Synthesis Modulo Recursive Functions

Etienne Kneuss[1] Viktor Kuncak[1] Ivan Kuraj[1] Philippe Suter[1,2]

[1]École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2]IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{firstname.lastname}@epfl.ch, psuter@us.ibm.com

Abstract

We describe techniques for synthesis and verification of recursive functional programs over unbounded domains. Our techniques build on top of an algorithm for satisfiability modulo recursive functions, a framework for deductive synthesis, and complete synthesis procedures for algebraic data types. We present new counterexample-guided algorithms for constructing verified programs. We have implemented these algorithms in an integrated environment for interactive verification and synthesis from relational specifications. Our system was able to synthesize a number of useful recursive functions that manipulate unbounded numbers and data structures.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords software synthesis; inductive learning; satisfiability modulo theories

1. Introduction

Software construction is a difficult problem-solving activity. It remains a largely manual effort today, despite significant progress in software development environments and tools. The development becomes even more difficult when the goal is to deliver *verified* software, which must satisfy specifications such as assertions, pre-conditions, and post-conditions. The thesis of this paper is that the development of verified software can be helped through tools that add synthesis techniques on top of a system with automated verification and error-finding capabilities.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509555

Whereas the construction of arbitrarily complex verified software is possible in principle, verifying programs after they have been developed is extremely time-consuming [21, 27] and it is difficult to argue that it is cost-effective. Our research therefore explores approaches that support *integrated software construction and verification*. An important aspect of such approaches are modular verification techniques which can check that a function conforms to its local specification. In such an approach, the verification of an individual function against its specification can start before the entire software system is completed and the resulting verification tasks are partitioned into small pieces. As a result, tools can provide rapid feedback that allows specifications and implementations to be developed simultaneously. Based on such philosophy of continuous rapid feedback, we have developed Leon, a verifier that quickly detects errors in functional programs and reports concrete counterexamples, yet can also prove the correctness of programs [4, 46–48]. We have integrated Leon into a web-browser-based IDE, resulting in a tool for convenient development of verified programs [4]. This verifier is the starting point of the tool we present in this paper.

Moving beyond verification, we believe that the development of verified software can benefit from techniques for *synthesis* from specifications. Specifications in terms of relational properties generalize existing declarative programming language paradigms by allowing the statement of *constraints* between inputs and outputs [16, 22] as opposed to always specifying outputs as functions from input to outputs. Unlike deterministic implementations, constraints can be composed using conjunctions, which enables description of the problem as a combination of orthogonal requirements.

This paper introduces synthesis algorithms, techniques and tools that integrate synthesis into the development process for functional programs. We present a synthesizer that can construct the bodies of functions starting solely from their contracts. The programs that our synthesizer produces typically manipulate unbounded data types, such as algebraic data types and unbounded integers. Thanks to the use of deductive synthesis and the availability of a verifier, when the synthesizer succeeds with a verified output, the generated code is correct for all possible input values.

Our synthesizer uses specifications as the description of the synthesis problems. While it can additionally accept input/output examples to illustrate the desired functionality, we view such illustrations as a special form of input/output relation: whereas input/output examples correspond to tests and provide a description of a finite portion of the desired functionality, we primarily focus on symbolic descriptions, which ensure the desired behavior over an arbitrarily large or even infinite domain. From such descriptions, our synthesizer can automatically generate input/output examples when needed, but can also directly transform specifications into executable code.

A notable degree of automation in our synthesizer comes from synthesis procedures [15, 23, 24], which compile specification fragments expressed in decidable logics. The present work is, in fact, the first implementation of the synthesis procedure for algebraic data types we previously developed [46].

Note however, that, to capture a variety of scenarios in software development, we also support the general problem of synthesis from specifications expressed in a Turing-complete language. We achieve this using a framework for cost-guided application of deductive synthesis rules, which decompose the problems into subproblems.

We have integrated our synthesizer into Leon, where it tightly cooperates with the underlying verifier, allowing it to achieve orders of magnitude better performance than using simpler generate-and-test approaches. Techniques we use include symbolic transformation based on synthesis procedures, as well as synthesis of recursive functions using counterexample-guided strategies. We have evaluated a number of system architectures and trade-offs between symbolic and concrete reasoning in our implementation and arrived at an implementation that appears successful, despite the large space of possible programs. We thus believe we have achieved a new level of automation for a broad domain of recursive functional programs. We consider as a particular strength of our system that it can synthesize code that satisfies a given relational specification for all values of inputs, and not only given input/output pairs.

Despite aiming at a high automation level, we are aware that any general-purpose automated synthesis procedure will ultimately face limitations: the user may wish to synthesize larger code than the scalability of automated synthesis allows, or they may wish to control the structure and not only the observational behavior of the code to be constructed. We therefore deploy the synthesis algorithm as an interactive assistance that allows the developer to interleave manual and automated development steps. In our system, the developer can decompose a function and leave the subcomponents to the synthesizer, or, conversely, the synthesizer can decompose the problem, solve some of the subproblems, and leave the remaining open cases for the developer. To facilitate such synergy, we deploy an anytime synthesis procedure, which maintains a ranked list of current problem decompositions. The user can interrupt the synthesizer at any time to display the current solution and continue manual development.

1.1 Contributions

The overall contribution of this paper is an integrated synthesis and development system for automated and interactive development of verified programs. A number of techniques from deductive and inductive reasoning need to come together to make such system usable.

Implemented synthesis framework. We developed a deductive synthesis framework that can accept a given set of synthesis rules and apply them according to a cost function. The framework accepts 1) a path condition that encodes program context, and 2) a relational specification. It returns the function from inputs to outputs as a solution, as well as any necessary strengthening of the precondition needed for the function to satisfy the specification. We have deployed the framework in a web-browser-based environment with continuous compilation and the ability to interrupt the synthesis to obtain a partial solution in the form of a new program with a possibly simpler synthesis problem.

Within the above framework we have implemented rules for synthesis of algebraic data type equations and disequations [46], as well as a number of general rules for decomposing specifications based on their logical structure or case splits on commonly useful conditions.

Support for recursion schemas and symbolic term generators. One of the main strengths in our framework is a new form of counterexample-guided synthesis that arises from a combination of several rules.

- A set of built-in recursion schemas can solve a problem by generating a fresh recursive function. To ensure well-foundedness we have extended our verifier with termination checking, and therefore generate only terminating function calls using this rule.
- To generate bodies of functions, we have symbolic term generators that systematically generate well-typed programs built from a selected set of operators (such as algebraic data type constructors and selectors). To test candidate terms against specifications we use the Leon verifier. To speed up this search, the rule accumulates previously found counterexamples. Moreover, to quickly bootstrap the set of examples it uses systematic generators that can enumerate in a fair way any finite prefix of a countable set of structured values. The falsification of generated bodies is done by direct execution of code. For this purpose, we have developed a lightweight bytecode compiler for our functional implementation language (a subset of Scala), allowing us to use code execution as a component of counterexample search in the constraint solver.

Function generation by condition abduction. We also present and evaluate a new method, implemented as a rule

in our framework, for synthesis of recursive functions, with the following properties.

- The most distinctive aspect of this rule is the handling of conditional expressions. The rule synthesizes expressions by collecting relevant terms that satisfy a notable number of derived test inputs, and then synthesizing predicates that imply the correctness of candidate terms. This is an alternative to relying on splitting rules to eagerly split the specification according to simple commonly found conditions. Effectively, the additional rule performs abduction of conditions until it covers the entire input space with a partition of conditions, where each partition is associated with a term.
- Instead of specialized term evaluators, the rule uses a general expression enumerator based on generating all expressions of a given type [12]. This results in a broad coverage of expressions that the rule can synthesize. The rule uses a new lazy enumeration algorithm for such expressions with polynomial-time access to the next term to enumerate [26]. It filters well-typed expressions using counterexamples generated from specifications and previous function candidates, as well as from structured value generators.

Evaluation. We evaluate the current reach of our synthesizer in fully automated mode by synthesizing recursive functions on nested algebraic data types, including those that perform operations on lists defined using a general mechanism for algebraic data types, as well as on other custom data types (e.g. trees) defined by the user. This paper presents a description of all the above techniques and a snapshot of our results. We believe that the individual techniques are interesting by themselves, but we also believe that having a system that combines them is essential to understand the potential of these techniques in addressing the difficult problem as synthesis. To gain full experience of the feeling of such a development process, we therefore invite readers to explore the system themselves at

`http://lara.epfl.ch/w/leon`

2. Examples of Synthesis in Leon

We start by illustrating through a series of examples how developers use our system to write programs that are correct by construction. We first illustrate our system and the nature of interaction with it through familiar operations on (sorted) user-defined lists, reflecting along the way on the usefulness of programming with specifications. We then present a somewhat more complex example to illustrate the reach of automated synthesis steps in our system.

2.1 List Manipulation

We start by showing how our system behaves when synthesizing operations on lists. The developer partially specifies lists using their effect on the set of elements. As shown in Figure 1 we start from a standard recursive definition of lists along with recursive functions computing their size as a non-negative integer, and their content as a set of integers.

```
sealed abstract class List
case class Cons(head: Int, tail: List) extends List
case object Nil extends List

def size(lst : List) : Int = lst match {
  case Nil ⇒ 0
  case Cons(_,rest) ⇒ 1 + size(rest) } ensuring(_ ≥ 0)

def content(lst : List) : Set[Int] = lst match {
  case Nil ⇒ Set.empty
  case Cons(e,rest) ⇒ Set(i) ++ content(rest) }
```

Figure 1. User-defined list structure with the usual size and content abstraction functions. Here and throughout the paper, the content abstraction computes a *set* of elements, but it can easily be extended to handle multisets (bags) using the same techniques [8] if stronger contracts are desired

Splitting a list. We first consider the task of synthesizing the split function as used in, e.g., merge sort. As a first attempt to synthesize split, the developer may wish to try the following specification:

```
def split(lst : List) : (List,List) = choose { (r : (List,List)) ⇒
  content(lst) == content(r._1) ++ content(r._2)
}
```

Because it tends to generate simpler solutions before more complex ones, Leon here instantly generates the following function:

```
def split(lst : List) : (List,List) = (lst, Nil)
```

Although it satisfies the contract, it is not particularly useful. This does show the difficult in using specifications, but the advantage of a synthesizer like ours is that it allows the developer to quickly refine the specification and obtain a more desirable solution. To avoid getting a single list together with an empty one, the developer refines the specification by enforcing that the sizes of the resulting lists should not differ by more than one:

```
def split(lst : List) : (List,List) = choose { (r : (List,List)) ⇒
  content(lst) == content(r._1) ++ content(r._2)
  && abs(size(r._1)−size(r._2)) ≤ 1
}
```

Again, Leon instantly generates a correct, useless, program:

```
def split(lst : List) : (List,List) = (lst, lst)
```

