Technical Report, April 2013

On Verification by Translation to Recursive Functions

An Overview of the Leon Verification System

Régis Blanc Etienne Kneuss Viktor Kuncak Philippe Suter

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

firstname.lastname@epfl.ch

Abstract

We present the Leon verification system for a subset of the Scala programming language. Along with several functional features of Scala, Leon supports imperative constructs such as mutations and loops, using a translation into recursive functional form. Both properties and programs in Leon are expressed in terms of user-defined functions. We discuss several techniques that led to an efficient semi-decision procedure for first-order constraints with recursive functions, which is the core solving engine of Leon. We describe a generational unrolling strategy for recursive templates that yields smaller satisfiable formulas and ensures completeness for counter-examples. We evaluate the benefits of these techniques on a set of examples.

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

1. Introduction

Scala supports the development of reliable software in a number of ways: concise and readable code, an advanced type system, and testing frameworks such as Scalacheck. This paper adds a new dimension to this reliability toolkit: an automated program verifier for a Scala subset. Our verifier, named Leon, leverages existing run-time checking constructs for Scala, the **require** and **ensuring** clauses [31], allowing them to be proved statically, for all executions. The specification constructs use executable Scala expressions, possibly containing function calls. Developers therefore need not learn a new specification language, but simply obtain additional leverage from executable assertions, and additional motivation to write them. Thanks to Leon, assertions can be statically checked, providing full coverage over all executions.

Leon thus brings strong guarantees of static types to the expressive power of tests and run-time checks. It is difficult to argue with usefulness of having the same specification and implementation language, especially when this language has clear semantics. Although not universally used, such approaches have been adopted in the past, most notably in the ACL2 system and its predecessors [20], which have been used to verify an impressive set of real-world systems [21].

At the core of Leon is a verifier for a purely functional subset of Scala. The verifier makes use of contracts when they are available, but does not require fully inductive invariants and can be used even with few or no annotations. Like bounded model checking algorithms, the algorithms inside Leon are guaranteed to find an error if it exists, even if the program has no auxiliary annotations. We have found this aspect of Leon to be immensely useful in practice for debugging both specifications and the code. In addition to the ability to find all errors, the algorithms inside Leon also terminate for correct programs when they belong to well-specified fragments of decidable theories with recursive functions [34, 35].

The completeness makes Leon suitable for extended type checking. It can, for example, perform semantic exhaustiveness checks for advanced pattern matching constructs, and predictably verify invariants on algebraic data types. A notable feature is that Leon is guaranteed to accept a correct program in such fragments, will not accept an incorrect program, and is guaranteed to find a counterexample if the program is not correct. A combination of these features is something that neither typical type systems nor verification techniques achieve; this has been traditionally reserved for model checking algorithms on finite-state programs. The techniques in Leon now bring these benefits to functional programs that manipulate unbounded data types.

Leon can thus be simultaneously viewed as a theorem prover and as a program verifier. It tightly integrates with the Z3 theorem prover [10], mapping functional Scala data types directly to mathematical data types of Z3. This direct mapping means that we can use higher-level reasoning than employed in many imperative program verifiers that must deal with pointers and complex library implementations. As a prover, Leon extends the theory of Z3 with recursive functions. To handle such functions Leon uses an algorithm for iterative unfolding with under- and over-approximation of recursive calls. The implementation contains optimizations that leverage incremental solving of Z3 to make the entire process efficient. Leon thus benefits from the ideas of sym-

bolic execution, but without limitations on the number of memory cells in the initial state, and without explicit enumeration of program paths. Completeness for counterexamples is possible in Leon due to the executable nature of its language. We use executability in Leon not only to provide guarantees on the algorithm, but also to improve the performance of the solver: in a number of scenarios we can replace constraint solving in the SMT solver with direct execution of the original program. For that purpose, we have built a simple and fast bytecode compiler inside Leon.

