An Update on Deductive Synthesis and Repair in the Leon Tool

Manos Koukoutos   Etienne Kneuss   Viktor Kuncak

EPFL, Switzerland

`firstname.lastname@epfl.ch`

We report our progress in scaling deductive synthesis and repair of recursive functional Scala programs in the Leon tool. We describe new techniques, including a more precise mechanism for encoding the space of meaningful candidate programs. Our techniques increase the scope of synthesis by expanding the space of programs we can synthesize and by reducing the synthesis time in many cases. As a new example, we present a run-length encoding function for a list of values, which Leon can now automatically synthesize from specification consisting of the decoding function and the local minimality property of the encoded value.

1 Introduction

This tool paper presents our recent improvements to deductive synthesis and repair of the Leon tool [12,13]. The tool aims to synthesize (or repair) purely functional programs in a subset of Scala containing mutually recursive functions. The generated code should provably satisfy a specification, which is given by the programmer in the form of function pre- and postconditions [23], as well as (possibly symbolic) input-output examples [12].

Although our system does support interaction in synthesis [11, Page 13], the evaluation in this paper focuses on fully automated synthesis, based on searching a space of applicable rules. We employ a set of deductive synthesis rules, that either decompose a synthesis problem into simpler ones, or, if possible, solve it directly by synthesizing a satisfying solution. The most notable closing rule is Symbolic Term Exploration, which generates symbolic terms based on an expression grammar. The grammar is type-directed and depends on the particular synthesis problem, e.g. it produces constants and variables in scope, as well as calls to available functions.

As demonstrated by our experimental evaluation, our improvements allow the tool to synthesize larger expressions, as well as to synthesize a wider variety of expressions. This was made possible by refinements throughout the synthesis framework and its rules. A notable novelty is a more general notion of program grammars whose non-terminals are equipped with attributes. These attributes enable us to produce certain types of expressions in their normal form only and thus skip other expressions that are syntactically different yet semantically equivalent. We exploit for instance algebraic laws for arithmetic operators. Such refined grammars may thus prove useful for future versions of syntax-guided synthesis format [2].

Related work. Related approaches are based on input/output examples [1,16,18], bounded specifications and sketches [10], as well as types [6, 19], some in combination with weights derived from corpus [7–9, 17, 21]. Frameworks that enable definition of synthesizers have also emerged [3, 20]. By presenting a publicly available snapshot of our system and benchmarks we hope to contribute to establishing a new baseline for recursive program synthesis and repair.

```
def decode[A](l: List[(BigInt, A)]): List[A] =        def encode[A](l : List[A]): List[(BigInt, A)] = {
  l match {                                             l match {
    case Nil() ⇒ Nil()                                    case Nil() ⇒ Nil()
    case Cons((i, x), xs) ⇒                               case Cons(h0, t0) ⇒
      List.fill(i, x) ++ decode(xs) }                       val rec = encode(t0)
                                                            rec match {
                                                              case Nil() ⇒ List((BigInt(1), h0))
def legal[A](l: List[(BigInt, A)]): Boolean =                 case Cons(h1 @ (h1_1, h1_2), t1) ⇒
  l match {                                                     if (h0 == h1_2) {
    case Nil() ⇒ true                                             Cons((h1_1 + BigInt(1), h1_2), t1)
    case Cons((i, _), Nil()) ⇒ i > 0                            } else {
    case Cons((i, x), tl@Cons((_, y), _)) ⇒                       Cons((BigInt(1), h0), Cons(h1, t1))
      i > 0 && x != y && legal(tl) }                          } } }
                                                      } ensuring { res ⇒
def encode[A](l: List[A]): List[(BigInt, A)] =            legal[A](res) && decode[A](res) == l
  choose { res ⇒                                      }
    legal(res) && decode(res) == l }
```

Figure 1: Run-length encoding problem (left) and solution (right)

The topic of deductive synthesis from specifications has been explored actively over the past decades [5, 14]. A key practical question that we aim to address is scalability on program tasks containing recursive functions. Most existing systems require sketches, specifications of the building block operators relevant for a given problem, or definitions of domain-specific languages. Through such additional specification users reduce the search space compared to a more general case that our tool addresses. The SyGuS synthesis competition [2] helps objective evaluation of synthesis tasks thanks to a grammar as an explicit input. The benchmarks we discuss in this paper require more expressive power than the current SyGuS competition categories. We hope that in the future there will be richer categories, and that we will also understand good ways to leverage synthesis in restricted categories to synthesize more complex programs. Here the situation is analogous to verification systems that generate verification conditions in SMT-LIB input format to prove correctness of programs whose full semantics is beyond the abilities of SMT solvers.