We can refine the specification by stating that the *sum* of the sizes of the two lists should match the size of the input one:

```
def split(lst : List) : (List,List) = choose { (r : (List,List)) ⇒
  content(lst) == content(r._1) ++ content(r._2)
  && abs(size(r._1) − size(r._2)) ≤ 1
```

```
def isSorted(lst : lst) : Boolean = lst match {
  case Nil ⇒ true
  case Cons(_, Nil) ⇒ true
  case Cons(x1, xs @ Cons(x2, _)) ⇒ x1 ≤ x2 && isSorted(xs) }

def insertSorted(lst : List, v: Int): List = {
  require(isSorted(lst))
  choose { (r: List) ⇒
    isSorted(r) && content(r) == content(lst) ++ Set(v) } }

def sort(lst : List): List = choose { (r: List) ⇒
  isSorted(r) && content(r) == content(lst) }
```

Figure 2. Specification of sorting suitable for insertion sort

```
def insertSorted(lst: List, v: Int): List = {
  require(isSorted(lst))
  lst match {
    case Nil ⇒ Cons(v, Nil)
    case Cons(h, tail) ⇒
      val r = insertSorted(t, v)
      if (v > h) Cons(h, r)
      else if (h == v) r
      else Cons(v, Cons(h, t)) }

def sort(lst : List): List = lst match {
  case Nil ⇒ Nil
  case Cons(h, t) ⇒ insertSorted(sort(t), h) }
```

Figure 3. Synthesized insertion sort for Figure 2

```
  && (size(r._1) + size(r._2)) == size(lst)
}
```

We then finally obtain a useful split function:

```
def split(lst: List): (List, List) = lst match {
  case Nil ⇒ (Nil, Nil)
  case Cons(h, Nil) ⇒ (Nil, Cons(h, Nil))
  case Cons(h1, Cons(h2, t2)) ⇒
    val r = split(t2)
    (Cons(h1, r._1), Cons(h2, r._2)) }
```

We observe that in this programming style, users can write (or generate) code by conjoining orthogonal requirements, such as constraints on the sizes and contents, which are only indirectly related. The rapid feedback makes it possible to go through multiple candidates rapidly, strengthening the specification as required.

We believe this rapid feedback is mandatory when developing from specifications. One reason is that, since contracts are typically partial, results obtained from under-specifications can be remote from the desired output. Thus, a desirable strategy is to rapidly iterate and refine specifications until the output matches the expectations.

Insertion sort. Sorting is an example often used to illustrate declarative descriptions of problems. We therefore continue this overview of Leon's synthesis capabilities by showing how it synthesizes an implementation of several sorting algorithms, starting from insertion sort. Figure 2 shows the specification of the problem. From this, Leon generates the solution in Figure 3 within seconds and without further hints.

Advantages of Specifications. Comparing Figure 2 and Figure 3, which have similar size, the reader may wonder what we have gained by using specifications instead of implementations. Whereas only widespread use of synthesis systems will give the true answer, we anticipate at least three reasons (with 3. partly following from 2.):

1. **flexibility**: by supporting synthesis from specifications, we do not eliminate the ability to directly write implementations when this is more desirable, but rather add the freedom and the expressive power to describe problems in additional ways that may be appropriate; the new mechanism does not harm performance of readability when not used;
2. **narrower gap between requirements and software**: natural language and mathematical descriptions of structures often have the form of *conjunctions* that more directly map to **choose** constructs than to recursive functions that compute the precise objects. We view the sorting process as one of the many possible ways of obtaining a collection that has (1) the same elements and (2) is sorted, as opposed to thinking a particular sorting algorithm.
3. **reusability when introducing new operations**: once we specify key invariants and abstraction functions, we can reuse them to define new versions of these operations; a related concept is the ability to express orthogonal requirements independently [14].

We next illustrate the last point using examples of reusability as we add new operations: synthesizing removal from a sorted list given the specification for insertion, and synthesizing merge sort given a specification for sort.

Removal and merge for sorted lists. Suppose that, after synthesizing insertion into a sorted list, the developer now wishes to specify removal and merge of two sorted lists. Figure 4 shows the specification of these operations. Note that, once we have gone though the process of defining the invariant for what a sorted list means using function isSorted in Figure 2, to specify these two new operations we only need to write the concise specification in Figure 4. The system then automatically synthesizes the full implementations in Figure 5. We expect that the pay-off from such re-use grows as the complexity of structures increases.

Merge sort. Suppose now that the developer wishes to ensure that the system, given sorting specification, synthesizes merge sort instead of insertion sort. To do this, the developer may then, instead of the insertSorted function, try to introduce the function merge into the scope. In our current version of the system, Leon then synthesizes the following code in less than ten seconds:

```
def delete(in1: List, v: Int) = {
  require(isSorted(in1))
  choose { (out : List) ⇒ isSorted(out) &&
      (content(out) == content(in1) -- Set(v)) } }

def merge(in1: List, in2: List) = {
  require(isSorted(in1) && isSorted(in2))
  choose { (out : List) ⇒ isSorted(out) &&
      (content(out) == content(in1) ++ content(in2)) } }
```

Figure 4. Specification of removal from a sorted list.

```
def delete(in1: List, v: Int): List = {
  require(isSorted(in1))
  in1 match {
    case Nil ⇒ Nil
    case Cons(h, t) ⇒ {
      if (v == h) delete(t, v)
      else Cons(h, delete(t, v)) }}
} ensuring {(out : List) ⇒ isSorted(out) &&
              (content(out) == content(in1) -- Set(v))}

def merge(in1: List, in2: List): List = {
  require(isSorted(in1) && isSorted(in2))
  in1 match {
    case Nil ⇒ Nil
    case Cons(h, t) ⇒ union(t, insertSorted(in2, h)) }
} ensuring {(out : List) ⇒ isSorted(out) &&
             (content(out) == content(in1) ++ Set(v))}
```

Figure 5. Implementation synthesized for Figure 4

```
def sort(lst : List): List = lst match {
  case lst @ Nil ⇒ lst
  case Cons(h, tail) ⇒ merge(Cons(h, Nil), sort(tail))
}
```

Although a valid synthesis output according to the given contract, the result is actually still an implementation of the insertion sort algorithm, because merge is called on a list that is split in a systematically unbalanced way. Even if the split function we synthesized or implemented before is in the scope, the system may decide not to use it in the generated code.

Interactive synthesis and verified refactoring. In situation such as above, where more control is needed, we allow the developer to refine the code, either though manual edits or by applying synthesis rules in the form of *verified refactoring* steps (such as those around which entire systems were built [6]). For example, using our "long induction rule", which solves the problem by induction on the structure of list with two base cases, with two clicks the user can refine the specification into a recursion pattern, then insert manually a local invocation of split. The result is the following:

```
def sort(lst : List): List = lst match {
  case Nil ⇒
    choose { (res : List) ⇒
      contents(res) == contents(list) && isSorted(res) }
  case Cons(_, Nil) ⇒
    choose { (res : List) ⇒
      contents(res) == contents(list) && isSorted(res) }
  case _ ⇒ {
    val p = split(list)
    choose { (res : List) ⇒
      contents(res) == contents(list) && isSorted(res) }
  }
}
```

From this point, Leon can perform synthesis with automatic search by filling out the implementation of the introduced cases individually, to synthesize the following code, in less than fifteen seconds:

```
def sort(lst : List): List = lst match {
  case Nil ⇒ lst
  case Cons(_, Nil) ⇒ lst
  case _ ⇒ {
    val p = split(list)
    merge(sort(p.fst), sort(p.snd))
  }
}
```

This is a valid code implementing the merge sort algorithm.

2.2 Address Book

We next illustrate automated synthesis of somewhat larger user code from specifications. In this case, the user defines their own structure of addresses with relevant information encapsulated in a separate Scala case class. (Note that we replace definitions that are clear from the context by ellipses. The full code is in available in the Appendix.)

```
case class Info(
  address: Int,
  zipcode: Int,
  local: Boolean
)
case class Address(info: Info, priv: Boolean)
```

Addresses are categorized as private or business, according to a boolean flag, and each address book structure contains two lists, one per each category.

```
sealed abstract class List
case class Cons(a: Address, tail: List) extends List
case object Nil extends List

case class AddressBook(business: List, pers: List)
```

After defining some simple operations and predicates:

```
def content(l: List) : Set[Address] = ...
def size(l: List) : Int = ...

def size(ab: AddressBook): Int =
  size(ab.business) + size(ab.pers)
def isEmpty(ab: AddressBook) = size(ab) == 0
def content(ab: AddressBook) : Set[Address] =
  content(ab.pers) ++ content(ab.business)
```

```
def makeAddressBook(l: List): AddressBook = l match {
  case Nil ⇒ AddressBook(Nil, l)
  case Cons(a,tail) ⇒ if (allPrivate(l)) {
    AddressBook(Nil, l)
  } else if (allPrivate(tail)) {
    AddressBook(Cons(a, Nil), tail)
  } else if (a.priv) {
    AddressBook(makeAddressBook(tail).business,
      Cons(a, makeAddressBook(tail).pers))
  } else {
    AddressBook(Cons(a, makeAddressBook(tail).business),
      makeAddressBook(tail).pers)
  }
}
```

Figure 6. Synthesized AddressBook.make example

and defining an invariant that should hold for each valid address book:

```
def allPrivate(l: List): Boolean = ...
def allBusiness(l: List): Boolean = ...

def invariant(ab: AddressBook) =
  allPrivate(ab.pers) && allBusiness(ab.business)
```

the user can formulate a synthesis problem for constructing an address book from a list of addresses as:

```
def makeAddressBook(l: List): AddressBook =
  choose { (res: AddressBook) ⇒
    size(res) == size(l) && invariant(res)
  }
```

After less than nine seconds our system synthesizes the solution in Figure 6, which has around 50 syntax tree nodes. The solution is not minimal, but is correct; the synthesizer introduced two additional branches due to eager approach to infer branches of interest.

If instead of invoking the synthesizer at this point, the user had decided to introduce two helper functions that add an address to the private and business category of an address book respectively:

```
def addToPers(ab: AddressBook, adr: Address) =
  AddressBook(ab.business, Cons(adr, ab.pers))
def addToBusiness(ab: AddressBook, adr: Address) =
  AddressBook(Cons(adr, ab.business), ab.pers)
```

the synthesizer would have found a more compact solution

```
def makeAddressBook(l: List): AddressBook = l match {
  case Nil ⇒ AddressBook(Nil, l)
  case Cons(a,tail) ⇒ if (a.priv) {
      addToPers(makeAddressBook(tail), a)
    } else {
      addToBusiness(makeAddressBook(tail), a)
    }
}
```

in less than five seconds.