Although the core language of Leon engine is a set of pure recursive functions, Leon also supports several extensions to accept more general forms of programs as input. In particular, it supports nested function definitions, mutable local variables, local mutable arrays, and while loops. Such fragment is related to those used in modeling languages such as VDM [18, 19], and abstract state machines [9]. Leon translates such extended constructs into the flat functional code, while preserving input-output behavior. In contrast to many verification-condition generation approaches that target SMT provers, Leon's semantic translation does not require invariants, it preserves validity, and also preserves counterexamples. We expect to continue following such methodology in the future, as we add more constructs into the subset that Leon supports. Note that a basic support for higher-order functions was available in a past version of Leon [23]; it is currently disabled, but a new version is under development.

We show that usefulness of Leon on a number of examples that include not only lightweight checking but also on more complex examples of full-functional verification. Such tasks are usually associated with less predictable and less automated methods, such as proof assistants. We have found Leon to be extremely productive for development of such programs and specifications. Although Leon does ultimately face limitations for tasks that require creative use of induction and lemmas, we have found it to go a long way in debugging code and specification that match. To further improve usefulness of Leon, we have built a web-based interface, running at:

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

The web interface supports continuous compilation and verification of programs as well as sharing verified programs through stable links. Leon also supports automated and interactive program synthesis [22]. This functionality heavily relies on verification, but is beyond the scope of the present paper.

In its current state, we believe that Leon is very useful for modeling and verification tasks. We have used it to verify and find errors in a number of complex functional data structures and algorithms, some of which we also illustrate in this paper. The design of Leon purposely avoids heavy annotations and complex worst-case encoding of imperative program semantics. Leon is therefore as much a verification project as it is a language design and implementation project: it aims to keep the verification tractable while gradually increasing the semantics and the complexity of programs and problems that it can handle.

```
def insert(e: Int, l: List): List = {
  require(isSorted(l))
  l match {
    case Nil ⇒ Cons(e,Nil)
    case Cons(x,xs) ⇒
      if (x ≤ e) Cons(x,insert(e, xs)) else Cons(e, l)
  }
} ensuring(res ⇒
    contents(res) == contents(l) ++ Set(e) && isSorted(res))

def sort(l: List): List = (l match {
  case Nil ⇒ Nil
  case Cons(x,xs) ⇒ insert(x, sort(xs))
})ensuring(res ⇒ contents(res) == contents(l) && isSorted(res))

def contents(l: List): Set[Int] = l match {
  case Nil ⇒ Set.empty[Int]
  case Cons(x,xs) ⇒ contents(xs) ++ Set(x)
}
```

Figure 1. Insertion sort.

In the Spring 2013 semester we have also used Leon in the master's course on Synthesis, Analysis, and Verification. Web framework allowed us to use zero-setup to get students to start verifying examples. During the course we have formulated further assignments and individual projects for students to add functionality to the verifier. We are also in the process of making the repository of Leon public and look forward to community contributions and experiments.

2. Examples

We introduce the flavor of verification and error finding in Leon through sorting and data structure examples. We focus on describing three structures of three examples; Section 7 presents our results on a larger selection. The online interface at `http://lara.epfl.ch/leon/` provides the chance to test the system and its responsiveness.