1.1   Example

The left column of Figure 1 specifies a run-length encoding algorithm. Consider the encode function specification. It is expressed as a nondeterministic **choose** construct, which our tool will try to convert into a deterministic (executable) program that satisfies the predicate within. This predicate expresses encode as the inverse of the decode function that generates legal run-length encodings. Leon is able to generate and formally verify the solution shown on the right in under 20 seconds. It is one of the very few tools that can solve similar problems with this level of automation.

The closest to the refinement of our 2013 approach that we here present is the recent SYNQUID system [19] that can synthesize both encode and decode functions from a specification based on liquid types in a very short amount of time. The version of the corresponding benchmark [1] that was pointed to us, however, explicitly lists the zero constant, successor, and predecessor function as the only primitive building blocks for arithmetic expressions. In contrast, our system explores trees that, in addition to

---

[1] http://comcom.csail.mit.edu/demos/#run-length

constants, contain general binary arithmetic operations including addition and substraction operators. As a result, our search space is notably larger. In our attempts, adding components corresponding to our search space made the SYNQUID web example timeout after the 120 second limit. Given that we are not experts in using SYNQUID, a more systematic comparison remains to be done in the future.

1.2 Basic synthesis notation

We will repeat a very brief overview of basic synthesis notation as given in [12]. For more details, see [13].

A synthesis problem is written as $[\![\bar{a} \ \langle \Pi \triangleright \phi \rangle \ \bar{x}]\!]$, where $\bar{a}$ are the input variables of the problem, Π is the current path condition, ϕ is the problem specification and $\bar{x}$ are the output variables. Π is a function of $\bar{a}$, whereas ϕ of both $\bar{a}$ and $\bar{x}$. A solution to a synthesis problem is a pair $\langle P \mid T \rangle$, where T is the program term generated by synthesis, and P is a precondition under which T is a valid solution. We illustrate the notation for decomposition rules with a rule for splitting a problem containing a top-level disjunction:

$$\frac{[\![\bar{a} \ \langle \Pi \triangleright \phi_1 \rangle \ \bar{x}]\!] \vdash \langle P_1 \mid T_1 \rangle \qquad [\![\bar{a} \ \langle \Pi \triangleright \phi_2 \rangle \ \bar{x}]\!] \vdash \langle P_2 \mid T_2 \rangle}{[\![\bar{a} \ \langle \Pi \triangleright \phi_1 \vee \phi_2 \rangle \ \bar{x}]\!] \vdash \langle P_1 \vee P_2 \mid \mathsf{if}(P_1) \ \{T_1\} \ \mathsf{else} \ \{T_2\} \rangle}$$

This rule should be interpreted as follows: from an input synthesis problem $[\![\bar{a} \ \langle \Pi \triangleright \phi_1 \vee \phi_2 \rangle \ \bar{x}]\!]$, the rule decomposes it in two subproblems: $[\![\bar{a} \ \langle \Pi \triangleright \phi_1 \rangle \ \bar{x}]\!]$ and $[\![\bar{a} \ \langle \Pi \triangleright \phi_2 \rangle \ \bar{x}]\!]$. Given corresponding solutions $\langle P_1 \mid T_1 \rangle$ and $\langle P_2 \mid T_2 \rangle$, the rule solves the initial problem with $\langle P_1 \vee P_2 \mid \mathsf{if}(P_1) \ \{T_1\} \ \mathsf{else} \ \{T_2\} \rangle$.

2 Recursive calls

A key feature of Leon's synthesis is the ability to synthesize programs with recursive function calls.

In [12] we present a method to introduce recursive calls that have good chance of not introducing non-termination. Let us say we are trying to synthesize a function foo with formal arguments $\bar{a}$. Leon would track these arguments with a construct $\Downarrow[foo(\bar{a})]$. The arguments $\bar{a}$ would then be transformed as needed by various decomposition rules. When Symbolic Term Exploration is invoked, it will look for the current $\Downarrow[foo(\bar{a}')]$ construct, and introduce recursive calls to foo such that at least one argument is smaller than initially, for a type-dependent definition of "smaller". The rest of the arguments would be left free to be generated by symbolic term exploration.

Our new deployment of synthesis in Leon changes this approach, and instead uses a dedicated deductive rule which introduces recursive calls to the synthesis context. As before, it forces one argument of the function to be smaller, but the rest of the arguments are fixed. This rule replaces all rules which introduced induction in any way.

