can be seen as an implicit use of induction. There is, of course, a tradeoff. By compiling induction into specialized rules, *schema-guided synthesis* and *inductive synthesis* can take larger steps than *deductive synthesis*; however, they are more specialized. In particular, by building only iterative programs, the *inductive synthesis* method presented can sharply reduce the search space, but at the price of limited expressibility.

The underlying mechanisms are, in some respects, fundamentally different. Although all three methods are based on first-order logic, any system implementing *deductive synthesis* (respectively *schema-guided synthesis*) will require higher-order unification (respectively higher-order matching). This is necessary to construct substitution instances for variables in rules and schemas that range over functions, relations, and more generally, contexts (terms with holes); the downside is that higher-order matching and unification are more difficult than their first-order counterparts, and the existence of multiple unifiers (respectively matchers) can lead to large branching points in the synthesis search space. The operator form of COMBILOG means that rules in *inductive synthesis* manipulate only first-order terms. Moreover, all complications concerning object language variables are eliminated. This simplifies the metainterpreter and reduces the synthesis to search in the space of operator combinations subjected to well-modedness constraints.

Finally, the differing nature of the specifications, in particular, complete versus incomplete information, makes a substantial difference in the underlying semantics of the different methods and the relationship of the synthesized program to its specification. As presented here, both *deductive synthesis* and *schema-guided synthesis* construct programs that are (possibly under conditions) equivalent to some initial specification. In the case of *inductive synthesis*, equivalence is weakened to implication or entailment. This changes, of course, the semantics of the rules. Moreover it has a significant impact on extra-logical considerations, i.e., considerations that are not formalized in the synthesis logic (e.g., the program synthesized should have a particular syntactic form or complexity). In *inductive synthesis* these considerations (in particular, having a syntactically small recursive program that entails the examples) become central to the synthesis process and it is important to use a well-specified strategy, embodied in a metainterpreter, to ensure them.

6.4 Heuristics

Each of the methods presented has an infinite search space. However, the spaces are differently structured and different heuristics may be employed in searching them.

In *deductive synthesis*, one proceeds in a top-down fashion, employing induction and simplification. The search space has both infinite branching points associated with the application of higher-order unification (as there may be infinitely many unifiers) and branches of unbounded length (as induction may be applied infinitely often and simplification may not necessarily terminate). In practice, an effective heuristic is to follow an induction step by eager simplification; here, rippling can be used to control the simplification process and guarantee its termination. Moreover, with the exception of applying induction, unification problems are usually of a restricted form, involving "second-order patterns," which can be easily solved [51]. Hence, it is possible, in some cases, to use heuristics to reduce the search space to the point where synthesis can be completely automated.

Schema-guided synthesis uses a strict recursive divide-and-conquer strategy in the selection of operators and the synthesis of the parameter programs. It also employs a stepwise composition/specialization of programs where the objective is to reuse existing code. Analogous to *deductive synthesis*, critical branch-points include schema selection and selection of a substitution (higher-order matching is required as the same schema can be used in different ways). Search can be conducted as an iterative deepening search employing heuristics. Although *schema-guided synthesis* also has an infinite search space, it is fair to say that when a program is in the search space, one is likely to find it more quickly than with *deductive synthesis* since the steps in *schema-guided synthesis* are larger, and hence the program is at a shallower ply in the search tree.

The search space in *inductive synthesis* is more difficult to navigate than in the other two methods because of the additional extra-logical concerns mentioned previously. Here a strict control (dictated by a metainterpreter) is required to generate candidate programs in a particular order. To make automated search practical, the search space is restricted, a priori, by restrictions in the method. For example, the programs synthesizable are restricted to those involving iteration, instead of general recursion, and the use of combinators ensures that first-order (Prolog) unification suffices for program construction. In addition there is the well-modedness requirement and, to reduce explosive branching, the use of *or* is restricted. It is an interesting question as to whether any of these pruning measures could be profitably used in the other approaches.