2.1 Insertion Sort

Figure 1 shows insertion sort implemented in the subset of the language that Leon supports. `List` is defined as a recursive algebraic data types storing list of integers. Note that Leon also supports the usual pattern matching on algebraic data types, and our example functions rely on it. The example illustrates the syntax for preconditions (**require**) and postconditions (**ensuring**). When compiled with `scalac` these constructs are interpreted as dynamic contracts that throw corresponding exceptions, whereas Leon tries to prove statically that their conditions hold. The `contents` and `isSorted` functions are user defined functions defined

```
def add(x: Int, t: Tree): Tree = {
  require(redNodesHaveBlackChildren(t) && blackBalanced(t))

  def ins(x: Int, t: Tree): Tree = {
    require(redNodesHaveBlackChildren(t) && blackBalanced(t))
    t match {
      case Empty ⇒ Node(Red,Empty,x,Empty)
      case Node(c,a,y,b) ⇒
        if (x < y) balance(c, ins(x, a), y, b)
        else if (x == y) Node(c,a,y,b)
        else balance(c,a,y,ins(x, b))
  }}ensuring (res ⇒
      content(res) == content(t) ++ Set(x) &&
      size(t) ≤ size(res) && size(res) ≤ size(t) + 1 &&
      redDescHaveBlackChildren(res) && blackBalanced(res))

  def makeBlack(n: Tree): Tree = {
    require(redDescHaveBlackChildren(n) && blackBalanced(n))
    n match {
      case Node(Red,l,v,r) ⇒ Node(Black,l,v,r)
      case _ ⇒ n
  }}ensuring(res ⇒
      redNodesHaveBlackChildren(res) && blackBalanced(res))
  // body of add:
  makeBlack(ins(x, t))
}ensuring (res ⇒ content(res) == content(t) ++ Set(x) &&
    redNodesHaveBlackChildren(res) && blackBalanced(res))
```

Figure 2. Adding an element into a red-black tree.

```
def balance(c: Color, a: Tree, x: Int, b: Tree): Tree = {
Node(c,a,x,b) match {
  case Node(Black,Node(Red,Node(Red,a,xV,b),yV,c),zV,d) ⇒
    Node(Red,Node(Black,a,xV,b),yV,Node(Black,c,zV,d))
  case Node(Black,Node(Red,a,xV,Node(Red,b,yV,c)),zV,d) ⇒
    Node(Red,Node(Black,a,xV,b),yV,Node(Black,c,zV,d))
  case Node(Black,a,xV,Node(Red,Node(Red,b,yV,c),zV,d)) ⇒
    Node(Red,Node(Black,a,xV,b),yV,Node(Black,c,zV,d))
  case Node(Black,a,xV,Node(Red,b,yV,Node(Red,c,zV,d))) ⇒
    Node(Red,Node(Black,a,xV,b),yV,Node(Black,c,zV,d))
  case Node(c,a,xV,b) ⇒ Node(c,a,xV,b)
}} ensuring (res ⇒ content(res) == content(Node(c,a,x,b)))
```

Figure 3. Balancing a red-black tree.

also recursively for the purpose of expressing specifications. Leon supports sets, which is useful for writing abstractions of container structures.

We have also verified or found errors in more complex algorithms, such as merge sort and a mutable array-based implementation of a quick sort.

2.2 Red-Black Trees

Leon is also able to handle complex data structures. Figure 2 shows the insertion of an element into a red-black tree, establishing that the algebraic data type of trees satisfies a number of complex invariants [32]. Leon proves, in particular, that the insertion maintains the coloring and height invariant of red-black trees, and that it correctly updates the set of elements after the operation. These invariants are expressed using recursive functions that take an algebraic data type value and return a boolean value indicating whether the property holds. This example also introduces an additional feature of Leon which is the possibility to define local functions. Local functions help build a clean interface to a function by keeping local operations hidden. Figure 3 shows a balancing operation of a red-black tree. A functional description of this operation is very compact and also very easy for Leon to handle: the correct version in the figure verifies instantly, whereas a bug that breaks its correctness is instantly identified with a counterexample. Note that, although the function is non-recursive, its specification uses a recursive function content.

```
def maxSum(a: Array[Int]): (Int, Int) = {
  require(a.length > 0)
  var sum = 0
  var max = 0
  var i = 0
  (while(i < a.length) {
    if(max < a(i))
      max = a(i)
    sum = sum + a(i)
    i = i + 1
  }) invariant (sum ≤ i ∗ max && 0 ≤ i && i ≤ a.length)
  (sum, max)
} ensuring(res ⇒ res._1 ≤ a.length ∗ res._2)
```

Figure 4. Sum and max of an array.