Among other transformations on address books is merging two address books which can be implemented with the help of a merge function similar as defined in previous examples:

```
def merge(l1: List, l2: List): List = ...
```

Leon solves the problem given in the following definition:

```
def mergeAddressBooks(ab1: AddressBook,
ab2: AddressBook) = {
  require(invariant(ab1) && invariant(ab2))
  choose {
    (res: AddressBook) ⇒ invariant(res) &&
      (sizeA(res) == sizeA(ab1) + sizeA(ab2))
  }
}
```

in less than nine seconds while outputting a compact and valid solution:

```
def merge(l1: List, l2: List): List =
  AddressBook(merge(ab1.business, ab2.business),
    merge(ab2.pers, ab1.pers))
```

3. Background: the Verifier within Leon

The results presented in this paper focus on the *synthesis* component of Leon. They rely in important ways on the underlying *verifier* which was the subject of previous work [4, 48]. The language of Leon is a subset of Scala, as illustrated through the examples of Section 2. Besides integers and user-defined recursive data types, Leon supports booleans, sets and maps.

Solver algorithm. At the core of Leon's verifier is an algorithm to reason about formulas that include user-defined recursive functions, such as size, content, and isSorted in Section 2. The algorithm proceeds by iteratively examining longer and longer execution traces through the recursive functions. It alternates between an over-approximation of the executions, where only unsatisfiability results can be trusted, and an under-approximation, where only satisfiability results can be concluded. The status of each approximation is checked using the state-of-the-art SMT solver Z3 from Microsoft Research [7]. The algorithm is a *semi-decision procedure*, meaning that it is theoretically complete for counterexamples: if a formula is satisfiable, Leon will eventually produce a model [48]. Additionally, the algorithm works as a decision procedure for a certain class of formulas [47].

In the past, we have used this core algorithm in the context of verification [48], but also as part of an experiment in providing run-time support for declarative programming using constructs similar to **choose** [22]. We have in both cases found the performance in finding models to be suitable for the task at hand. [1]

[1] We should also note that since the publication of [48], our engineering efforts as well as the progress on Z3 have improved running times by 40%.

Throughout this paper, we will assume the existence of an algorithm for deciding formulas containing arbitrary recursive functions. Whenever completeness is an issue, we will mention it and describe the steps to be taken in case of, e.g. timeout.

Compilation-based evaluator. Another component of Leon on which we rely in this paper is an interpreter based on on-the-fly compilation to the JVM. Function definitions are typically compiled once and for all, and can therefore be optimized by the JIT compiler. This component is used during the search in the core algorithm, to validate models and to sometimes optimistically obtain counterexamples. We use it to quickly reject candidate programs during synthesis (see sections 5 and 6).

Ground term generator. Our system also leverages Leon's generator of ground terms and its associated model finder. Based on a generate-and-test approach, it can generate small models for formulas by rapidly and fairly enumerating values of any type. For instance, enumerating Lists will produce a stream of values Nil(), Cons(0, Nil()), Cons(0, Cons(0, Nil())), Cons(1, Nil()), ...

4. Deductive Synthesis Framework

The approach to synthesis we follow in this paper is to derive programs by a succession of independently validated steps. In this section, we briefly describe the formal reasoning behind these constructive steps and provide some illustrative examples. Whereas an earlier (purely theoretical) version of the framework was presented in [15], the new framework supports the notion of path condition, and is the first time we report on the practical realization of this framework.

4.1 Synthesis Problems

A synthesis problem is given by a predicate describing a desired relation between a set of input and a set of output variables, as well as the context (program point) at which the synthesis problem appears. We represent such a problem as a quadruple

$$[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$$

where:

- $\bar{a}$ denotes the set of *input variables*,
- $\bar{x}$ denotes the set of *output variables*,
- ϕ is the *synthesis predicate*, and
- Π is the *path condition* to the synthesis problem.

The free variables of ϕ must be a subset of $\bar{a} \cup \bar{x}$. The path condition denotes a property that holds for input at the program point where synthesis is to be performed, and the free variables of Π should therefore be a subset of $\bar{a}$.

As an example, consider the following call to **choose**:

```
def f(a : Int) : Int = {
  if(a ≥ 0) {
    choose((x : Int) ⇒ x ≥ 0 && a + x ≤ 5)
  } else ...
}
```

The representation of the corresponding synthesis problem is:

$$[\![a\ \langle a \geq 0 \rhd x \geq 0 \wedge a + x \leq 5 \rangle\ x]\!] \tag{1}$$

4.2 Synthesis Solutions

We represent a solution to a synthesis problem as a pair

$$\langle P \mid \bar{T} \rangle$$

where P is the *precondition*, and $\bar{T}$ is the *program term*. The free variables of both P and $\bar{T}$ must range over $\bar{a}$. The intuition is that, whenever the path condition and the precondition are satisfied, evaluating $\phi[\bar{x} \mapsto \bar{T}]$ should evaluate to true, i.e. $\bar{T}$ are realizers for a solution to $\bar{x}$ in ϕ given the inputs $\bar{a}$. Furthermore, for a solution to be as general as possible, the precondition must be as weak as possible.

Formally, for such a pair to be a solution to a synthesis problem, denoted as

$$[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle$$

the following two properties must hold:

- *Relation refinement:*

  $$\Pi \wedge P \models \phi[\bar{x} \mapsto \bar{T}]$$

  This property states that whenever the path- and precondition hold, the program $\bar{T}$ can be used to generate values for the output variables $\bar{x}$ such that the predicate ϕ is satisfied.

- *Domain preservation:*

  $$\Pi \wedge (\exists \bar{x} :\ \phi) \models P$$

  This property states that the precondition P cannot exclude inputs for which an output would exist such that ϕ is satisfied.

As an example, a valid solution to the synthesis problem (1) is given by:

$$\langle a \leq 5 \mid 0 \rangle$$

The precondition $a \leq 5$ characterizes exactly the input values for which a solution exists, and for all such values, the constant 0 is a valid solution term for x. Note that the solution is in general not unique; alternative solutions for this particular problem include, for example, $\langle a \leq 5 \mid 5 - a \rangle$, or $\langle a \leq 5 \mid \mathsf{if}(a < 5)\ a + 1\ \mathsf{else}\ 0 \rangle$.

A note on path conditions. Strictly speaking, declaring the path condition separately from the precondition does not add expressive power to the representation of synthesis problems: one can easily verify that the space of solution terms for $[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$ is isomorphic to the one for

$[\![\bar{a}\ \langle \mathsf{true} \rhd \Pi \wedge \phi \rangle\ \bar{x}]\!]$. Note however that with the second representation, solving the synthesis problem would always result in a precondition at least as strong as Π, since it appears in the synthesis predicate and does not contain any output variables. Keeping the path condition explicit thus avoids computing this redundant information, an effect we found to be important in our implementation.

4.3 Inference Rules for Synthesis

Building on our correctness criteria for synthesis solutions, we now describe *inference rules* for synthesis. Such rules describe relations between synthesis problems, capturing how some problems can be solved by reduction to others. We have shown in previous work how to design a set of rules to ensure *completeness* of synthesis for a well-specified class of formulas, e.g. integer linear arithmetic relations [23] or simple term algebras [15]. In the interest of remaining self-contained, we shortly describe some generic rules. We then proceed to presenting inference rules which allowed us to derive synthesis solutions to problems that go beyond such decidable domains.

The validity of each rule can be established independently from its instantiations, or from the contexts in which it is used. This in turn guarantees that the programs obtained by successive applications of validated rules are correct by construction.

Generic reductions. As a first example, consider the rule ONE-POINT in Figure 7. It reads as follows; "if the predicate of a synthesis problem contains a top-level atom of the form $x_0 = t$, where x_0 is an output variable not appearing in the term t, then we can solve a simpler problem where t is substituted for x_0, obtain a solution $\langle P \mid \bar{T} \rangle$ and reconstruct a solution for the original one by first computing the value for t and then assigning as the result for x_0".

Conditionals. In order to synthesize programs that include conditional expressions, we need rules such as CASE-SPLIT in Figure 7. The intuition behind CASE-SPLIT is that a disjunction in the synthesis predicate can be handled by an if-then-else expression in the synthesized code, and each subproblem (corresponding to predicates ϕ_1 and ϕ_2 in the rule) can be treated separately. As one would expect, the precondition for the final program is obtained by taking the disjunction of the preconditions for the subproblems. This matches the intuition that the disjunctive predicate should be realizable if and only if one of its disjuncts is. Note as well that even though the disjunction is symmetrical, in the final program we necessarily privilege one branch over the other one. This has the interesting side-effect that we can, as shown in the rule, add the negation of the precondition P_1 to the path condition of the second problem. This has the potential of triggering simplifications in the solution of ϕ_2. An extreme case is when the first precondition is true and the "else" branch becomes unreachable.

The CASE-SPLIT rule as we presented it applies to disjunctions in synthesis predicates. We should note that it is sometimes desirable to explicitly introduce such disjunctions. For example, our system includes rules to introduce branching on the equality of two variables, to perform case analysis on the types of variables (pattern-matching), etc. These rules can be thought of as introducing first a disjunct, e.g. $a = b \vee a \neq b$, then applying CASE-SPLIT.

Recursion schemas. We now show an example of an inference rule that produces a recursive function. A common paradigm in functional programming is to perform a computation by recursively traversing a structure. The rule LIST-REC captures one particular form of such a traversal for the List recursive type used in the examples of Section 2. The goal of the rule is to derive a solution consists of a single invocation to a recursive function rec. The recursive function has the following form:

```
def rec(a0, ā) = {
  require(Π2)
  a0 match {
    case Nil ⇒ T̄1
    case Cons(h, t) ⇒
      lazy val r̄ = rec(t, ā)
      T̄2
  }
} ensuring(r̄ ⇒ φ[x̄ ↦ r̄])
```

where a_0 is of type List. The function iterates over the list a_0 while preserving the rest of the input variables (the environment) $\bar{a}$. Observe that its postcondition corresponds exactly to the synthesis predicate of the original problem. The premises of the rule are as follows.

