Trends in Software Synthesis

Summary on the seminar offered at Saarland University in WS 2015/16, with focus on *Software Synthesis Modulo Recursive Functions*

Lukas Convent

Universität des Saarlandes

Abstract

This summary depicts a method in software synthesis which was first introduced in "Synthesis Modulo Recursive Functions" ([11]). Relying on a combination of different synthesis techniques, the method can be used to generate recursive programs. After giving an overview of software synthesis in general, the mechanism of this method will be presented. Among its classification we will also show connections to other techniques used in the field of synthesis. Finally, we draw a conclusion and look at what paths in this field can be taken next.

1 Introduction

Software synthesis is the broad field of systematically inferring computer programs from specifications. There exists a variety of aims and approaches in doing so. Whereas in some of the first works on software synthesis ([1, 2]) the aim was to synthesize complete programs, in recent decades the focus shifted towards solving more specific problems, such as filling out missing parts in source code (hole-plugging, e. g. in [3]) or offering extended pattern matching ([4]).

More specifically, the aims and methods of software synthesis vary as follows:

- **Correctness guarantee**. While correctness is in many cases indispensable, there might be situations where one could accept a solution which behaves correctly for most inputs. For example, by using the heuristic setting in [5] by simulating the conjecture oracle through random sampling, an approximation to a solution is gained.
- **Quality**. Traditionally, the focus lies on finding **a** correct solution. However, software synthesis can be used to generate a solution which is optimized with regard to some criterion, like speed or memory usage. For instance this is used in [6], where the precision of fixed-point numbers is maximized.
- **Representation of specification**. Depending on its field of application, the specification can be integrated in the target programming language (e. g. through the *choose* statement in [4]) or given as a quantified formula (e. g. in [7]). Also, a correct program itself can serve as a specification for a better (deobfuscated, faster...) program.
- **(In)finite Domain and Recursion**. Most programs operate on a (theoretically) infinite domain or at least a very large finite domain. Many solutions, when required to operate on such large domains, need recursion to compute. The main method of proving a recursive program correct is induction, which is not trivial to apply. In order to be complete for a certain class of problems, some synthesis procedures will only produce loop-free programs (as in [3]). More about synthesis of recursive functions will follow on the next pages.

As the title "Application of *Theorem Proving* to Problem Solving" ([1]) of one of the first papers on this topic demonstrates, theorem proving with its approach of **natural deduction**

is an instrument heavily used in software synthesis. Its compositionality allows frameworks which offer the synthesis of programs through the inference by rules. At the same time, since logically deduced, the resulting programs are proven correct. Examples for such frameworks are presented in [2] (sequent tableaux) and [8] (deduction framework).

Another major technique used for synthesis is **SMT-solving** (SMT = *satisfiability modulo theories*). An SMT solver can decide the satisfiability of a certain class of formulas, not being restricted to boolean formulas. "Plugging in" a theory such as *integer linear arithmetic* allows formulas including connectives like $<$, $\leq$.

Some synthesis approaches are directly set on top of SMT solvers, profiting from their speed thanks to extensive research in the past. Typically, an SMT solver is used to validate a program candidate by checking whether an input for a program candidate exists which does not meet the specification. If such an input is found, it can serve as a counterexample. Otherwise there is no counterexample and thus the program candidate can be seen as valid ([7, 3]).

An extension of this technique, called *Sciduction* ([9, 10]), uses the same principle in its deductive engine. While the inductive engine's job is to derive new programs, having the observed counter-examples "in mind", the deductive engine tries to prove these generated program candidates wrong. If there is an input which causes misbehavior, it will be conveyed to the inductive engine which then can do a better job on its next run.

A combination of both techniques (Natural Deduction and SMT solving) will be depicted in the following:

2 Software Synthesis Modulo Recursive Functions

The method presented in [11] tackles the synthesis of recursive programs. Recursion is needed in almost every application in some way when dealing with large input domains (e. g. lists in a sorting algorithm). In one of the first approaches to software synthesis, recursion was not to be synthesized because induction as a proof principle for recursive programs was not established in the deduction system ([1]). In [2] then, an "induction rule" enables to apply structural induction on an inductive data type like natural numbers. Nevertheless, synthesis of recursive programs has not become easy: Finding the right way to apply the induction principle (including possible strengthening of the goal) remains highly non-trivial.