Let $(\Pi \wedge a \leftarrow e)$ bind a fresh variable a to the value e in the path condition Π of a problem. Then the rule can be formally written as follows:

INTRODUCE REC. CALLS
$$\frac{[\![\bar{a} \ \langle \Pi \wedge rec \leftarrow foo(a_1, \ldots, a_i', \ldots, a_n) \triangleright \phi_1 \rangle \ \bar{x}]\!] \vdash \langle P_1 \mid T_1 \rangle \qquad a_i' \in argsSmaller(a_i, \Pi)}{[\![\bar{a} \ \langle \Downarrow[foo(a_1, \ldots, a_i, \ldots, a_n)] \wedge \Pi \triangleright \phi_1 \rangle \ \bar{x}]\!] \vdash \langle P_1 \mid T_1 \rangle}$$

To define *argsSmaller*, let us consider an abstract class type *AC* with a concrete descendant *CC*, and

let *F* be the fields of *CC*. Then

$$
\begin{aligned}
argsSmaller(i : Int, && i > 0 \wedge \Pi) &= \{i - 1\} \\
argsSmaller(i : Int, && i < 0 \wedge \Pi) &= \{i + 1\} \\
argsSmaller(i : BigInt, && i > 0 \wedge \Pi) &= \{i - 1\} \\
argsSmaller(i : BigInt, && i < 0 \wedge \Pi) &= \{i + 1\} \\
argsSmaller(c : AC, && c : CC \wedge \Pi) &= \{c.f \ \cup \ argsSmaller(c.f, \Pi) \mid f \in F \wedge f : AC\} \\
argsSmaller(v, && \Pi) &= \emptyset \qquad \text{otherwise}
\end{aligned}
$$

This approach cannot generate recursive calls where more than one argument changes; for example, it cannot generate a recursive call to a function which updates an accumulator while traversing a data structure. However, it has the benefit that the variable *rec*, which is bound to the result of the recursive call, is now available to further decomposition rules. This allows for new forms of programs to be synthesized. For example, see the 6th line of the solution in Figure 1: the introduced variable *rec* is pattern-matched on by another rule, which is necessary to solve the particular benchmark.

3   Term Grammars

The main terminal rule of our framework, Symbolic Term Exploration, generates symbolic terms with a context-free grammar. The grammar takes into account the context of the current problem: for example, it generates expressions containing variables in scope, as well as calls to available functions.

Our previous implementation of expression grammars simply used types as nonterminal symbols. For example, a grammar for integers could be

$$
Int ::= Int + Int \mid Int \text{ - } Int \mid \mathbf{0} \mid \mathbf{a} \mid \mathbf{foo(} \, Bool \, \mathbf{)}
$$

where a: Int is a parameter of the function under synthesis and foo: Bool $\Rightarrow$ Int is a function in scope.

However, such simple expression grammars have the disadvantage of generating too many redundant terms. One reason for that is that they are highly ambiguous. For example, the above grammar would generate the term a + a + foo(**true**) in two different ways. The other reason is that even syntactically distinct generated expressions are very often semantically equivalent; in our example, consider a + foo(**true**) versus foo(**true**) + a, or a versus a + 0 versus a + 0 + 0.

Our current work addresses these issues by using a richer representation of grammars. Nonterminal symbols are enhanced with additional information beyond the type; we refer to this contextual information as *attributes*. Attributes refine and filter production rules of an existing grammar and enable us to fine-tune the shape of the expression terms they represent.

In [18], the authors apply a similar disambiguation technique. In their case, the disambiguation happens after the terms have been generated. In contrast, our attributes affect the grammar itself, meaning that all terms produced are automatically good candidates. The bottom-up term generation technique used in TRANSIT [24] merges even more equivalent expressions thanks to its evaluation-based under-approximation of expression equivalence.

We now describe several of the attributes defined in Leon and how they affect the grammar productions. A generic production rule can be written as

$$
T ::= \mathbf{f} \, (\, T_1, T_2, \ldots, T_n \,) \tag{1}
$$

where **f** is a function of the nonterminal symbols on the right-hand side of the rule. The nonterminals *T* and T_i may be plain types, or may already be annotated with attributes. We represent attributes associated with a nonterminal in braces.