- The condition $(\Pi_1 \wedge P) \implies \Pi_2$ ensures that the initial call to rec in the final program satisfy its precondition. We can often let $P \equiv \mathsf{true}$ and $\Pi_2 \equiv \Pi_1$.
- The condition $\Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \implies \Pi_2[a_0 \mapsto t]$ states that the precondition of rec should be inductive, i.e. whenever it holds for a list, it should also hold for its tail. This is necessary to ensure that the recursive call will satisfy the precondition.
- The subproblem $[\![\bar{a}\ \langle \Pi_2 \rhd \phi[a_0 \mapsto \mathsf{Nil}] \rangle\ \bar{x}]\!]$ corresponds to the base case (Nil), and thus does not contain the input variable a_0.
- The final subproblem is the most interesting, and corresponds to the case where a_0 is a Cons, represented by the fresh input variables h and t. Because the recursive structure is fixed, we can readily represent the result of the invocation $\mathsf{rec}(t,\bar{a})$ by another fresh variable r. We can assume that the postcondition of rec holds for that particular call, which we represent in the path condition as $\phi[a_0 \mapsto t, \bar{x} \mapsto \bar{r}]$. The rest of the problem is obtained by substituting a_0 for $\mathsf{Cons}(h,t)$ in the path condition and in the synthesis predicate.

$$\textsc{One-point}\ \frac{[\![\bar{a}\ \langle \Pi \rhd \phi[x_0 \mapsto t]\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad x_0 \notin \mathrm{vars}(t)}{[\![\bar{a}\ \langle \Pi \rhd x_0 = t \wedge \phi\rangle\ x_0, \bar{x}]\!] \vdash \langle P \mid \mathsf{val}\ \bar{x} := \bar{T};\ (t, \bar{x})\rangle} \qquad \textsc{Ground}\ \frac{\mathcal{M} \models \phi \qquad \mathrm{vars}(\phi) \cap \bar{a} = \emptyset}{[\![\bar{a}\ \langle \Pi \rhd \phi\rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \mathcal{M}\rangle}$$

$$\textsc{Case-split}\ \frac{[\![\bar{a}\ \langle \Pi \rhd \phi_1\rangle\ \bar{x}]\!] \vdash \langle P_1 \mid \bar{T}_1\rangle \qquad [\![\bar{a}\ \langle \Pi \wedge \neg P_1 \rhd \phi_2\rangle\ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2\rangle}{[\![\bar{a}\ \langle \Pi \rhd \phi_1 \vee \phi_2\rangle\ \bar{x}]\!] \vdash \langle P_1 \vee P_2 \mid \mathsf{if}(P_1)\ \{\bar{T}_1\}\ \mathsf{else}\ \{\bar{T}_2\}\rangle}$$

$$\textsc{List-rec}\ \frac{\begin{array}{c}(\Pi_1 \wedge P) \implies \Pi_2 \qquad \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \implies \Pi_2[a_0 \mapsto t] \qquad [\![\bar{a}\ \langle \Pi_2 \rhd \phi[a_0 \mapsto \mathsf{Nil}]\rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}_1\rangle \\ [\![\bar{r}, h, t, \bar{a}\ \langle \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \wedge \phi[a_0 \mapsto t, \bar{x} \mapsto \bar{r}] \rhd \phi[a_0 \mapsto \mathsf{Cons}(h,t)]\rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}_2\rangle\end{array}}{[\![a_0, \bar{a}\ \langle \Pi_1 \rhd \phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \mathsf{rec}(a_0, \bar{a})\rangle}$$

Figure 7. Selected synthesis inference rules for one-point rule, static computation, and splitting, as well as an illustrative instance (for lists) of a general rule for structural recursion on algebraic data types (see text for the definition of rec)

LIST-REC generates programs that traverse lists. Our system automatically generates such a rule for each recursive datatype occurring in the context of synthesis. The rule for a binary tree type, for example, will spawn a subproblem for the Leaf case and another problem with two recursive calls for the Node case. Our rule for integers uses a recursion scheme that approaches zero for both negative and positive integers.

Terminal rules. For a synthesis problem to be completely solved, some rules must be terminal, that is, not generate any subproblem. Terminal rules are by definition the only ones which can close a branch in the derivation tree.

One such example is GROUND in Figure 7. The rule states that if a synthesis problem does not refer to any input variable, then it can be treated as a satisfiability problem: any model for the predicate ϕ can then be used as a ground solution term for $\bar{x}$.

All terminal rules that we consider have the form:

$$\textsc{Terminal}\ \frac{\forall \bar{a} :\ \Pi \implies \phi[\bar{x} \mapsto \bar{T}]}{[\![\bar{a}\ \langle \Pi \rhd \phi\rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}\rangle}$$

which essentially encodes relation refinement. Domain preservation is automatically enforced by considering only solutions where the precondition is true, as the rule shows. While we could in principle devise algorithms that try to solve for a program term and a minimal precondition at the same time, we found the approach of deriving the preconditions using non terminal rules only to be sufficient for our test cases. The remaining challenge is therefore to efficiently compute $\bar{T}$ given Π and ϕ. Sections 5 and 6 detail two algorithms to this effect.

5. Symbolic Term Exploration Rule

In this section, we describe the first of our two most important terminal rules, which is responsible for closing many of the branches in derivation trees. We call it Symbolic Term Exploration (STE).

The core idea behind STE is to symbolically represent many possible terms (programs), and to iteratively prune them out using counterexamples and test case generation until either 1) a valid term is proved to solve the synthesis problem or 2) all programs in the search space have been shown to be inadequate. Since we already have rules that take care of introducing branching constructs or recursive functions, we focus STE on the search for terms consisting only of constructors and calls to existing functions.

Recursive generators. We start from a universal non-deterministic program that captures all the (deterministic) programs which we wish to consider as potential solutions. We then try to resolve the non-deterministic choices in such a way that the program realizes the desired property. Resolving the choices consists in fixing some values in the program, which we achieve by running a counterexample driven search.

We describe our non-deterministic programs as a set of recursive non-deterministic *generators*. Intuitively, a generator for a given type is a program that produces arbitrary values of that type. For instance, a generator for positive integers could be given by:

def genInt() : Int = **if**($\star$) 0 **else** (1 + genInt())

where $\star$ represents a non-deterministic boolean value. Similarly a non-deterministic generator for the List type could take the form:

def genList() : List = **if**($\star$) Nil **else** Cons(genInt(), genList())

It is not required that generators can produce *every* value for a given type; we could hypothesize for instance that our synthesis solutions will only need some very specific constants, such as 0, 1 or -1. What is more likely is that our synthesis solutions will need to use input variables and existing functions. Our generators therefore typically include variables of the proper type that are accessible in the synthesis environment. Taking these remarks into account, if a and b are integer variables in the scope, and f is a function from Int to Int, a typical generator for integers would be:

```
def genInt() : Int = if(⋆) 0 else if(⋆) 1 else if(⋆) −1
        else if(⋆) a else if(⋆) b else f(genInt())
```

From generators to formulas. Generators can in principle be any function with unresolved non-deterministic choices. For the sake of the presentation, we assume that they are "flat", that is, they consist of a top-level non-deterministic choice between n alternatives. (Note that the examples given above all have this form.)

Encoding a generator into an SMT term is done as follows: introduce for each invocation of a generator an uninterpreted constant c of the proper type, and for each non-deterministic choice as many boolean variables $\bar{b}$ as there are alternatives. Encode that exactly one of the $\bar{b}$ variables must be true, and constrain the value of c using the $\bar{b}$ variables.

Recursive invocations of generators can be handled similarly, by inserting another c variable to represent their value and constraining it appropriately. Naturally, these recursive instantiations must stop at some point: we then speak of an *instantiation depth*. As an example, the encoding of the genList generator above with an instantiation depth of 1 and assuming for simplicity that genInt only generates 0 or a is:

$$
\begin{aligned}
& (b_1 \vee b_2) \wedge (\neg b_1 \vee \neg b_2) \\
\wedge\ & b_1 \Rightarrow c_1 = \mathsf{Nil} \wedge b_2 \Rightarrow c_1 = \mathsf{Cons}(c_2,c_3) \\
\wedge\ & (b_3 \vee b_4) \wedge (\neg b_3 \vee \neg b_4) \\
\wedge\ & b_3 \Rightarrow c_2 = 0 \wedge b_4 \Rightarrow c_2 = a \\
\wedge\ & (b_5 \vee b_6) \wedge (\neg b_5 \vee \neg b_6) \\
\wedge\ & b_5 \Rightarrow c_3 = \mathsf{Nil} \wedge b_6 \Rightarrow c_3 = \mathsf{Cons}(c_4,c_5) \\
\wedge\ & \neg b_6
\end{aligned}
$$

The clauses encode the following possible values for c_1: Nil, Cons(0, Nil) and Cons(a, Nil). Note the constraint $\neg b_6$ which enforces the instantiation depth of 1, by preventing the values beyond that depth (namely c_4 and c_5) to participate in the expression.

For a given instantiation depth, a valuation for the $\bar{b}$ variables encodes a determinization of the generators, and as a consequence a program. We solve for such a program by running a refinement loop.

Refinement loop: discovering programs. Consider a synthesis problem $[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$, where we speculate that a generator for the types of $\bar{x}$ can produce a program that realizes ϕ. We start by encoding the non-deterministic execution of the generator for a fixed instantiation depth (typically, we start with 0). Using this encoding, the problem has the form:

$$\phi \wedge B(\bar{a}, \bar{b}, \bar{c}) \wedge C(\bar{c}, \bar{x}) \tag{2}$$

where ϕ is the synthesis problem, B is the set of clauses obtained by encoding the execution of the generator and C is a set of equalities tying $\bar{x}$ to a subset of the $\bar{c}$ variables. Note that by construction, the values for $\bar{c}$ (and therefore for $\bar{x}$) are uniquely determined when $\bar{a}$ and $\bar{b}$ are fixed.

We start by finding values for $\bar{a}$ and $\bar{b}$ such that (2) holds. If no such values exist, then our generators at the given instantiation depth are not expressive enough to encode a solution to the problem. Otherwise, we extract for the model the values $\bar{b}_0$. They describe a candidate program, which we put to the test.

Refinement loop: falsifying programs. We search for a solution to the problem:

$$\neg\phi \wedge B(\bar{a}, \bar{b}_0, \bar{c}) \wedge C(\bar{c}, \bar{x}) \tag{3}$$

Note that $\bar{b}_0$ are constants, and that $\bar{c}$ and $\bar{x}$ are therefore uniquely determined by $\bar{a}$ this intuitively comes from the fact that $\bar{b}_0$ encodes a deterministic program, that $\bar{c}$ encodes intermediate values in the execution of that program, and that $\bar{x}$ encodes the result. With this in mind, it becomes clear that we are really solving for $\bar{a}$.

If no such $\bar{a}$ exist, then we have found a program that realizes ϕ and we are done. If on the other hand we can find $\bar{a}_0$, then this constitutes an input that witnesses that our program does not meet the specification. In this case, we can discard the program by asserting $\neg \bigwedge \bar{b}$, and going back to (2).

Eventually, because the set of possible assignments to $\bar{b}$ is finite (for a given instantiation depth) this terminates. If we have not found a program, we can increase the instantiation depth and try again. When the maximal depth is reached, we give up.

Filtering with concrete execution. While termination is in principle guaranteed by the successive elimination of programs in the refinement loop, the formula encoding the non-deterministic term typically grows exponentially as the instantiation depth increases. As the number of candidates grows, the difficulty for the solver to satisfy (2) or (3) also increases. As an alternative to symbolic elimination, we can often use concrete execution on a set of input tests to rule out many programs.