6.5 Background Knowledge

The three approaches formalize background knowledge in different ways. For *deductive synthesis*, background knowledge about data types is given by a standard first-order theory augmented with appropriately reformulated (for synthesis) induction schemas (e.g., *ind*). For *schema-guided synthesis*, background knowledge must be formalized in terms of a base of program schemas, capturing a range of program classes, which may (or may not) directly incorporate information about data types, as well as a database of reusable programs and information about well-founded relations (typically associated with data types). Here, more work is usually required to formalize background knowledge, but the payoff is that this work is done once and for all and the resulting schemas can be used to reduce search and guide development to specialized classes of programs. For *inductive synthesis*, the background knowledge is basically the elementary operators (*const, id, cons*, etc.), which encode knowledge about iterative programs operating over lists. As with the other approaches, this knowledge is domain-dependent, and synthesizing programs operating over other data types would require additional rules.

6.6 Human Interaction and Scalability

The *deductive synthesis* proof presented was constructed interactively. There, within a first-order formalization of list theory, specialized rules for synthesis were derived, and interactively applied. However, proof search can also be automated using tactics and one can adjust the size of proof steps by deriving new proof rules (analogous to complex program schemas). This process of writing tactics and deriving new rules is open, leads to a customizable approach, and can, at least in theory, scale arbitrarily. The use of tactics also makes it possible to arbitrarily mix automation with human interaction.

Conversely, the *schema-guided synthesis* method was presented as fully automatable, although a human could be used to drive the selection of schemas and substitution instances. Indeed, as with *deductive synthesis*, this is often preferable, as it provides a way of influencing extra-logical concerns, such as the complexity of the synthesized program. The approach scales well as specialized schemas can be tuned to particular classes of problems (divide and conquer, global search, etc.). Moreover, there is a natural mechanism for the reuse of programs.

For the moment, there is no human interaction in the presented method for *inductive synthesis*. It is not clear either how feasible this is, given the importance that extra-logical concerns play in the synthesis process. How would a human know, for example, that steps suggested will generate the simplest possible program? The reuse of existing programs also is not handled.

It is not clear how the inductive synthesis approach can be scaled up to synthesize more complex programs with recursion or iteration. For complex examples, the incomplete nature of the input specification makes the program space so intractable that human interaction, heuristics, support for reuse, and "more complete" specification information, such as properties [30, 28], appear necessary. But even with these extensions, the purely inductive approach to the synthesis of programs with recursion or iteration remains very hard, and it seems doubtful whether this approach will ever scale up to the synthesis of complex, real-life programs.

When the synthesized program does *not* feature recursion or iteration (and methods for this are outside the scope of this paper) then the inductive synthesis approach *can* usefully scale. This is witnessed by recent progress in ILP, on problems in domains, such as face recognition [54], where only (large) sets of input/output examples are available as humans have difficulty writing a formal, complete specification [34].

6.7 Tool Support

For *deductive synthesis*, we used Isabelle [59], a generic logical framework, for our implementation. For *schema-guided synthesis*, the higher-order proof planning system $\lambda Clam$ can be used, upon reformulation of the program schemas as proof planning methods [35]; this has the nice side-effect that the proof obligations of *schema-guided synthesis* can also be discharged using the same theorem proving machinery. For *inductive synthesis*, a specialized Prolog implementation was used.

It is interesting to speculate on whether generic logical frameworks, like Isabelle, could be effectively used for all three approaches. And could the approaches even be profitably combined?

Our discussion at the top of Section 6.3 suggests that a generic logical framework can effectively be used for *schema-guided synthesis*. Of course, there are some potential drawbacks. First, a logical framework requires recasting any synthesis method as one based on theorem proving; for instance, *schema-guided synthesis* was not cast this way in Section 4. This may require some contortions; see [9] for an example of this. Second, the logical framework will impose its own discipline for presenting and structuring theories, and this may deviate from that desired by a particular synthesis method; e.g., specification frameworks [52] provide more structuring possibilities than those possible using the Isabelle system. Finally, a hand-coded synthesis system will probably be more efficient. Although it is easy to write a Prolog interpreter (to realize *inductive synthesis*) as a tactic in a logical framework, this involves a layer of metainterpretation and a corresponding slow-down in execution time. The price may be too high when substantial search is involved.