2.3 Sum and Max

To illustrate imperative constructs in Leon, Figure 4 shows an example inspired by the Problem 1 of the VSTTE competition in 2010. Note that the example uses arrays, loops, and mutable local variables. Leon proves its correctness instantly by first translating the while loop into a nested tail-recursive pure function, hoisting the generated nested function outside, and verifying the resulting functional program.

3. Leon Language

We now describe the Leon input language, a subset of the Scala programming language. This subset is composed of two parts: a purely functional part referred to as PureScala and a selected set of extensions. The formal grammar of this subset can be found in Figure 5. It covers most first-order features of Scala, including case classes and pattern matching. It also supports special data types such as sets, maps, and arrays. However, only a selected number of methods are supported for these types.

This subset is expressive enough to concisely define custom data-structures and their corresponding operations. The specifications for these operations can be provided through **require** and **ensuring** constructs. Contracts are also written in this subset and can leverage the same expressiveness. Pro-

grams and contracts are thus defined using the same executable language.

While having a predominant functional flavor, Scala also supports imperative constructs such as mutable fields and variables. It is however common to see mutation being limited to the scope of a function, keeping the overall function free from observable side-effects. Indeed, it is often easier to write algorithms with local mutation and loops rather than using their equivalent purely functional forms. For this reason, we extended PureScala with a set of imperative constructs, notably permitting local mutations and while loops. We describe later in Section 5 how we handle these extensions specifically.

4. Core Algorithm

In this section, we give an overview of an algorithm to solve constraints over PureScala expressions. (For the theoretical foundations and the first experiments on functional programs, please see [33, 35].)

This procedure is the core of Leon's symbolic reasoning capabilities: more expressive constructs are reduced to this subset (see Section 5).

The idea of the algorithm is to determine the truth value of a PureScala boolean expression (*formula*) through a succession of under- and over-approximations. PureScala is a Turing-complete language, so we cannot expect this to always succeed. Our algorithm, however, has the desirable theoretical property that it always finds counter-examples to invalid formulas. It is thus a *semi-decision procedure* for PureScala formulas.

All the data types of PureScala programs are readily supported by state-of-the-art SMT solvers, which can efficiently decide formulas over combinations of theories such as boolean algebra, integer arithmetic, term algebras (ADTs), sets or maps [6, 10, 12]. The remaining challenge is in handling user-defined recursive functions. SMT solvers typically support *uninterpreted function symbols*, and we leverage those in our procedure. Uninterpreted function symbols are a useful over-approximation of interpreted function symbols; because the SMT solver is allowed to assume *any* model for an uninterpreted function, when it reports that a constraint is unsatisfiable it implies that, in particular, there is also no solution when the correct interpretation is assumed. On the other hand, when the SMT solver produces a model for a constraint assuming uninterpreted functions, we cannot reliably conclude that a model exists for the correct interpretation. The challenge that Leon's algorithm essentially addresses is to find reliable models in this later case.

To be able to perform both over-approximation and under-approximation, we transform functional programs into logical formulas that represent partial deterministic paths in the program. For each function in a Leon program,

Purely functional subset (PureScala):