To execute these such concrete tests, we first generate a set of input candidates that satisfy the path condition. For this, we use Leon's ground term generator (see Section 3). We then test candidate programs on this set of inputs. Programs that fail on at least one input can be discarded.

To make testing efficient in our implementation, we compile the expression $\phi[\bar{x} \mapsto \mathsf{genX}(\bar{b})]$ on the fly to JVM bytecode, where genX denotes the non-deterministic generator for the types of $\bar{x}$, and $\bar{b}$ are boolean literals that control the decisions in genX. In other words, the expression uses both the inputs $\bar{a}$ and an assignment to $\bar{b}$ to compute whether the program represented by the choices $\bar{b}$ succeeds in producing a valid output for $\bar{a}$.

This encoding of all candidate programs into a single executable function allows us to rapidly test and potentially discard hundreds or even thousands of candidates within a fraction of a second. Whenever the number of discarded candidates is deemed substantial, we regenerate a new formula

for (2) with much fewer boolean variables and continue from there. The speedup achieved through filtering with concrete executions is particularly important when STE is applied to a problem it cannot solve. In such cases, filtering often rules out all candidate programs and symbolic reasoning is never applied.

6. Condition Abduction Rule

We next describe our second major terminal rule, which synthesizes recursive functions with conditional statements. We refer to the underlying technique (and its implementation as a rule in our system) as Condition Abduction (CA).

We assume that we are given a function header and a post-condition, and that we aim to synthesize a recursive function body. The body expression must be 1) a well-typed term with respect to the context of the program and 2) valid according to the imposed formal specification. An approach to solving such a synthesis problem could be based on searching the space of all expressions that can be built from all declarations visible at the program point, filtering out those that do not type-check or return the desired type, and find one that satisfies the given formal specification. Unless the process of generating candidate solutions is carefully guided, the search becomes unfeasible (as we have observed in previous versions of our tool).

6.1 Condition Abduction

Our idea for guiding the search and incremental construction of correct expressions is influenced by abductive reasoning [18, 19]. Abductive reasoning, sometimes also called "inference to the best explanation", is a method of reasoning in which one chooses a hypothesis that would explain the observed evidence in the best way. The motivation for applying abductive reasoning to program synthesis comes from examining implementations of practical purely functional, recursive algorithms. The key observation is that recursive functional algorithms often consist of a top-level case analysis expression (if-then-else or pattern matching) with recursive calls in the branches. This pattern is encoded with a branching control flow expression that partitions the space of input values such that each branch represents a correct implementation for a certain partition. Such partitions are defined by conditions that guard branches in the control flow.

This allows us to synthesize branches separately, by searching for implementations that evaluate correctly only for certain inputs, thus reducing the search space. Rather than speculatively applying CASE-SPLIT rule to obtain subproblems and finding solutions for each branch by case analysis (as described in Section 4), this idea applies a similar strategy in the reverse order—first obtaining a candidate program and then searching for a condition that makes it correct. Abductive reasoning can guess the condition that defines a valid partition, i.e. "abduce" the explanation for a partial implementation with respect to a given candidate program. Our rule CA progressively applies this technique and enables effective search and construction of a control flow expression that represents a correct implementation for more and more input cases, eventually constructing an expression that is a solution to the synthesis problem.

Algorithm 1 Synthesis with condition abduction

Require: path condition Π, predicate ϕ, a collection of expressions s ▷ synthesis problem $[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$

$p' = true$ ▷ maintain the current partition
$sol = (\lambda x.x)$ ▷ maintain a partial solution
$\mathcal{M} = \text{SAMPLEMODELS}(\bar{a})$ ▷ set of example models
repeat
  get a set of expressions E from s ▷ candidates
  for each e in E **do** ▷ count passed examples p_e for e
    $p_e = |\{m \in \mathcal{M} \mid e(m) \text{ is correct}\}|$ ▷ evaluate
  $\bar{r} = \arg\max_{e \in E}\ p_e$ ▷ the highest ranked expression
  if solution $\langle \Pi \wedge p' \mid \bar{r} \rangle$ is valid **then**
    return $\langle \Pi \mid (sol\ \bar{r}) \rangle$ ▷ a solution is found
  else
    extract new counterexample model m
    $\mathcal{M} = \mathcal{M} \cup m$ ▷ accumulate examples
    $c = \text{BRANCHSYN}(\bar{r}, p, q, s)$ ▷ call Algorithm 2
    if $c \neq$ FALSE **then** ▷ a branch is synthesized
      $sol = (\lambda x.\ (sol\ (\text{if } c \text{ then } \bar{r} \text{ else } x)))$
      $p' = p' \wedge \neg c$ ▷ update current partition
until s is empty

6.2 Synthesizing Conditional Recursive Functions

Algorithm 1 presents our rule that employs a new technique for guiding the search with ranking and filtering based on counterexamples, as well as constructing expressions from partially correct implementations.

The algorithm applies the idea of abducing conditions to progressively synthesize and verify branches of a correct implementation for an expanding partition of inputs. The input to the algorithm is a path condition Π, a predicate ϕ (defined by synthesis problem $[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$), and a collection of expressions s (see Section 6.3 below for details on how we obtain a suitable s in our implementation).

Condition p' defines which inputs are left to consider at any given point in the algorithm; these are the inputs that belong to the current partition. The initial value of p' is $true$, so the algorithm starts with a partition that covers the whole initial input space constrained only by the path condition Π. Let $p_1, \ldots, p_k$, where $k > 0$, be conditions abduced up to a certain point in the algorithm. Then p' represents the conjunction of negations of abduced conditions, i.e. $p' = \neg p_1 \wedge \ldots \wedge \neg p_k$. Together with the path condition, it defines the current partition which includes all input values for which there is no condition abduced (nor correct implementation found). Thus, the guard condition for the current partition is defined by $\Pi \wedge p'$. The algorithm maintains the

partial solution sol, encoded as a function. sol encodes an expression which is correct for all input values that satisfy any of the abduced conditions and this expression can be returned as a partial solution at any point. Additionally, the algorithm accumulates example models in the set $\mathcal{M}$. Ground term generator, described in Section 3, is used to construct the initial set of models in $\mathcal{M}$. To construct a model, for each variable in $\bar{a}$, the algorithm assigns a value sampled from the ground term generator. Note that more detailed discussion on how examples are used to guide the search is deferred to Section 6.3.

The algorithm repeats enumerating all possible expressions from the given collection until it finds a solution. In each iteration, a batch of expressions E is enumerated and evaluated on all models from $\mathcal{M}$. The results of such evaluation are used to rank expressions from E. The algorithm considers the expression of the highest rank $\bar{r}$ as a candidate solution and checks it for validity. If $\bar{r}$ represents a correct implementation for the current partition, i.e. if $\langle \Pi \wedge p' \mid \bar{r} \rangle$ is a valid solution, then the expression needed to complete a valid control flow expression is found. The algorithm returns it as solution for which $[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!] \vdash \langle \Pi \mid (sol\ \bar{r}) \rangle$ holds. Otherwise, the algorithm extracts the counterexample model m, adds it to the set $\mathcal{M}$, and continues by trying to synthesize a branch with expression $\bar{r}$ (it does so by calling Algorithm 2 which will be explained later). If BRANCHSYN returns a valid branch condition, the algorithm updates the partial solution to include the additional branch (thus extending extending the space of inputs covered by the partial solution), and refines the current partition condition. The new partition condition reduces the synthesis to a subproblem, ensuring that the solution in the next iteration covers cases where c does not hold. The algorithm eventually, given the appropriate terms from s, finds an expression that forms a complete correct implementation for the synthesis problem.

Algorithm 2 Synthesize a branch

Require: expression $\bar{r}$, condition p', predicate q, and a collection of expressions s ▷ passed from Algorithm 1

1: **function** BRANCHSYN($\bar{r}, p', q, s$)
2: $\mathcal{M}' = \emptyset$ ▷ set of accumulated counterexamples
3: get a set of expressions E' from s ▷ candidates
4: **for each** c in E' **do**
5: **if for each** model m in $\mathcal{M}'$, $c(m) = false$ **then**
6: **if** solution $\langle \Pi \wedge c \mid \bar{r} \rangle$ is valid **then**
7: **return** c ▷ a condition is abduced
8: **else**
9: extract the new counterexample model m
10: $\mathcal{M}' = \mathcal{M}' \cup m$ ▷ accumulate counterexamples
11: return FALSE ▷ no condition is found

Algorithm 2 tries to synthesize a new branch by abducing a valid branch condition c. It does so by enumerating a set of expressions E' from s and checking whether it can find a valid condition expression, that would guard a partition for which the candidate expression $\bar{r}$ is correct.The algorithm accumulates counterexample models in $\mathcal{M}'$ and considers a candidate expression c only if it prevents all accumulated counterexamples. The algorithm checks this by evaluating c on m, i.e. $c(m)$, for each accumulated counterexample m. If a candidate expression c is not filtered out, the algorithm checks if c represents a valid branch condition, i.e. whether $\langle \Pi \wedge c \mid \bar{r} \rangle$ is a valid solution. If yes, the algorithm returns c which, together with $\bar{r}$, comprises a valid branch in the solution to $[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$. Otherwise, it adds a new counterexample model to $\mathcal{M}'$ and continues with the search. If no valid condition is in E', the algorithm returns FALSE.

6.3 Organization of the Search

For getting the collection of expressions s, CA uses a term enumerator that generates all well-typed terms constructible from the datatypes and functions in the scope of the synthesis problem (in our implementation, we reused the typed expression enumerator from the InSynth tool [12, 26]).

The completeness property of such generators ensures systematic enumeration of all candidate solutions that are defined by the set of given type constraints. For verification, CA uses the Leon verifier component, that allows checking validity of expressions that are supported by the underlying theories and obtaining counterexample models.

The context of CA as a rule in the Leon synthesis framework imposes limits on the portion of search space explored by each instantiation. This allows incremental and systematic progress in search space exploration and, due to the mixture with other synthesis rules, offers benefits in both expressiveness and performance of synthesis. CA offers flexibility in adjusting necessary parameters and thus a fine-grain control over the search. For our experiments, the size of candidate sets of expressions enumerated in each iteration n is 50 (and doubled in each iteration) for Algorithm 1 and 20 for Algorithm 2.

Using (counter-)examples. A technique that brings significant performance improvements when dealing with large search spaces is guiding the search and avoiding considering candidate expressions according to the information from examples generated during synthesis. After checking an unsatisfiable formula, CA queries Leon's verifier for the witness model. It accumulates such models and uses them to narrow down the search space.