As to the question whether the approaches could be profitably combined, the answer is a clear 'yes' for *deductive synthesis* and *schema-guided synthesis*, and we will develop this point in the next sub-section. Combining *inductive synthesis* with the other approaches raises the question of how to deal with the ensuing redundancy in the overall specification, as the incomplete part supposedly is a logical consequence of the complete one. To a human programmer, examples attached to a specification that is intended to be complete often facilitate the understanding of the task. But an automated synthesizer probably does not need such help. Should there be a contradiction between the complete specification and the examples, then the overall specification is almost certainly wrong. In the absence of such a contradiction, one knows nothing about the quality of the overall specification and thus has to forge ahead. The question then arises of how to exploit the redundancy. A convincing proposal was made by Minton [57]: to cope with the instance sensitivity of the heuristics used to efficiently solve ubiquitous, NP-hard, constraint satisfaction problems, industry-strength solver synthesizers should use training instances (i.e., the input parts of examples) in addition to the specification of the problem, so that the most suitable heuristics can be empirically determined during synthesis. As long as the actual runs of the synthesised program are on instances within the distribution of the training instances, a good performance can be guaranteed.

6.8 Implicit *versus* Explicit Use of Schema

A central part of our comparison has been that the boundaries between *deductive synthesis*, *schema-guided synthesis*, and *inductive synthesis* are somewhat fluid with respect to the use of schemas. In particular, from the appropriate viewpoint, the difference between *deductive synthesis* and *schema-guided synthesis* is vanishingly small. We would like to close the comparison by driving these points home.

The derived rules in *deductive synthesis* for reasoning about equivalences are rule schemas, i.e., rules with metavariables ranging over predicates. These are metavariables from the view of a metalogic, but they also can be viewed as uninterpreted relations in the object logic and play the same role as the open relation symbols in *schema-guided synthesis*. Viewed this way, if the background theory of *deductive synthesis* is formalized as a specification framework, then the inference rules are a variation of the program schemas in *schema-guided synthesis*.

For example, the *ind* rule with its assumptions A_1–A_3 presented here in *deductive synthesis* is similar (although not equivalent) to the $\mathcal{DC}$ schema developed in *schema-guided synthesis*. In particular:

- *ind* commits to an induction parameter of type list, whereas $\mathcal{DC}$ has an open sort SX for the induction parameter;
- *ind* commits to one-step, head-tail decomposition of the induction parameter, whereas $\mathcal{DC}$ has an open relation *dec* for this;
- $\mathcal{DC}$ commits to always one recursive call in the step case, whereas *ind* is flexible (there can be any number of recursive calls);
- the assumption A_1 of *ind* plays the same role as the template DC in $\mathcal{DC}$, but they differ in content;
- the predicate variable B of *ind* plays the same role as the open relation *solve* in $\mathcal{DC}$;
- the assumption A_2 of *ind* plays the same role as the specification S_{solve} in $\mathcal{DC}$;
- the predicate variable S of *ind* does not play the same role as the open relation *comp* in $\mathcal{DC}$; indeed, an instance of S may include recursive call(s), whereas recursion is dictated by the template DC and is thus not considered when instantiating *comp*;
- the assumption A_3 of *ind* plays the same role as the specification S_{comp} in $\mathcal{DC}$, but they differ in content;
- there is no explicit equivalent of the constraints C_1, C_2, and C_3 and the specifications S_{min} and S_{dec} of $\mathcal{DC}$ in *ind*.

The differences here are not due to the underlying synthesis mechanism, but are an artifact of the particular implicit schema used (for reasons of simplicity) in this presentation of *deductive synthesis*. More elaborate rules and schemas, neither committed to a particular type nor a well-founded relation, have been developed in *deductive synthesis*, as presented in, e.g., [1, 3].