Size and commutative operators. Iteratively generating bigger terms can be done by gradually increasing the unfolding depth of the grammar. This however causes the number of terms per depth to explode double exponentially. Instead, we use a *Sized* attribute that restricts the size of terms produced [11]. For instance, $Int_{\{|5|\}}$ produces only integer expressions of size 5, such as a + b + c.

Starting with a production rule of the form 1 we can get the productions of $T_{\{|s|\}}$ with

$$T_{\{|s|\}} ::= \mathbf{f} \, (\, T_{1\{|s_1|\}}, T_{2\{|s_2|\}}, \ldots, T_{n\{|s_n|\}} \,)$$

for all combinations of $s_i > 0$ such that $\sum s_i = s - size(f)$, where $size(f)$ a cost we associate with f. If $n = 0$, the production is kept only if $size(f) = s$. Additionally, if f is a commutative operator, we require that $\forall i < j. \, s_i \geq s_j$. As a result, only left-heavy terms are produced by the grammar (i.e. (a $*$ b) + c and not the equivalent c + (a $*$ b)).

Associative operators. To remove redundancy caused by operator associativity, we require that all associative operators associate to the left. Let $\mathbf{f}$ be an associative operator and $\neg f$ an attribute that disallows production rules with operator $\mathbf{f}$. Then $T ::= \mathbf{f} \, (\, T_1, T_2 \,)$ becomes $T ::= \mathbf{f} \, (\, T_1, T_{2\{\neg f\}} \,)$, and rules of the form $T_{\{\neg f\}} ::= \mathbf{f} \, (\, T_1, T_2 \,)$ are removed.

Ground Terms. We want to avoid that our grammars generate ground terms for two reasons: firstly, because different combinations of ground terms may end up simplifying to equivalent programs (consider 1 + 3 and 2 + 2); secondly, because our system includes another dedicated rule which is much more efficient than Symbolic Term Exploration in discovering ground terms.

Let the attribute G denote that a ground term is expected, whereas $\neg G$ that a ground term is disallowed. Then

$$T_{\{G\}} ::= \quad \mathbf{f} \, (\, T_{1\{G\}}, T_{2\{G\}}, \ldots, T_{n\{G\}} \,)$$
$$T_{\{\neg G\}} ::= \quad \mathbf{f} \, (\, T_{1\{G_1\}}, T_{2\{G_2\}}, \ldots, T_{n\{G_n\}} \,)$$

for all combinations of $G_i \in \{G, \neg G\}$ such that at least one G_i is $\neg G$. If $\mathbf{c}$ is a constant and $\mathbf{v}$ is a variable, rules of the forms $T_{\{\neg G\}} ::= \mathbf{c}$ and $T_{\{G\}} ::= \mathbf{v}$ are removed.

The $\neg G$ attribute is attached to the top-level symbol of the grammar.

Neutral elements. Several arithmetic operators have so-called neutral or absorbing operands that are sources of redundancies. For example, terms such as e + 0, e / 1, or e $*$ 0 are all equivalent to a shorter form. We eliminate these from the grammar by using an attribute that excludes neutral elements: $Int ::= Int + Int$ becomes $Int ::= Int_{\{\neg 0\}} + Int_{\{\neg 0\}}$. Naturally, the production $Int ::= \mathbf{0}$ is excluded from $Int_{\{\neg 0\}}$.

4   Symbolic Term Exploration

The main closing rule that our tool employs is Symbolic Term Exploration. Although the algorithm has not changed much conceptually since previously presented in [13], our implementation has matured as we gained experience using it. In this section we document our current design choices. Additionally, we provide detailed pseudocode for our approach, hoping it will serve as a starting point for similar implementations.

Symbolic Term Exploration (STE) unfolds a grammar as described in the Section 3 to create a set of candidate programs, which are represented all together with a symbolic program tree [13]. These programs are first filtered with concrete execution based on a set of tests. The ones that survive concrete

testing have to be handled symbolically with the aid of a Leon solver [22, 23], which reduces verification of a subset of Scala to a stream of queries for SMT solvers (currently Z3 [15] and CVC4 [4]).

Our synthesis tool uses two approaches to determine if the representation of a set of programs contains a valid program: (1) if the number of remaining programs is relatively small, we try to prove or disprove each one separately with the Leon solver. If a program is proven correct, we have a satisfactory solution; otherwise, the solver generates a fresh counterexample which we add to our test base, as it may help exclude further programs. (2) if the number of remaining programs is large, we query the solver for a program that satisfies the specification for at least one input. If no such program exists, then no program in our candidate set satisfies the specification, and STE fails; otherwise, we hope that this program has good chances to be a satisfactory solution, and we try to prove it valid as in (1).