Algorithm 2 uses accumulated counterexamples to filter out unnecessary candidate expressions when synthesizing a branch. It makes sense to consider a candidate expression for a branch condition, c, for a check whether c makes $\bar{r}$ a correct implementation, only if c prevents all accumulated counterexamples that already witnessed unsatisfiability of the correctness formula for $\bar{r}$, i.e. if $\forall m \in \mathcal{M}'.\ c \rightarrow \neg m$. Otherwise, if $\exists m \in \mathcal{M}'.\neg(c \rightarrow \neg m)$, then m is a valid counterexample to the verification of $\langle \Pi \wedge c \mid \bar{r} \rangle$. This effectively guides the search by the results of previous veri-

fication failures while filtering out candidates before more expensive verification check are made.

Algorithm 1 uses accumulated models to quickly test and rank expressions by evaluating models according to the specification. The current set of candidate expressions E is evaluated on the set of accumulated examples $\mathcal{M}$ and the results of such evaluation are used to rank the candidates. We call an evaluation of a candidate e on a model m correct, if m satisfies path condition Π and the result of the evaluation satisfies given predicate ϕ. The algorithm counts the number of correct evaluations, ranks the candidates accordingly, and considers only the candidate of the highest rank. The rationale is that the more correct evaluations, the more likely the candidate represents a correct implementation for some partition of inputs. Note that evaluation results may be used only for ranking but not for filtering, because each candidate may represent a correct implementation for a certain partition of inputs, thus incorrect evaluations are expected even for valid candidates. Because the evaluation amounts to executing the specification, this technique is efficient in guiding the search towards correct implementations while avoiding unnecessary verification checks.

7. Exploring the Space of Subproblems

In the previous three sections, we described a general formal framework in which we can describe what constitutes a synthesis problem and a solution. We have shown rules that decompose problem into pieces and presented two more complex terminal rules that solve larger sub-problems: symbolic term exploration and condition-abduction. In this section, we describe how all these rules are instantiated in practice, and how they work together to derive a complete solution to a synthesis problem.

Inference rules are non-deterministic by nature. They justify the correctness of a solution, but do not by themselves describe how one finds that solution. Our search for a solution alternates between considering 1) which rules apply to given problems, and 2) which subproblems are generated by rule instantiations.

The task of finding rules that apply to a problem intuitively correspond to finding an inference rule whose conclusion matches the structure of a problem. For example, to apply GROUND, the problem needs to mention only output variables. Similarly, to apply LIST-REC to a problem, it needs to contain at least one input variable of type List.

Computing the subproblems resulting from the application of a rule is in general straightforward, as they correspond to problems appearing in its premise. The GROUND rule, for example, generates no subproblem, while LIST-REC generates two.

AND/OR *search.* To solve one problem, it suffices to find a complete derivation from *one* rule application to that problem. However, to fully apply a rule, we need to solve *all* generated subproblems. This corresponds to searching for a closed branch in an AND/OR tree [31].

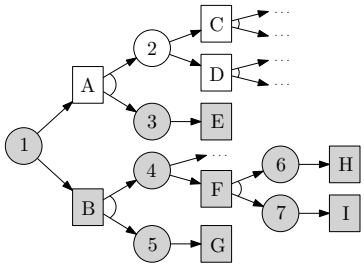

Figure 8. An AND/OR search tree used to illustrate our search mechanism. Circles are OR nodes and represent problems, while boxes are AND nodes and represent our rule applications. Nodes in grey are closed (solved).

We now describe the expansion of such a tree using an example. Consider the problem of removing a given element e from a list a. In our logical notation –using α as an abbreviation for content– the problem is:

$$[\![a\,, e\ \langle \mathsf{true} \rhd \alpha(x) = \alpha(a) \setminus \{e\}\rangle\ x]\!]$$

We denote this problem by 1 in the tree of Figure 8. While we haven't given an exhaustive list of all rules used in our system, it is fair to assume that more than one can apply to this problem. For example, we could case-split on the type of a, or apply LIST-REC to a. We represent these two options by A and B respectively in the tree.

Following the option B and applying LIST-REC with the path condition $\Pi_2 \equiv \mathsf{true}$ trivially satisfies the first two premises of the rules, and generates two new problems (4 and 5). Problem 5 is:

$$[\![e\ \langle \mathsf{true} \rhd \alpha(x) = \alpha(\mathsf{Nil}) \setminus \{e\}\rangle\ x]\!]$$

where the predicate simplifies to $\alpha(x) = \emptyset$. This makes it possible to apply the GROUND rule (node G). This generates no subproblem, and closes the subbranch with the solution solution $\langle \mathsf{true} \mid \mathsf{Nil}\rangle$. Problem 4 has the form:

$$[\![r\,, h\,, t\,, e\ \langle \alpha(r) = \alpha(t) \setminus \{e\} \rhd \\ \alpha(x) = \alpha(\mathsf{Cons}(h,t)) \setminus \{e\}\rangle\ x]\!]$$

Among the many possible rule applications, we can choose to case-split on the equality $h = e$ (node F). This generates two subproblems. Problem 6

$$[\![r\,, h\,, t\,, e\ \langle \alpha(r) = \alpha(t) \setminus \{e\} \wedge e = h \rhd \\ \alpha(x) = \alpha(\mathsf{Cons}(h,t)) \setminus \{e\}\rangle\ x]\!]$$

and a similar problem 7, where $e \neq h$ appears in the path condition instead of $e = h$. Both subproblems can be solved by using a technique we will describe in Section 5 to derive a term satisfying the synthesis predicate, effectively closing

the complete branch from the root. The solutions for problem 6 and 7 are $\langle \mathsf{true} \mid r \rangle$ and $\langle \mathsf{true} \mid \mathsf{Cons}(h,r) \rangle$ respectively. A complete reconstruction of the solution given by the branch in gray yields the program:

```
def rec(a : List) : List = a match {
  case Nil ⇒ Nil
  case Cons(h,t) ⇒
    val r = rec(t)
    if(e == h) r
    else Cons(h,r)
}
```

In the interest of space, we have only described the derivations that lead to the solution. In practice, not all correct steps are taken in the right order. The interleaving of expansions of AND and OR nodes is driven by the *estimated cost* of problems and solutions.

Cost models. To drive the search, we assign to each problem and to each rule application an estimated cost, which is supposed to under-approximate the actual final cost of a closed branch. For OR nodes (problems), the cost is simply the minimum of all remaining viable children, while for AND nodes (rule applications) we take the sum of the cost of each children plus a constant. That constant intuitively corresponds to the extra complexity inherent to a particular rule.

A perfect measure for cost would be the running time of the corresponding program. However, this is particularly hard to estimate, and valid under-approximations would most likely be useless. We chose to measure program size instead, as we expect it to be a reasonable proxy for complexity. We measure the size of the program as the number of branches, weighted by their proximity to the root. We found this to have a positive influence on the quality of solutions, as it discourages near-top-level branching.

Using this metric, the cost inherent to a rule application roughly corresponds to the extra branches it introduces in the program. We use a standard algorithm for searching for the best solution [31], and the search thus always focuses on the current most promising solution. In our example in Figure 8, we could imagine that after the case split at F, the B branch temporarily became less attractive. The search then focuses for a while on the A branch, until expansion on that side (for example, by case-splitting on the type of the list) reaches a point where the minimal possible solution is worse than the B branch. We note that the complete search takes about two seconds.

Anytime synthesis. Because we maintain the search tree and know the current minimal solution at all times, we can stop the synthesis at any time and obtain a partial program that is likely to be good. This option is available in our implementation, both from the console mode and the web interface. In such cases, Leon will return a program containing new invocations of **choose** corresponding to the open subproblems.

Operation	Size	Calls	Proof	sec.
`List.Insert`	3	0	√	0.6
`List.Delete`	19	1	√	1.8
`List.Union`	12	1	√	2.1
`List.Diff`	12	2	√	7.6
`List.Split`	27	1	√	9.3
`SortedList.Insert`	34	1		9.9
`SortedList.InsertAlways`	36	1	√	7.2
`SortedList.Delete`	23	1	√	4.1
`SortedList.Union`	19	2	√	4.5
`SortedList.Diff`	13	2	√	4.0
`SortedList.InsertionSort`	10	2	√	4.2
`SortedList.MergeSort`	17	4	√	14.3
`StrictSortedList.Insert`	34	1	√	14.1
`StrictSortedList.Delete`	21	1		15.1
`StrictSortedList.Union`	19	2	√	3
`UnaryNumerals.Add`	11	1	√	1.3
`UnaryNumerals.Distinct`	12	0	√	1.1
`UnaryNumerals.Mult`	12	1	√	2.7
`BatchedQueue.Checkf`	14	4	√	7.4
`BatchedQueue.Snoc`	7	2	√	3.7
`AddressBook.Make`	50	14		8.8
`AddressBook.MakeHelpers`	21	5		4.9
`AddressBook.Merge`	11	3		8.9

Figure 9. Automatically synthesized functions using our system. We consider a problem as synthesized if the solution generated is correct after manual inspection. For each generated function, the table lists the size of its syntax tree and the number of function calls it contains. √ indicates that the system also found a proof that the generated program matches the specification: in many cases proof and synthesis are done simultaneously, but in rare cases merely a large number of automatically generated inputs passed the specification. The final column shows the total time used for both synthesis and verification.

Parallelization The formulation of our search over programs using AND/OR graphs with independent sub-problems allows us to parallelize its exploration. We implemented the parallel search in a system composed of several worker actors managed by a central coordinator. The share-nothing approach of the actor model is particularly adequate in our case given the independence of sub-problems. For all non-trivial benchmarks, the speed-up induced by concurrent workers exceeds the setup and communication overheads.

8. Implementation and Results

We have implemented these techniques in Leon, a system for verification and synthesis of functional program, thus extending it from the state described in Section 3. Our implementation and the online interface are available from `http://lara.epfl.ch/w/leon/`.

As the front end to Leon, we use the first few phases of the reference Scala compiler (for Scala 2.10). The Scala compiler performs parsing, type checking, and tasks such as the expansion of implicit conversions, from which Leon directly benefits. We then filter out programs that use Scala features not supported in Leon, and convert the syntax trees to Leon's internal representation. Leon programs can also execute as valid Scala programs.

We have developed several interfaces for Leon. Leon can be invoked as a batch command-line tool that accepts verification and synthesis tasks and outputs the results of the requested tasks. If desired, there is also a console mode that allows applying synthesis rules in a step-by-step fashion and is useful for debugging purposes.

To facilitate interactive experiments and the use of the system in teaching, we have also developed an interface that executes in the web browser, using the Play framework of Scala as well as JavaScript editors. Our browser-based interface supports continuous compilation of Scala code, allows verifying individual functions with a single keystroke or click, as well as synthesizing any given **choose** expression. In cases when the synthesis process is interrupted, the synthesizer can generate a partial solution that contains a program with further occurrences of the **choose** statement.

In order to evaluate our system, we developed benchmarks with reusable abstraction functions. The example section already pointed out to some of the results we obtained. We next summarize further results and discuss some of the remaining benchmarks. The synthesis problem descriptions are available in the appendix, along with their solution as computed by Leon.