A similar comparison can be made between the *foldr* and *foldl* operators in *inductive synthesis*, and the $\mathcal{DC}$ schema in *schema-guided synthesis*. The *foldr* and *foldl* operators can also be seen as implicit program schemas. More elaborate rules could also be used to build COMBILOG programs in larger steps.

Program schemas are thus used (implicitly or explicitly) in the different synthesis approaches. In the literature, program schemas are often reduced to templates, formalized as higher-order expressions, and applied using higher-order unification. As shown in *schema-guided synthesis*, such templates must be enhanced with semantic information, expressed for instance through axioms, constraints, and specifications. Viewing such schemas as derivation rules, and schema application as logical inference, the distinction vanishes between the schema-guided and deductive/constructive approaches. For instance, in [1] it is shown how schemas for transformational development can be formalized as derived rules and combined with other kinds of verification and synthesis. In [30, 28], a $\mathcal{DC}$-like schema is used in the context of inductive synthesis.

7 Conclusion

In this paper, we have analyzed and compared representative methods of three approaches to program synthesis in computational logic. Despite their differences, we established strong similarities. In particular, program schemas are used (implicitly or explicitly) in each of the methods and are central in driving the synthesis process and exploiting synergies. We would therefore like to conclude by discussing some limitations of schemas and open issues.

Despite their central role, schemas have their limitations. Schemas are usually expressed in some logical language, but any given language has syntactical restrictions that in turn restrict what can be expressed as a schema. For example, a first-order language fixes the arity of predicates and functions, their associated types, etc. There is no way to capture certain simple kinds of generalization or extra-logical annotations, for example to employ term or atom ellipses $t_1, \ldots, t_n$ of variable length n. As an example of this limitation, consider the *ind* rule of Section 3.2. There we used $\overline{X}$ to denote a sequence of zero or more variables and hence the induction rule given cannot be captured by a single schema, but rather requires a family of schemas, one for each n. Extensions here are possible; [64, 28, 70, 39, 20] provide notions of *schema patterns* that describe such families and can be specialized as needed before, or during, synthesis.

Schemas are here defined as abstractions of classes of programs. At the same time, they formalize particular design strategies, such as *divide-and-conquer* or *global search*; part of the associated strategy can also be specified by associated tactics, which choose induction parameters, find appropriate well-founded relations, and so on. However, in their present form, schemas cannot handle more sophisticated design strategies, namely strategies abstracting a class of programs that cannot be obtained by instantiation with formulae. Typical examples are so-called *design patterns* [38], which aim at the description of software design solutions and architectures (typically described by UML diagrams and text). How to extend schemas to handle such strategies is an open problem in program synthesis.

Overall, by examining the relationships and differences between the chosen synthesis methods, we have sought to bring out synergies and possibilities for cross-fertilization, as well as limitations. The primary synergies involve a common mechanism: a notion of schematic rule and the use of unification to apply rules in a top-down way that incrementally construct a program, during a derivation that demonstrates its correctness. The primary differences concern the nature of the specifications, in particular the information present; this also manifests itself in different semantics and radically different search spaces for the different methods. As it is, the purely inductive approach to the synthesis of programs with recursion or iteration remains very hard, and it seems doubtful whether this approach will ever scale up to the synthesis of complex, real-life programs. Fortunately, fruitful combinations of these synthesis approaches exist.

In the end, we believe that progress in this field will be based on exploiting the identified synergies and possibilities for cross-fertilization, as well as supporting an enhanced, flexible use of schemas. We hope, with this paper, to have made a constructive analysis of the last decade of research, thereby showing a possible path for the next decade.

Acknowledgements

We would like to thank the anonymous referees for their feedback and our co-investigators on research related to this paper.