The method to be presented tries to show a new approach by making use of the two concepts introduced above: First, it extends the **deduction framework** which was introduced in [8]. Second, it relies on the *Leon Verifier*, an **SMT solver** capable of reasoning over user-defined recursive functions. By adding **three new methods**, plugged into the deduction framework as three new rules, recursive programs like *insertion of an element into a sorted list* can be synthesized.

2.1 Deduction Framework

The deduction frameworks enables one to first formulate a synthesis problem (represented as a quatruple) and then to infer a solution for it. The following statement asserts that the synthesis problem $[\![\overline{a}\ \langle \Pi \triangleright \phi \rangle\ \overline{x}]\!]$ is solved by $\langle P \mid \overline{T} \rangle$:

$$[\![\overline{a}\ \langle \Pi \triangleright \phi \rangle\ \overline{x}]\!] \quad \vdash \quad \langle P \mid \overline{T} \rangle$$

The essential components of a synthesis problem are its input variables $\overline{a}$, output variables $\overline{x}$ and a synthesis predicate ϕ which links input to output (the existence of a path condition Π only adds flexibility to the framework).

The essence of a derived solution is the program vector $\overline{T}$ which describes how the output is computed from the input. The precondition P defines the domain of the program, serving as a "global guard".

In order to justify its correctness, the statement above ("solved synthesis problem") has to respect two properties:

1. **Relation refinement**. Whenever Π and P hold, executing the program $\overline{T}$ will produce an output which satisfies ϕ.
2. **Domain preservation**. If there exists a matching output for an input, this input must not be excluded by P.

In our synthesis process, the statement above (with unknown P and $\overline{T}$) can be understood as our goal which we need to infer, using inference rules provided by the deduction framework. As an example for such an inference rule consider the GROUND-rule:

$$\frac{\mathcal{M} \models \phi \qquad vars(\phi) \cap \overline{a} = \emptyset}{[\![\overline{a} \langle\, \Pi \triangleright \phi \,\rangle \overline{x}]\!] \vdash \langle\, true \mid \mathcal{M} \,\rangle} \text{ GROUND}$$

Although a new letter $\mathcal{M}$ is introduced here, the rule really states a straightforward fact: If ϕ does not depend on the input and can be fulfilled by some model $\mathcal{M}$, this model serves as final output (thus no computation needs to be done).

While the GROUND-rule is a terminal rule in the sense that it does not require solving a subproblem as its premise, there are other rules (e.g. CASE-SPLIT) which decompose a synthesis problem into subproblems. Three new rules (including a compositional one: the Recursion Rule) will be presented on the next pages.

2.2 Leon Verifier

The Leon Verifier is an SMT solver which is able to reason over user-defined recursive functions. Given a formula, the procedure is

- complete for **counterexamples**:
  e.g. `size(xs) < 3` $\rightsquigarrow$ *Counterexample* `1::2::3::nil`
- complete for **proofs over certain classes** of formulas:
  e.g. `size(1::2::xs) > 1`

2.3 Extension 1: Recursion Rule

For every inductive data type, there is a recursion rule. Using this rule, a problem can be decomposed into several cases and solved independently.

Central idea. Typically, a recursive function involves decreasing the argument of an inductive data type (e. g. *List*). Its well-foundedness gives us an induction principle which we can use to treat the different constructors separately.

Each inductive data type has a program scheme which needs to be "filled out". This is a scheme for *List*:

```
1 def rec(a0, ā) =
2   ...
3   a0 match {
```

```
4|     case Nil ⇒ T̄1
5|     case Cons(h, t) ⇒
6|       lazy val r̄ = rec(t, ā)
7|       T̄2
8|   }
9| ensuring (r̄ ⇒ φ[x̄ → r̄])
```

The synthesis problem is thus reduced to the individual cases for the given problem (ϕ). Once sub-solutions $\overline{T}_1$ and $\overline{T}_2$ are found — together with the constructors serving as guards — a solution for the entire problem is found.

This rule is very similar to the *Case Split Rule* from [8], in the same way as a simple case analysis resembles an induction and is therefore a very natural extension.

2.4 Extension 2: Symbolic Term Exploration Rule

The Symbolic Term Exploration Rule is used to find solutions consisting only of constructors and function calls (i.e. loop-free terms). The rule can then be used to finish cases produced before by the recursion rule. Essential for this method is the use of (recursive) generators which non-deterministically ($\star$) generate values of a data type. This is a trivial example of a generator:

```
def genNat(): Nat =
  if (⋆) 0 else 1 + genNat()
```

Central idea. Using a generator which produces values that take into account the environment (like input variables), we assume it will produce a solution eventually.