Our set of benchmarks displayed in Figure 9 covers the synthesis of various operations over custom data structures with invariants, specified through the lens of abstraction functions. These benchmarks use specifications that are both easy to understand and shorter than resulting programs (except in trivial cases). Most importantly, the specification functions are easily reused across synthesis problems. We believe these are key factors in the evaluation of any synthesis procedure.

Figure 9 shows the list of functions we successfully synthesized. In addition to the address book and sorted list examples shown in Section 2, our benchmarks include operations on unary numerals, defined as is standard as "zero or successor", and on an amortized queue implemented with two lists from a standard book on functional data structure implementation [34]. Each synthesized program has been manually validated to be a solution that a programmer might expect. Synthesis is performed in order, meaning that an operation will be able to reuse all previously synthesized ones, thus mimicking the usual development process. For instance, multiplication on unary numerals is synthesized as repeated invocations of additions.

Our system typically also proves automatically that the resulting program matches the specification for all inputs. In some cases, the lack of inductive invariants prevents fully-automated proof of the synthesize code (we stop verification after a timeout of 3 seconds). In most cases the synthesis succeeds sufficiently fast for a reasonable interactive experience. Among the largest benchmarks is synthesis of creation of the address book (AddressBook.make), which automatically derives a function to classify a list into two sublists that are inserted into the appropriate fields of an address book. This example is at the frontier of what is possible today; our system can synthesize it but would need additional inductive strengthening of the specification to verify it. In such cases, it is possible that the returned solution is incorrect for the inputs that our verifier and counterexample finder did not consider within a given time limit. On the one hand, this shows a limitation when requiring fully verified solutions. On the other hand, it shows the importance of verification, and points to another possible use of our system: it can automatically synthesize buggy benchmarks for software testing and verification tools, benchmarks that are close to being correct yet for which it is difficult to automatically find test inputs that witness the buggy behavior.

9. Related Work

Our approach is similar in the spirit to deductive synthesis [29, 30, 39], which incorporates transformation of specifications, inductive reasoning, recursion schemes and termination checking, but we extend it with modern SMT techniques, new search algorithms, and a new cost-based synthesis framework. The type-driven counterexample-guided synthesis with condition abduction (Section 6) directly uses the complete completion technique [12] including the succinct representation of types. However, our use adds several crucial dimensions to the basic functionality of generating well-typed terms: we add mechanisms to ensure that the terms make sense *semantically* and not only in terms of types, though the use of a verifier and automatically generated counterexamples. Moreover, this is only our starting point and the main novelty is the addition of the inference of conditional recursive programs. We currently do not use the full power of [12] because we make no use of 1) ranking of solutions based on the occurrence of symbols in the corpus nor 2) the ability of [12] to generate first-class functions (the functions we try synthesize here are first-order).

The origins of our deductive framework is in complete functional synthesis, which was used previously for integer linear arithmetic [24]. In this paper we do not use synthesis rules for linear integer arithmetic. Instead, we here use synthesis procedure rules for algebraic data types [15, 46], which were not reported in an implemented system before. This gives us building blocks for synthesis of recursion-free code. To synthesize recursive code we developed new algorithms, which build on and further advance

the counterexample-guided approach to synthesis [40], but applying it to the context of an SMT instead of SAT solver, and using new approaches to control the search space.

Deductive synthesis frameworks. Early work on synthesis [29, 30] focused on synthesis using expressive and undecidable logics, such as first-order logic and logic containing the induction principle.

Programming by refinement has been popularized as a manual activity [3, 53]. Interactive tools have been developed to support such techniques in HOL [6]. A recent example of deductive synthesis and refinement is the Specware system from Kesterel [39]. We were not able to use the system first-hand due to its availability policy, but it appears to favor expressive power and control, whereas we favor automation.

A combination of automated and interactive development is analogous to the use of automation in interactive theorem provers, such as Isabelle [33]. However, whereas in verification it is typically the case that the program is available, the emphasis here is on constructing the program itself, starting from specifications.

Work on synthesis from specifications [44] resolves some of these difficulties by decoupling the problem of inferring program control structure and the problem of synthesizing the computation along the control edges. The work leverages verification techniques that use both approximation and lattice theoretic search along with decision procedures, but appears to require more detailed information about the structure of the expected solution than our approach.

Synthesis with input/output examples. One of the first works that addressed synthesis with examples and put inductive synthesis on a firm theoretical foundation is the one by Summers [45]. Subsequent work presents extensions of the classical approach to induction of functional Lisp-programs [13, 20]. These extensions include synthesizing a set of equations (instead of just one), multiple recursive calls and systematic introduction of parameters. Our current system lifts several restrictions of previous approaches by supporting reasoning about arbitrary datatypes, supporting multiple parameters in concrete and symbolic I/O examples, and allowing nested recursive calls and user-defined declarations.

Inductive (logic) programming that explores automatic synthesis of (usually recursive) programs from incomplete specifications, most often being input/output examples [9, 32], influenced our work. Recent work in the area of programming by demonstration has shown that synthesis from examples can be effective in a variety of domains, such as spreadsheets [38]. Advances in the field of SAT and SMT solvers inspired counter-example guided iterative synthesis [11, 40], which can derive input and output examples from specifications. Our tool uses and advances these techniques through two new counterexample-guided synthesis approaches.

ESCHER, recently presented inductive synthesis algorithm that is completely driven by input/output examples and focuses on synthesis of recursive procedures, shares some similarities with some of our rules [1]. By following the goal graph, which is similar in function as the AND/OR search tree, ESCHER tries to detect if two programs can be joined by a conditional. The split goal rule in ESCHER can speculatively split goals and is thus similar to our splitting rules. One of the differences is that ESCHER can split goals based on arbitrary choices of satisfied input/output example pairs, while our rules impose strictly predefined conditions that correspond to common branching found in programs. We found it difficult to compare the two frameworks because ESCHER needs to query the oracle (the user) for input/output examples each time a recursive call is encountered (in the SATURATE rule). We do not consider it practical to allow the synthesizer to perform such extensive querying, because the number of recursive calls during synthesis tends to be very large. Thus, ESCHER appears more suitable for scenarios such as reverse-engineering a black-box implementation from its observable behavior than for synthesis based on user's specification.

Our approach complements the use of SMT solvers with additional techniques for automatic generation of input/output examples. Our current approach is domain-agnostic although in principle related to techniques such as Korat [5] and UDITA [10].

Synthesis based on finitization techniques. Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [40–42]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. The tool we presented shows one way to move the ideas of sketching towards infinite domains. In this generalization we leverage reasoning about equations as much as SAT techniques.

Reactive synthesis. Synthesis of reactive systems generates programs that run forever and interact with the environment. Known complete algorithms for reactive synthesis work with finite-state systems [37] or timed systems [2]. Finite-state synthesis techniques have applications to control the behavior of hardware and embedded systems or concurrent programs [50]. These techniques usually take specifications in a fragment of temporal logic [36] and have resulted in tools that can synthesize useful hardware components [17]. Recently such synthesis techniques have been extended to repair that preserves good behaviors [51], which is related to our notion of partial programs that have remaining **choose** statements. These techniques have been applied to the component-based synthesis problem for finite-state components [28]; we focus on infinite domains, but for simpler, input/output computation model.

TRANSIT combines synthesis and model checking to bring a new model for programming distributed protocols

[49], which is a challenging case of a reactive system. Specification of a protocol is given with a finite-state-machine description augmented with snippets that can use concrete and symbolic values to capture intended behavior. Similarly to our STE rule, the main computational problem solving in TRANSIT is based on the counter-example guided inductive synthesis (CEGIS) approach while the execution of concrete specification is used to prune the parts of the synthesis search space. Although similarity exists between the concept of progressive synthesis of guarded transitions in TRANSIT and inferring branches in our case splitting and condition abduction rules, the crucial difference is that our framework infers the entire implementation, including the control flow (with recursive calls), without the need of approximate control flow specification. On the other hand, the effectiveness of TRANSIT is increased by focusing on a particular application domain, which is the direction we will leave for the future.

Automated inference of program fixes and contracts. These areas share the common goal of inferring code and rely on specialized software synthesis techniques [35, 52]. Inferred software fixes and contracts are usually snippets of code that are synthesized according to the information gathered about the analyzed program. The core of these techniques lies in the characterization of runtime behavior that is used to guide the generation of fixes and contracts. Such characterization is done by analyzing program state across the execution of tests; state can be defined using user-defined query operations [52], and additional expressions extracted from the code [35]. Generation of program fixes and contracts is done using heuristically guided injection of (sequences of) routine calls into predefined code templates.

Our synthesis approach works with purely functional programs and does not depend on characterization of program behavior. It is more general in the sense that it focuses on synthesizing whole correct functions from scratch and does not depend on already existing code. Moreover, rather than using execution of tests to define starting points for synthesis and SMT solvers just to guide the search, our approach utilizes SMT solvers to guarantee correctness of generated programs and uses execution of tests to speedup the search. Coupling of flexible program generators and the Leon verifier provides more expressive power of the synthesis than filling of predefined code schemas. Our approach does not find errors and infer contracts, but can be utilized for those tasks if the appropriate reformulation of the synthesis problem is made - desired code needs to be in the place of a priori located errors or inside contracts.

10. Conclusions and Analysis

Software synthesis is a difficult problem but we believe it can provide substantial help in software development. We have presented a new framework for synthesis that combines transformational and counterexample-guided approaches. Our implemented system can synthesize and prove correct functional programs that manipulate unbounded data structures such as algebraic data types. We have used the system to synthesize algorithms that manipulate list and tree structures. Our approach leverages the state of the art SMT solving technology and an effective mechanism for solving certain classes of recursive functions. Thanks to this technology, we were already able to synthesize programs over unbounded domains that are guaranteed to be correct for all inputs. Our automated system can be combined with manual transformations or run-time constraint solving [25] to cover the cases where static synthesis does not fully solve the problem. It can further be improved by adding additional rules for manually verified refactoring and automatic synthesis steps [24], by informing the search using statistical information from a corpus of code [12] and using domain-specific higher-order combinators [43], as well as by further improvements in decision procedures to enhance the class of verifiable programs.

Acknowledgments

We thank Tihomir Gvero for the InSynth implementation [12] whose modification [26] we use in our condition-abduction rule to enumerate terms of a given type. We thank Régis Blanc for his contribution to the Leon verification infrastructure, including the implementation of rules for Presburger arithmetic synthesis. We thank Ruzica Piskac and Mikaël Mayer for useful discussions on synthesis.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.

[2] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems*, pages 1–20, 1994.

[3] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer-Verlag, 1998.

[4] R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *Scala Workshop*, 2013.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[6] M. Butler, J. Grundy, T. Langbacka, R. Ruksenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In *Formal Methods Pacific*, 1997.