References

1. P. Anderson and D. Basin. Program development schemata as derived rules. *Journal of Symbolic Computation*, 30(1):5–36, 2000.
2. A. Ayari and D. Basin. Generic system support for deductive program development. In T. Margaria and B. Steffen, editors, *Proc. of TACAS'96*, volume 1055 of *LNCS*, pages 313–328. Springer-Verlag, 1996.
3. A. Ayari and D. Basin. A higher-order interpretation of deductive tableau. *Journal of Symbolic Computation*, 2002. To Appear.
4. R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, 1985.
5. H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, second, revised edition, 1984.
6. D. Basin. IsaWhelk: Whelk interpreted in Isabelle. In P. Van Hentenryck, editor, *Proc. of ICLP'94*, page 741. The MIT Press, 1994.
7. D. Basin. Logic frameworks for logic programs. In L. Fribourg and F. Turini, editors, *Proc. of LOPSTR'94 and META'94*, volume 883 of *LNCS*, pages 1–16. Springer-Verlag, 1994.
8. D. Basin. Logical-framework-based program development. *ACM Computing Surveys*, 30(3es):1–4, 1998.
9. D. Basin and S. Friedrich. Modeling a hardware synthesis methodology in Isabelle. *Formal Methods in Systems Design*, 15(2):99–122, September 1999.
10. D. Basin and B. Krieg-Brückner. Formalization of the development process. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of System Specification*, pages 521–562. Springer-Verlag, 1998.
11. D. Basin and S. Matthews. Adding metatheoretic facilities to first-order theories. *Journal of Logic and Computation*, 6(6):835–849, 1996.
12. D. Basin and T. Walsh. Annotated rewriting in inductive theorem proving. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.
13. A.W. Biermann. Automatic programming. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 59–83. John Wiley, second, extended edition, 1992.
14. A.W. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques*. Macmillan, 1984.
15. L. Blaine, L. Gilham, J. Liu, D.R. Smith, and S. Westfold. PlanWare: Domain-specific synthesis of high-performance schedulers. In *Proc. of ASE'98*, pages 270–279. IEEE Computer Society Press, 1998.
16. A. Bundy, A. Smaill, and G.A. Wiggins. The synthesis of logic programs from inductive proofs. In J.W. Lloyd, editor, *Computational Logic*, Esprit Basic Research Series, pages 135–149. Springer-Verlag, 1990.

17. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.
18. W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36(2):375–399, 1988.
19. C.-L. Chang and R.C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.
20. E. Chasseur and Y. Deville. Logic program schemas, constraints and semi-unification. In N.E. Fuchs, editor, *Proc. of LOPSTR'97*, volume 1463 of *LNCS*, pages 69–89. Springer-Verlag, 1998.
21. H. Christiansen. Implicit program synthesis by a reversible metainterpreter. In N.E. Fuchs, editor, *Proc. of LOPSTR'97*, volume 1463 of *LNCS*, pages 90–110. Springer-Verlag, 1998.
22. E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
23. M.D. Coen. Interactive program derivation. Technical Report 272, Cambridge University Computer Laboratory, UK, 1992.
24. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, pages 95–120, 1988.
25. Y. Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming. Addison-Wesley, 1990.
26. Y. Deville and K.-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19–20:321–350, 1994.
27. A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In E.L. Lusk and R.A. Overbeek, editors, *Proc. of CADE'88*, volume 310 of *LNCS*, pages 61–80. Springer-Verlag, 1988.
28. P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.
29. P. Flener. Achievements and prospects of program synthesis. In A.C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond; Essays in Honour of Robert A. Kowalski*, volume 2407 of *Lecture Notes in Artificial Intelligence*, pages 310–346. Springer-Verlag, 2002.
30. P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation*, 15(5–6):775–805, 1993.
31. P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In *Proc. of ASE'97*, pages 153–160. IEEE Computer Society Press, 1997.
32. P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N.E. Fuchs, editor, *Proc. of LOPSTR'97*, volume 1463 of *LNCS*, pages 124–143. Springer-Verlag, 1998.
33. P. Flener, K.-K. Lau, M. Ornaghi, and J.D.C. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, 2000.
34. P. Flener and D. Partridge. Inductive programming. *Automated Software Engineering*, 8(2):131–137, 2001.
35. P. Flener and J.D.C. Richardson. A unified view of programming schemas and proof methods. In A. Bossi, editor, *Proc. of LOPSTR'99*, pages 75–82. Tech. rept. CS-99-16, Univ. of Venice, Italy, 1999. Also see Technical Report 2003-008 at the Department of Information Technology, Uppsala University, Sweden, 2003.
36. P. Flener and S. Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2–3):141–195, 1999.