Our experience using STE since the previous iterations of the tool has shown that, even for thousands of programs and tens of tests, concrete execution is usually faster than a single SMT query. An explanation is that many of our tests are generated by automatic data generators, and they tend to be quite small (small numeric values or data structures of few nodes). For that reason, we have adapted our implementation to rely as much as possible on concrete execution with the following adjustments:

- We concretely test candidate programs against every counterexample as soon as it is discovered by the Leon solver.

- We make sure to utilize parts of the solution that have already been discovered by other deductive rules [11]. From these parts we construct a partial solution, with a placeholder in place of the current problem. For example, consider we are trying to solve the example of Figure 1. After case-splitting on l and solving the Nil case, the partial solution would be

  ```
  l match {
    case Nil() ⇒ Nil()
    case Cons(h0, t0) ⇒ ???
  }
  ```

  This expression will temporarily replace the original implementation of encode (the **choose**) in the program. During concrete execution and validation, the placeholder ??? is set each time to the program we are currently testing. Similarly, during the discovery of a tentative program, the placeholder is set to the tree representation of the set of available programs.

- Another simple heuristic we introduced that yields good results in practice is to sort available tests according to the number of programs that failed on them. This way, tests that have been more successful in the past in excluding programs get executed first.

Algorithm 1 shows an overview of our current STE implementation. The main function, STE, takes as arguments a synthesis problem, an expression grammar G and a desired maximum size of generated programs. It uses auxiliary functions VALIDATE and CONCRETETEST shown below, as well as two global variables I and P.

5   Evaluation

We evaluated the improvements presented in the previous sections against the previous version of Leon and we present the results in Table 1.

The first column of Table 1 gives an indication of the difficulty of each benchmark: *Prog* indicates the total size of the program in AST nodes, and *Sol* indicates the size of the solution generated by the

---

Algorithm 1 Symbolic Term Exploration

---

  var I = Initial list of examples
  var $P = \emptyset$ ▷ Set of examples represented as a tree

  function STE($[\![\bar{a} \ \langle \Pi \rhd \phi \rangle \ \bar{x}]\!]$, G, *maxSize*)
    for $n \leftarrow 1$ **to** *maxSize* **do**   ▷ STE loop over program sizes
      P = UNFOLD(G, n)   ▷ Generate programs of size n from grammar G
      CONCRETETEST(I, ϕ)   ▷ Exclude programs by concrete execution
      while $P \neq \emptyset$ **do**   ▷ Main STE Loop
        if $|P|$ sufficiently reduced **then**
          for all $p \in P$ **do**   ▷ Try to validate individually
            if VALIDATE($[\![\bar{a} \ \langle \Pi \rhd \phi \rangle \ \bar{x}]\!]$, p) **then return** p   ▷ Found valid solution!
        let $f = (\Pi \wedge p \in P \wedge \phi[\bar{x}/p(\bar{a}')])$, $\bar{a}'$ fresh
        if $\neg$LEONSOLVERSAT(f) **then**   ▷ No program of this size works
          Break to the next value of n
        else
          let p_0 = LEONSOLVERMODEL(f)
          if VALIDATE($[\![\bar{a} \ \langle \Pi \rhd \phi \rangle \ \bar{x}]\!]$, p_0) **then**
            return p_0 ▷ Found valid solution!
    return FAIL   ▷ No program found for any program size

  function VALIDATE($[\![\bar{a} \ \langle \Pi \rhd \phi \rangle \ \bar{x}]\!]$, p)   **procedure** CONCRETETEST(J, ϕ)
    let $f = (\Pi \wedge \neg\phi[\bar{x}/p(\bar{a})])$ **for all** $p \in P$ **do**
    if $\neg$LEONSOLVERSAT(f) **then return** true   **for all** $\bar{a}_0 \in J$ **do**
    else   **if** $\neg$EXECUTE($\phi[\bar{x}/p(\bar{a}_0)]$) **then**
      $P = P \setminus \{p\}$ INCREASEPRIORITY(J, $\bar{a}_0$)
      let a_0 = LEONSOLVERMODEL(f) $P = P \setminus \{p\}$
      CONCRETETEST($\{a_0\}, \phi$)
      $I = I \cup \{a_0\}$
      return false

---

latest version of Leon. For each version of Leon tested and each benchmark, we list the running time as well as whether the tool produced and verified a solution ($\sqrt{}$), failed altogether (X), or produced a solution but could not verify it ($\sqrt{}$ in parentheses). An X means synthesis failed for this benchmark; either the benchmark timed out after 200 seconds, or the synthesizer exhausted its search space without coming up with a solution.