Unrolling the generator up to some instantiation depth ("cutting off" the recursion) results in a function taking as arguments the determinization ($\star$) and the input. The determinization ($\star$) determines the chosen program. Now we let the Leon Verifier look for a satisfying assignment (a determinization $\overline{a}_0$ and some input $\overline{b}_0$).

If no model is found, the generator or the instantiation depth does not suffice.

If a model is found, we try to falsify it, i.e. check whether some input satisfies $\neg\phi$. Not being able to falsify means we have found a valid program, otherwise we reject it and repeat the procedure.

To speed up this process, the "cut-off" generator function can be first compiled into machine code. Rapidly running a set of example inputs on the candidates can filter out many incorrect ones before the Leon Verifier is applied. Since we are talking about executing machine code, this pruning rapidly narrows down the actual generator which will then later be used, using the method described above.

2.5 Extension 3: Condition Abduction Rule

The last contribution, the Condition Abduction Rule, will be used to find recursive function bodies. Instead of focusing on how to divide a problem into subproblems (cf. Recursion Rule), we first focus on program terms:

Central idea. We first find program candidates which solve the problem for some inputs. Then we "abduce" a guard which covers as much of the input space as possible, making sure that the guarded input space will be solved correctly. This procedure will be repeated until the whole input space is covered (cf. Figure 1). Finding the program candidates in the first place is again done by consulting suitable generators.

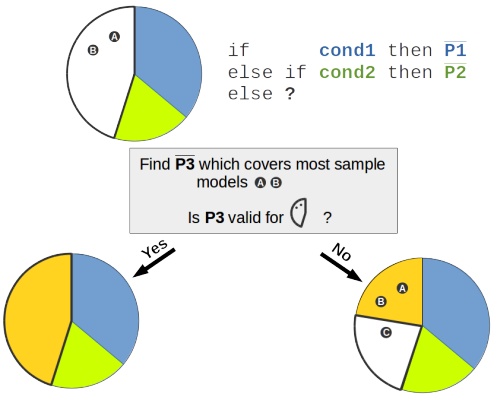

Figure 1: Successively covering the input space

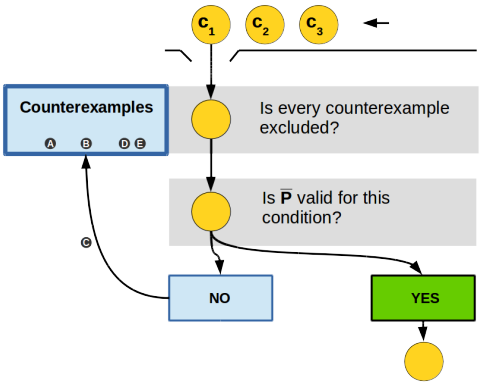

Figure 2: Abducing a condition for $\overline{P}$

Abducing a condition. This time our condition (guard) may not just be a constructor, as it was the case with the Recursion Rule. We use generators producing boolean values to find fitting expressions. Remember that we have already fixed a program $\overline{P}$. A condition candidate is now checked for validity: If any input that satisfies the condition will not produce a correct output, the condition is not acceptable. In order to sort out candidates more quickly, we keep track of counterexamples seen so far and execute those first on a candidate, so that the Leon Verifier does not need to run every time (cf. Figure 2).

2.6 Final Assembly & Classification

Having added these three new rules, finding recursive programs now becomes possible. The task is now to apply the new rules in a sensible way. Besides aiming at finding one inference tree in a reasonable amount of time, other criteria could be to approach a minimal inference tree with respect to tree size or code size.

Howsoever, we are dealing with a problem which is very detached from the synthesis mechanisms introduced before. Aiming at an efficient search, the authors in [11] suggest to add weights to different strategies and search in a breadth-first manner.

Still, there will be situations where a correct solution cannot be found. This might be due to the Leon Verifier which is not complete for all formulas. Besides a problem could be the generators which have to come up with a correct solution eventually.

Another point which becomes apparent when trying out the demo implementation[1] is that even if synthesis does not succeed completely, intermediate results can be of interest to a programmer. It can already be helpful if one single *choose* statement gets replaced by several smaller ones, which then can be solved by the programmer.

Classification. Considering the criteria stated in the introduction 1, the presented synthesis approach will only generate **correct** programs. This is due to inferring ("proving") programs by sound rules (respecting *relation refinement*, *domain preservation*). Naturally, giving correctness guarantees is a property shared by most other synthesis methods.