$$
\begin{aligned}
program &::= \textbf{object}\ id\ \{\ definition^*\ \}\\
definition &::= \mathsf{abstract}\ \textbf{class}\ id\\
&\quad \mid \textbf{case class}\ id\ (\ decls\ )\ \textbf{extends}\ id\\
&\quad \mid fundef\\
fundef &::= \textbf{def}\ id\ (\ decls\ ) : type = \{\\
&\qquad \langle\ \textbf{require}(\ expr\ )\ \rangle?\\
&\qquad expr\\
&\quad \}\ \langle\ \textbf{ensuring}\ (\ id\ \Rightarrow\ expr\ )\ \rangle?\\
decls &::= \epsilon\ \mid\ id{:}\ type\ \langle\ ,\ id{:}\ type\ \rangle^*\\
expr &::= \mathsf{0}\ \mid\ \mathsf{1}\ \mid\ \ldots\ \mid\ \textbf{true}\ \mid\ \textbf{false}\ \mid\ id\\
&\quad \mid \textbf{if}\ (\ expr\ )\ expr\ \textbf{else}\ expr\\
&\quad \mid \textbf{val}\ id = expr;\ expr\\
&\quad \mid (\ \langle\ expr\ \langle\ ,\ expr\ \rangle^*\ \rangle?\ )\\
&\quad \mid id\ (\ \langle\ expr\ \langle\ ,\ expr\ \rangle^*\ \rangle?\ )\\
&\quad \mid expr\ \textbf{match}\ \{\ \langle\ \textbf{case}\ pattern\ \Rightarrow\ expr\ \rangle^*\ \}\\
&\quad \mid expr\ .\ id\\
&\quad \mid expr\ .\ id\ (\ \langle\ expr\ \langle\ ,\ expr\ \rangle^*\ \rangle?\ )\\
pattern &::= binder\ \mid\ binder : type\\
&\quad \mid binder\ \mathtt{@}\ id(\ \langle\ pattern\ \langle\ ,\ pattern\ \rangle^*\ \rangle?\ )\\
&\quad \mid binder\ \mathtt{@}\ (\ pattern\ \langle\ ,\ pattern\ \rangle^*\ )\\
&\quad \mid id(\ \langle\ pattern\ \langle\ ,\ pattern\ \rangle^*\ \rangle?\ )\\
&\quad \mid (\ pattern\ \langle\ ,\ pattern\ \rangle^*\ )\\
binder &::= id\ \mid\ \_\\
id &::= \mathsf{IDENT}\\
type &::= id\ \mid\ \mathsf{Int}\ \mid\ \mathsf{Boolean}\ \mid\ \mathsf{Set}[\ type\ ]\\
&\quad \mid \mathsf{Map}[\ type, type\ ]\ \mid\ \mathsf{Array}[\ type\ ]
\end{aligned}
$$

Imperative constructs and nested functions:

$$
\begin{aligned}
expr &::= \textbf{while}\ (\ expr\ )\ expr\ \langle\ \textbf{invariant}\ (\ expr\ )\ \rangle?\\
&\quad \mid \textbf{if}\ (\ expr\ )\ expr\\
&\quad \mid \textbf{var}\ id = expr\\
&\quad \mid id = expr\\
&\quad \mid id\ (\ expr\ ) = expr\\
&\quad \mid fundef\\
&\quad \mid \{\ expr\ \langle\ ;\ expr\ \rangle^*\ \}\\
&\quad \mid (\ )\\
type &::= \mathsf{Unit}
\end{aligned}
$$

Figure 5. Abstract syntax of the Leon input language.

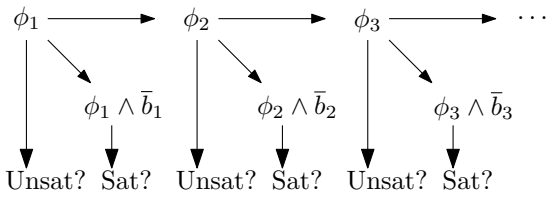


Figure 6. Successive over- and under-approximations.

we generate an equivalent representation as a set of clauses. For instance, for the function

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

we produce the clauses:

$$\begin{array}{l}(\mathsf{size}(\mathsf{lst}) = \mathsf{e}_1) \wedge (\mathsf{b}_1 \iff \mathsf{lst} = \mathsf{Nil}) \\ \wedge\ (\mathsf{b}_1 \implies \mathsf{e}_1 = 0) \wedge (\neg \mathsf{b}_1 \implies \mathsf{e}_1 = \mathsf{size}(\mathsf{lst.tail}))\end{array} \quad (1)$$

Intuitively, these clauses represent the relation between the input variable lst and the result. The important difference between the two representation is the introduction of variables that represent the status of branches in the code (in this example, the variable b_1). Explicitly naming branch variables allows us to control the parts of function definitions that the SMT solver can explore.