To make the effect of each individual improvement clearer, the following columns of the table showcase the performance of the tool as various features are added. The first column presents the original version of Leon. In that version, we constrained Symbolic Term Exploration to expressions of size up to 5. However, after the latest optimizations we found it is viable to increase the bound to 7. This immediately solves one additional benchmark, but is not viable by itself due to the large slowdowns it introduces. The results for this configuration are presented for completeness under the column *Size = 7*. The next columns introduce respectively the new improved Symbolic Term Exploration (*STE*), the new rule for recursive functions (*Rec*) and finally the optimized term grammars (*TG*).

Operation	Sizes		Prev		Size = 7		STE		Rec		TG	
	Pr	Sol	⊢	🕐	⊢	🕐	⊢	🕐	⊢	🕐	⊢	🕐
`BatchedQueue.enqueue`	92	26	X		(√)	22.8	(√)	18.7	(√)	18.8	(√)	15.4
`List.split`	84	33	X		X		X		√	2.4	√	2.5
`AddressBook.make`	43	36	X		X		X		√	4.2	√	4.0
`RunLength.encode`	118	39	X		X		X		√	18.7	√	20.3
`Diffs.diffs`	63	24	X		X		X		√	24.5	√	11.8
`List.insert`	61	3	√	0.9	√	0.6	√	0.8	√	0.8	√	0.7
`List.delete`	63	19	√	4.2	√	39.7	√	12.6	√	12.1	√	8.6
`List.union`	77	12	√	7.9	√	13.7	√	3.7	√	3.6	√	2.6
`List.diff`	109	12	√	6.4	√	110.9	√	23.2	√	24.7	√	12.4
`List.listOfSize`	38	11	√	1.4	√	1.7	√	1.1	√	1.6	√	1.4
`SortedList.insert`	94	30	(√)	18.0	(√)	125.0	(√)	17.5	(√)	24.3	(√)	16.0
`SortedList.insertAlways`	108	32	√	22.8	√	139.3	√	32.6	√	35.2	√	21.0
`SortedList.delete`	94	19	(√)	7.6	(√)	57.6	(√)	19.8	(√)	16.8	(√)	15.8
`SortedList.union`	142	12	√	7.5	√	12.7	√	4.1	√	4.5	√	3.3
`SortedList.diff`	140	12	√	5.8	√	104.0	√	12.4	√	13.6	√	6.7
`SortedList.insertionSort`	129	11	√	1.4	√	2.7	√	2.5	√	2.4	√	1.7
`StrictSortedList.insert`	94	30	√	13.4	√	111.5	√	16.7	√	24.9	√	15.8
`StrictSortedList.delete`	94	19	(√)	9.1	(√)	64.9	(√)	22.3	√	19.9	√	15.5
`StrictSortedList.union`	142	12	√	7.7	√	12.8	√	4.3	√	4.6	√	3.1
`UnaryNumerals.add`	46	10	√	4.6	√	4.3	√	3.1	√	3.8	√	3.1
`UnaryNumerals.distinct`	71	4	√	2.2	√	2.1	√	2.0	√	2.2	√	2.0
`UnaryNumerals.mult`	46	11	√	4.6	√	10.7	√	6.9	√	5.6	√	5.7
`BatchedQueue.dequeue`	68	12	(√)	13.0	(√)	10.2	(√)	21.5	(√)	21.3	(√)	17.8
`AddressBook.merge`	104	17	(√)	6.0	(√)	7.4	(√)	18.7	(√)	19.0	(√)	17.6

Table 1: Benchmarks for synthesis

Observations: The new version of the tool was able to solve five new hard benchmarks which were out of scope for the previous versions. It also produces a much more concise, and thus verifiable, solution for an additional benchmark (`StrictSortedList.delete`). Some of the easier benchmarks do present some slowdown. This is mostly caused by failing STE instances that need to exhaust a larger search space. We believe this is a small price to pay: Our focus is to push the limits of what can be synthesized in a reasonable amount of time, rather than to optimize for simpler benchmarks.

Concerning individual improvements, we can see that the STE improvements greatly improve the performance of the tool (compared to the version with the same STE size, of course), without solving additional benchmarks. The improved term grammars also have a significant, if smaller, effect on running times; however, we do expect the improvement to become more significant as our tool scales to larger expression sizes due to the exponential nature of the problem. Finally, the new rule for recursive calls does not improve running times, but extends the search space of the tool and thus solves an additional four benchmarks.