[7] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[8] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, 2009.

[9] P. Flener and D. Partridge. Inductive programming. *Autom. Softw. Eng.*, 8(2):131–137, 2001.

[10] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *ICSE*, pages 225–234, 2010.

[11] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[12] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

[13] M. Hofmann. IgorII - an analytical inductive functional programming system (tool demo). In *PEPM*, pages 29–32, 2010.

[14] D. Jackson. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol.*, 4(4):365–389, 1995.

[15] S. Jacobs, V. Kuncak, and P. Suter. Reductions for synthesis procedures. In *VMCAI*, pages 88–107, 2013.

[16] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.

[17] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, pages 117–124, 2006.

[18] J. R. Josephson. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge University Press, 1994.

[19] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[20] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *JMLR*, 7:429–454, 2006.

[21] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.

[22] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL*, pages 151–164, 2012.

[23] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[24] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *CACM*, 55(2):103–111, 2012.

[25] V. Kuncak, E. Kneuss, and P. Suter. Executing specifications using synthesis and constraint solving (invited talk). In *Runtime Verification (RV)*, 2013.

[26] I. Kuraj. Interactive code generation. Master's thesis, EPFL, February 2013.

[27] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM*, pages 806–809, 2009.

[28] Y. Lustig and M. Y. Vardi. Synthesis from component libraries. In *FOSSACS*, pages 395–409, 2009.

[29] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[30] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1): 90–121, 1980.

[31] A. Martelli and U. Montanari. Additive AND/OR graphs. In *IJCAI*, pages 1–11, 1973.

[32] S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[33] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[34] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[35] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Code-based automated program fixing. *ArXiv e-prints*, 2011. `arXiv:1102.1059`.

[36] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.

[37] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[38] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012.

[39] D. R. Smith. Generating programs plus proofs by refinement. In *VSTTE*, pages 182–188, 2005.

[40] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[41] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[42] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[43] A. Spielmann, A. Nötzli, C. Koch, V. Kuncak, and Y. Klonatos. Automatic synthesis of out-of-core algorithms. In *SIGMOD*, 2013.

[44] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[45] P. D. Summers. A methodology for LISP program construction from examples. *JACM*, 24(1):161–175, 1977.

[46] P. Suter. *Programming with Specifications*. PhD thesis, EPFL, December 2012.

[47] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210, 2010.

[48] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.

[49] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.

[50] M. T. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS*, pages 139–154, 2009.

[51] C. von Essen and B. Jobstmann. Program repair without regret. In *CAV*, pages 896–911, 2013.

[52] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, pages 191–200, 2011.

[53] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

A. Benchmark Problems and Solutions

We report the key parts of the input source code and the synthesis result for several benchmarks shown in Section 8.

A.1 SortedList

Definitions.

```
sealed abstract class List
case class Cons(head : Int, tail : List) extends List
case object Nil extends List
def size(l : List) : Int = ...
def content(l : List) : Set[Int] = ...

def isSorted(list : List) : Boolean = list match {
  case Nil ⇒ true
  case Cons(_, Nil) ⇒ true
  case Cons(x1, Cons(x2, _)) if(x1 > x2) ⇒ false
  case Cons(_, xs) ⇒ isSorted(xs) }

def insert(in1 : List, v : Int) = {
  require(isSorted(in1))
  choose { (out : List) ⇒ isSorted(out) &&
    (content(out) == content(in1) ++ Set(v)) }}

def delete(in1 : List, v : Int) = {
  require(isSorted(in1))
  choose { (out : List) ⇒ isSorted(out) &&
    (content(out) == content(in1) -- Set(v)) }}

def merge(in1 : List, in2 : List) = {
  require(isSorted(in1) && isSorted(in2))
  choose { (out : List) ⇒ isSorted(out) &&
    (content(out) == content(in1) ++ content(in2)) }}

def diff(in1 : List, in2 : List) = {
  require(isSorted(in1) && isSorted(in2))
  choose { (out : List) ⇒ isSorted(out) &&
    (content(out) == content(in1) -- content(in2)) }}

def split(list : List) : (List,List) = {
  choose { (res : (List,List)) ⇒
    val s1 = size(res._1)
    val s2 = size(res._2)
    abs(s1 − s2) ≤ 1 && s1 + s2 == size(list) &&
    content(res._1) ++ content(res._2) == content(list) }}

def sort(list : List) : List = choose {
  (res : List) ⇒ isSorted(res) &&
    content(res) == content(list) }
```

Synthesized solutions.

```
def insert(in1 : List, v : Int) = {
  require(isSorted(in1))
  in1 match {
    case Nil ⇒ Cons(v, Nil)
    case Cons(h, t) ⇒ if (v < h) {
      Cons(v, in1)
    } else {
      Cons(h, insert(t, v)) }}}

def delete(in1 : List, v : Int) : List = {
  require(isSorted(in1))
  in1 match {
    case Nil ⇒ Nil
    case Cons(h, t) ⇒ if (v == h) {
      delete(t, v)
    } else {
      Cons(h, delete(t, v)) }}}

def merge(in1 : List, in2 : List) : List = {
  require(isSorted(in1) && isSorted(in2))
  in1 match {
    case Nil ⇒ Nil
    case Cons(h, t) ⇒ merge(t, insert(in2, h)) }}

def diff(in1 : List, in2 : List) : List = {
  require(isSorted(in1) && isSorted(in2))
  in2 match {
    case Nil ⇒ in1
    case Cons(h, t) ⇒ diff(delete(in1, h), t) }}

def split(in : List) : (List,List) = in match {
  case Nil ⇒ (Nil, Nil)
  case Cons(h, Nil) ⇒ (in, Nil)
  case Cons(h1, Cons(h2, t2)) ⇒
    val (s1, s2) = split(t2)
    (Cons(h1, s1), Cons(h2, s2)) }

def sort(lst : List) : List = lst match {
  case Nil ⇒ lst
  case Cons(_, Nil) ⇒ lst
  case _ ⇒ {
    val p = split(list)
    merge(sort(p.fst), sort(p.snd)) }}
```

A.2 UnaryNumerals

Definitions.

```
sealed abstract class Num
case object Z extends Num
case class S(pred : Num) extends Num
def value(n :Num) : Int = (n match {
  case Z ⇒ 0
  case S(p) ⇒ 1 + value(p)
}) ensuring (_ ≥ 0)

def add(x : Num, y : Num) : Num = {
  choose { (r : Num) ⇒
    value(r) == value(x) + value(y) }}

def mult(x : Num, y : Num) : Num = {
  choose { (r : Num) ⇒
    value(r) == value(x) ∗ value(y) }}

def distinct(x : Num, y : Num) : Num = {
  choose { (r : Num) ⇒
    value(r) != value(x) &&
    value(r) != value(y) }}
```

Synthesized solutions.

```
def add(x : Num, y : Num) : Num = x match {
  case Z ⇒ y
  case S(p) ⇒ add(p, S(y)) }

def mult(x : Num, y : Num) : Num = y match {
  case Z ⇒ Z
  case S(p) ⇒ add(x, mult(x, y−1)) }

def distinct(x : Num, y : Num) : Num = x match {
  case Z ⇒ S(y)
  case S(p) ⇒ y match {
    case Z ⇒ S(x)
    case S(p) ⇒ Z }}
```

A.3 BatchedQueue

Definitions.

```
sealed abstract class List
case class Cons(head : Int, tail : List) extends List
case object Nil extends List
def content(l : List) : Set[Int] = ...

case class Queue(f : List, r : List)
def content(p : Queue) : Set[Int] =
  content(p.f) ++ content(p.r)
def isEmpty(p : Queue) : Boolean = p.f == Nil

def revAppend(aList : List, bList : List) : List =
  aList match {
    case Nil ⇒ bList
    case Cons(x, xs) ⇒
      revAppend(xs, Cons(x, bList))
} ensuring (res ⇒
  content(_) == content(aList) ++ content(bList))

def invariantList(q : Queue, f : List, r : List) = {
  revAppend(q.f, q.r) == revAppend(f, r) &&
  q.f != Nil || q.r == Nil }

def reverse(list : List) = revAppend(list, Nil)
  ensuring (content(_) == content(list))

def checkf(f : List, r : List) : Queue =
  choose { (res : Queue) ⇒
    invariantList(res, f, r) }

def tail(p : Queue) : Queue = p.f match {
  case Nil ⇒ p
  case Cons(_, xs) ⇒ checkf(xs, p.r) }

def snoc(p : Queue, x : Int) : Queue = {
  choose { (res : Queue) ⇒
    content(res) == content(p) ++ Set(x) &&
    (isEmpty(p) || content(tail(res)) ++
      Set(x) == content(tail(res))) } }
```

Synthesized solutions.

```
def checkf(f : List, r : List) : Queue = r match {
  case r @ Nil ⇒ Queue(f, r)
  case _ ⇒ f match {
    case f @ Cons(_, _) = Queue(f, r)
    case _ ⇒ Queue(reverse(r), Nil) }}

def snoc(p : Queue, x : Int) : Queue =
  Queue(p.f, Cons(x, p.r))
```

A.4 AddressBooks

Definitions. (Section 2.2 shows solutions)

```
case class Info(address : Int, zipcode : Int,
                phoneNumber : Int)
case class Address(info : Info, priv : Boolean)

sealed abstract class List
case class Cons(a : Address, tail :List) extends List
case object Nil extends List
def content(l : List) : Set[Address] = ...
def size(l : List) : Int = ...

def allPrivate(l : List) : Boolean = l match {
  case Nil ⇒ true
  case Cons(a, l1) ⇒
    if (a.priv) allPrivate(l1)
    else false }
def allBusiness(l : List) : Boolean = l match {
  case Nil ⇒ true
  case Cons(a, l1) ⇒
    if (a.priv) false
    else allBusiness(l1) }

case class AddressBook(business : List, pers : List)
def size(ab : AddressBook) : Int =
  size(ab.business) + size(ab.pers)
def isEmpty(ab : AddressBook) = size(ab) == 0
def content(ab : AddressBook) : Set[Address] =
  content(ab.pers) ++ content(ab.business)

def invariant(ab : AddressBook) =
  allPrivate(ab.pers) && allBusiness(ab.business)

def makeAddressBook(l : List) : AddressBook =
  choose { (res : AddressBook) ⇒
    size(res) == size(l) &&
    invariant(res) }

def merge(l1 : List, l2 : List) : List = l1 match {
  case Nil ⇒ l2
  case Cons(a, tail) ⇒ Cons(a, merge(tail, l2)) }

def mergeAddressBooks(
  ab1 : AddressBook, ab2 : AddressBook) = {
  require(invariant(ab1) && invariant(ab2))
  choose { (res : AddressBook) ⇒
    (size(res) == size(ab1) + size(ab2)) &&
    invariant(res) }}
```