[1] *http://leon.epfl.ch/*

The method does **not focus on quality per se**. Aiming at synthesis of a broad class of programs, quality (speed, memory usage, ...) cannot be measured in a general way. This is different e.g. from [6], where narrowing down the problem class (to the domain of arithmetic) allows a concrete way of measurement (trying out a batch of inputs, then looking at precision).

The **integration of the specification into the programming language** is useful for at least two reasons: First, the Leon Verifier can reason about user-defined recursive functions and thus tries to reason about program code directly (looking for counter-examples). This is an essential part of the synthesis procedure. Second, through the use of *choose* statements we enable the programmer to let the synthesizer "fill out" the holes, as it was e.g. also proposed in [3].

As discussed so far, the aim is to reach out for synthesis of **recursive programs**. The techniques applied in the method have appeared before in other papers on software synthesis, but mostly not yet to synthesize recursive programs (e.g. the deduction framework of [8] only focuses on loop-free programs).

Further characteristics. As mentioned before, a central aspect is the use of counterexamples. While with *sciduction* ([9, 10]) the counterexamples produced by the deductive engine are used to improve the correctness of future candidate programs, this is not the case here. Generators remain static and will not adapt to counterexamples — this gives on the other hand the possibility to compile them into machine code and execute them quickly.

The paper also shows how a synthesis procedure can be set up in a modular way. Having established a framework of rules, one can enhance the synthesis process by new technology through the addition of rules. This approach of integrating any theory into the process has also been proposed in some other papers, e. g. [4] ("lifting decision procedures to synthesis procedures") and [12] ("integrate verification tools in synthesis procedures").

3 Conclusion & Outlook

Considering the papers discussed in the seminar, one can observe quite some development in the field of software synthesis. Early on, when research focused on logic engines to find programs ([1, 2]), enthusiasm was high that synthesis of recursive programs would be feasible in the following years. This turned out to be still very challenging without leaving the area of pure deduction systems.

The use of counterexample-driven methods like *sciduction* allows one to try out candidates and learn from counterexamples to come up with better candidates. Although trying out programs from generators is very efficient when directly executed as machine code ([11]), resorting to this method shows how complex the problem really is. Even if this method is promising for rather easy tasks, it seems far away from succeeding when complexity is raised.

Taking this into account, it is interesting to observe that many approaches have focused on solving rather "manageable" problems (like restricting one to loop-free programs) or only part of the work, letting the user guide them: E.g. in [12], the user gives a *scaffold* to the synthesis procedure. This scaffold, besides other restrictions, includes a *flowgraph template*. Thus the structure (loops etc.) of the program is fixed and serves as a guide for the procedure.

Thinking about how the field can evolve, one might consider to enhance the interaction between synthesizer and user. If there is information about the problem that the user possesses (without having to work too hard for it), it would be a waste to ignore this information.

Therefore the steps taken in recent publications as in [11] — focusing on an integration of synthesis in the normal programming process — seem very promising.

One could see in this development an analogy to interactive theorem provers. There, the idea is also to automatize as much as possible by tactics, deciding at central points of the proof manually, and letting the tactics do the details of the proof.

Another point which hints towards the decomposition of larger problems into smaller ones is the difficulty of coming up with a correct specification. Sometimes even for smaller problems mistakes can quickly emerge. The only way to find out these kinds of mistakes is to look over the synthesized program. Therefore, rapid feedback to small specifications seems helpful and time-saving.

References

[1] C. Cordell Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240. William Kaufmann, 1969.

[2] Zohar Manna and Richard Waldlnger. A deductive approach to program synthesis. In *IJCAI*, pages 542–551. William Kaufmann, 1979.

[3] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[4] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *Commun. ACM*, 55(2):103–111, 2012.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[6] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In *EMSOFT*, pages 22:1–22:10. IEEE, 2013.

[7] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, chapter Counterexample-Guided Quantifier Instantiation for Synthesis in SMT, pages 198–216. Springer International Publishing, Cham, 2015.

[8] Swen Jacobs, Viktor Kuncak, and Philippe Suter. Reductions for synthesis procedures. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 88–107. Springer, 2013.

[9] Sanjit A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. *CoRR*, abs/1201.0979, 2012.

[10] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224. ACM, 2010.

[11] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *OOPSLA*, pages 407–426. ACM, 2013.

[12] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326. ACM, 2010.