Table 2 displays results for a set of benchmarks for program repair identical to the one presented in [12]. The results here are not so interesting, so we just include the times for the initial and final versions of Leon with all the optimizations. We can see that the benchmarks generally show some delay

Operation	Sizes		Previous		Current	
	Prog	Sol	Test	Repair	Test	Repair
Compiler.desugar	670	3	1.3	3.8	1.0	1.9
Compiler.desugar	668	2	1.1	5.5	0.9	12.3
Compiler.desugar	672	7	1.0	3.4	0.6	1.4
Compiler.desugar	672	7	1.6	3.9	1.1	1.5
Compiler.desugar	672	14	1.8	5.2	1.0	12.8
Compiler.simplify	718	4	1.2	2.7	0.6	1.4
Heap.merge	341	3	4.2	10.3	1.4	2.7
Heap.merge	341	1	2.7	1.8	0.7	1.3
Heap.merge	341	3	4.1	9.8	1.4	2.6
Heap.merge	341	9	3.1	5.5	1.0	2.2
Heap.merge	343	5	3.6	11.5	0.9	2.6
Heap.merge	341	2	3.9	2.0	1.1	13.6
Heap.insert	304	8	10.9	7.6	4.3	1.0
Heap.makeN	343	7	3.8	8.3	2.0	1.3
List.pad	802	8	3.6	7.7	0.7	1.2
List.++	712	3	5.3	6.5	1.9	1.0
List.:+	744	1	9.4	2.1	1.5	1.0
List.replace	746	6	14.4	7.5	1.2	10.4
List.count	799	3	2.2	10.8	0.7	1.3
List.find	799	2	2.6	13.1	2.9	3.5
List.find	801	4	3.4	13.4	2.6	3.6
List.find	802	4	14.1	35.3	4.4	5.2
List.size	748	4	9.9	1.7	1.5	1.0
List.sum	746	4	1.8	2.7	1.1	1.3
List.-	746	1	2.2	11.5	1.1	1.0
List.drop	787	4	1.6	10.2	1.3	16.6
List.drop	787	3	3.1	14.2	X	
Numerical.power	172	5	0.7	5.5	0.2	1.0
Numerical.moddiv	121	3	0.6	1.9	0.2	0.8
MergeSort.split	228	5	3.2	3.0	1.8	2.8
MergeSort.merge	230	7	3.0	2.5	1.3	1.2
MergeSort.merge	230	3	3.2	5.1	1.2	1.7
MergeSort.merge	228	5	2.7	2.6	1.1	1.2
MergeSort.merge	230	1	3.0	1.5	1.4	20.9

Table 2: Benchmarks for repair

relatively to the previous version. This is mostly due to changes that increase the system's reliability, with some cost in performance (more robust nondeterministic evaluator, support for multiple synthesis solutions etc.) Additionally, one benchmark is not solvable any more due to the change in handling of recursive functions. However, we could arguably revert to grammar-generated recursive calls for repair benchmarks, as required recursive calls are likely to be present in the program already (of course, they would be subject to repair).

The version of Leon used for evaluation can be found at `https://github.com/epfl-lara/leon/tree/synt2016`, and the benchmarks at `https://github.com/epfl-lara/leon/tree/synt2016/testcases/synt2016`.

References

[1] Aws Albarghouthi, Sumit Gulwani & Zachary Kincaid (2013): *Recursive Program Synthesis*. In Natasha Sharygina & Helmut Veith, editors: *CAV*, *LNCS* 8044, Springer, pp. 934–950, doi:10.1007/978-3-642-39799-8_67.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2013): *Syntax-guided synthesis*. In: *FMCAD*, IEEE, pp. 1–8.

[3] James Bornholt, Emina Torlak, Dan Grossman & Luis Ceze (2016): *Optimizing synthesis with metasketches*. In Rastislav Bodík & Rupak Majumdar, editors: *POPL*, ACM, pp. 775–788, doi:10.1145/2837614.2837666.

[4] Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett & Cesare Tinelli (2014): *A tour of CVC4: How it works, and how to use it*. In: *FMCAD*, IEEE, p. 7, doi:10.1109/FMCAD.2014.6987586.

[5] Pierre Flener (1995): *Logic Program Synthesis from Incomplete Information*. Springer.