37. P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of CLP programs. In *Proc. of ASE'98*, pages 168–176. IEEE Computer Society Press, 1998.
38. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
39. T.S. Gegg-Harrison. Extensible logic program schemata. In J. Gallagher, editor, *Proc. of LOPSTR'96*, volume 1207 of *LNCS*, pages 256–274. Springer-Verlag, 1997.
40. A.T. Goldberg. Knowledge-based programming: A survey of program design and construction techniques. *IEEE Transactions on Software Engineering*, 12(7):752–768, 1986.
41. M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
42. C. Green. Application of theorem proving to problem solving. In *Proc. of IJCAI'69*, pages 219–239. Morgan Kaufmann, 1969.
43. A. Hamfelt and J. Fischer Nilsson. Inductive metalogic programming. In S. Wrobel, editor, *Proc. of ILP'94*, volume 237 of *GMD-Studien*, pages 85–96, 1994.
44. A. Hamfelt and J. Fischer Nilsson. Declarative logic programming with primitive recursive relations on lists. In M.J. Maher, editor, *Proc. of JICSLP'96*, pages 230–243. The MIT Press, 1996.
45. A. Hamfelt and J. Fischer Nilsson. Towards a logic programming methodology based on higher-order predicates. *New Generation Computing*, 15(4):421–448, 1997.
46. A. Hamfelt, J. Fischer Nilsson, and N. Oldager. Logic program synthesis as problem reduction using combining forms. *Automated Software Engineering*, 8(2):167–193, 2001.
47. P. Hill and J.W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.
48. J.R. Hindley and J.P. Seldin. *Introduction to Combinators and the λ-Calculus*. Cambridge University Press, 1986.
49. D.J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proc. of LICS'91*, pages 162–172. IEEE Computer Society Press, 1991.
50. I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K.-K. Lau and T. Clement, editors, *Proc. of LOPSTR'92*, Workshops in Computing Series, pages 1–14. Springer-Verlag, 1993.
51. I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996.
52. K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. of LOPSTR'94 and META'94*, volume 883 of *LNCS*, pages 104–121. Springer-Verlag, 1994.
53. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *Journal of Logic Programming*, 38(3):259–294, 1999.
54. C.L. Lisett and D.E Rumelhart. Facial recognition using a neural network. In *Proc. of the 11th International Florida AI Research Symposium FLAIRS-98*, pages 328–332, 1998.
55. M.J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1987.
56. P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.
57. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1–2):7–43, 1996.

58. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3–4):245–286, 1995.
59. L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.
60. A. Pettorossi and M. Proietti. Transformation of logic programs. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Clarendon Press, 1998.
61. F. Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
62. G.D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, 1970.
63. J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 135–151. Edinburgh University Press, 1970.
64. D.R. Smith. The structure of divide and conquer algorithms. Technical Report 52-83-002, Naval Postgraduate School, Monterey, California, USA, 1983.
65. D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.
66. D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.
67. D.R. Smith. Toward a classification approach to design. In M. Wirsing and M. Nivat, editors, *Proc. of AMAST'96*, volume 1101 of *LNCS*, pages 62–84. Springer-Verlag, 1996.
68. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
69. A. van Lamsweerde. Formal specification: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 147–159. ACM Press, 2000.
70. W.W. Vasconcelos and N.E. Fuchs. An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In M. Proietti, editor, *Proc. of LOPSTR'95*, volume 1048 of *LNCS*, pages 174–188. Springer-Verlag, 1996.
71. G.A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K.R. Apt, editor, *Proc. of JICSLP'92*, pages 351–365. The MIT Press, 1992.