[6] Jonathan Frankle, Peter-Michael Osera, David Walker & Steve Zdancewic (2016): *Example-directed synthesis: a type-theoretic interpretation*. In Rastislav Bodík & Rupak Majumdar, editors: *POPL*, ACM, pp. 802–815, doi:10.1145/2837614.2837629.

[7] Tihomir Gvero & Viktor Kuncak (2015): *Synthesizing Java expressions from free-form queries*. In Jonathan Aldrich & Patrick Eugster, editors: *OOPSLA*, ACM, pp. 416–432, doi:10.1145/2814270.2814295.

[8] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj & Ruzica Piskac (2013): *Complete completion using types and weights*. In Hans-Juergen Boehm & Cormac Flanagan, editors: *PLDI*, ACM, pp. 27–38, doi:10.1145/2462156.2462192.

[9] Tihomir Gvero, Viktor Kuncak & Ruzica Piskac (2011): *Interactive Synthesis of Code Snippets*. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pp. 418–423, doi:10.1007/978-3-642-22110-1_33.

[10] Jeevana Priya Inala, Xiaokang Qiu, Ben Lerner & Armando Solar-Lezama (2015): *Type Assisted Synthesis of Recursive Transformers on Algebraic Data Types*. *CoRR* abs/1507.05527. Available at `http://arxiv.org/abs/1507.05527`.

[11] Etienne Kneuss (2016): *Deductive Synthesis and Repair*. Ph.D. thesis, EPFL, doi:10.5075/epfl-thesis-6878.

[12] Etienne Kneuss, Manos Koukoutos & Viktor Kuncak (2015): *Deductive Program Repair*. In Daniel Kroening & Corina S. Pasareanu, editors: *CAV*, *LNCS* 9207, Springer, pp. 217–233, doi:10.1007/978-3-319-21668-3_13.

[13] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak & Philippe Suter (2013): *Synthesis modulo recursive functions*. In Antony L. Hosking, Patrick Th. Eugster & Cristina V. Lopes, editors: *OOPSLA*, ACM, pp. 407–426, doi:10.1145/2509136.2509555.

[14] Zohar Manna & Richard J. Waldinger (1980): *A Deductive Approach to Program Synthesis*. *ACM Trans. Program. Lang. Syst.* 2(1), pp. 90–121, doi:10.1145/357084.357090.

[15] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[16] Yu Pei, Carlo A. Furia, Martín Nordio & Bertrand Meyer (2015): *Automated Program Repair in an Integrated Development Environment*. In Antonia Bertolino, Gerardo Canfora & Sebastian G. Elbaum, editors: *ICSE*, IEEE Computer Society, pp. 681–684, doi:10.1109/ICSE.2015.222.

[17] Daniel Perelman, Sumit Gulwani, Thomas Ball & Dan Grossman (2012): *Type-directed completion of partial expressions*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pp. 275–286, doi:10.1145/2254064.2254098.

[18] Daniel Perelman, Sumit Gulwani, Dan Grossman & Peter Provost (2014): *Test-driven synthesis*. In Michael F. P. O'Boyle & Keshav Pingali, editors: *PLDI*, ACM, p. 43, doi:10.1145/2594291.2594297.

[19] Nadia Polikarpova, Ivan Kuraj & Armando Solar-Lezama (2016): *Program synthesis from polymorphic refinement types*. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pp. 522–538, doi:10.1145/2908080.2908093.

[20] Oleksandr Polozov & Sumit Gulwani (2015): *FlashMeta: a framework for inductive program synthesis*. In Jonathan Aldrich & Patrick Eugster, editors: *OOPSLA*, ACM, pp. 107–126, doi:10.1145/2814270.2814310.

[21] Alex Reinking & Ruzica Piskac (2015): *A Type-Directed Approach to Program Repair*. In Daniel Kroening & Corina S. Pasareanu, editors: *CAV*, *LNCS* 9206, Springer, pp. 511–517, doi:10.1007/978-3-319-21690-4_35.

[22] Philippe Suter (2012): *Programming with Specifications*. Ph.D. thesis, EPFL, doi:10.5075/epfl-thesis-5581.

[23] Philippe Suter, Ali Sinan Köksal & Viktor Kuncak (2011): *Satisfiability Modulo Recursive Programs*. In Eran Yahav, editor: *SAS*, *LNCS* 6887, Springer, pp. 298–315, doi:10.1007/978-3-642-23702-7_23.

[24] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin & Rajeev Alur (2013): *TRANSIT: specifying protocols with concolic snippets*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pp. 287–296, doi:10.1145/2462156.2462174.