As an example, consider a constraint $\phi \equiv \mathsf{size}(\mathsf{lst}) = 1$. We can create a formula equisatisfiable —assuming the correct interpretation of size— with ϕ by conjoining it with the clauses (1). We call this new formula ϕ_1. Now, assuming an uninterpreted function symbol for size, if ϕ_1 is unsatisfiable, then so is ϕ for *any* interpretation of size. If however ϕ_1 is satisfiable, it may be because the uninterpreted term size(lst.tail) was assigned an impossible value.[1] We control for this by checking the satisfiability of $\phi_1 \wedge \mathsf{b}_1$. This additional boolean literal forces the solver to ignore the branch containing the uninterpreted term. If this new formula is satisfiable, then so is ϕ_1 and we are done. If it is not, it may be because of the restricted branch. In this case, we introduce the definition of size(lst.tail) by instantiating the clauses (1) one more time, properly substituting lst.tail for lst, and using fresh variables for b_1 and e_1.

We can repeat these steps, thus producing a sequence of alternating approximations. This process is depicted in Figure 6. An important property is that, while it may not necessarily derive all proofs of unsatisfiability, this technique will always find counter-examples when they exist. Intuitively, this happens because a counter-example corresponds to an execution of the property resulting in false, and our technique enumerates all possible executions in increasing lengths.

[1] Note that there is a chance that the model is in fact valid. In Leon we check this by running an evaluator, and return the result if confirmed.

5. Handling Imperative Programs by Translation

We now present the transformations we apply to reduce the general input language of Leon to its functional core, PureScala. We present a recursive procedure to map imperative statements to a series of definitions (**val** and **def**) that form a new scope introducing fresh names for the program variables, and keeping a mapping from program variables to their current name inside the scope. The procedure is inspired from the generation of verification conditions from a program [11, 15, 28]. However such methods suffer from an exponential growth in the size of the program fragment. In some sense, our transformation to functional programs, followed by a later generation of verification conditions avoid the exponential growth similarly to the work of Flanagan et al. [13].

Intuitively, we can represent any imperative snippet as a series of definitions followed by a group of parallel assignments. These assignments rename the program variables to their new names, that is, the right hand side will be the new identifiers of the program variable (that have been introduced by the definitions) and the left hand side will be the program variables themselves. Those parallel assignments are an explicit representation of the mapping from program variables to their fresh names. As an example, consider the following imperative program:

```
x = 2
y = 3
x = y + 1
```

It can be equivalently written as follows:

```
val x1 = 2
val y1 = 3
val x2 = y1 + 1
x = x2
y = y1
```

This is the intuition behind this mapping from program variables to their fresh identifiers representation. The advantage is that we can build a recursive procedure and easily combine the results when we have sequences of statements.

5.1 Example

Let us first look at an example. The following program computes the floor of the square root of an integer n:

```
def sqrt(n : Int) : Int = {
  var toSub = 1
  var left = n
  while(left ≥ 0) {
    if(toSub % 2 == 1)
      left -= toSub
    toSub += 1
  }
  (toSub / 2) − 1
}
```

Our transformation starts from the innermost elements, in particular it will transform the conditional expression to the following:

```
val left2 = if(toSub % 2 == 1) {
  val left1 = left − toSub
  left1
} else {
  left
}
left = left2
```

Then it combines this expression with the rest of the body of the loop, yielding:

```
val left2 =
  if(toSub % 2 == 1) {
    val left1 = left − toSub
    left1
  } else {
    left
  }
val toSub1 = tuSub + 1
left = left2
toSub = toSub1
```

The final assignments can be seen as a mapping from program identifiers to fresh identifiers. The **while** loop is then translated to a recursive function using a similar technique:

```
def rec(left3: Int, toSub2: Int) = if(left3 ≥ 0) {
  val left2 =
    if(toSub3 % 2 == 1) {
      val left1 = left3 − toSub2
      left1
    } else {
      left3
    }
  val toSub1 = tuSub2 + 1
  rec(left2, toSub1)
} else {
  (left3, toSub2)
}
val (left4, toSub3) = rec(left, toSub)
left = left4
toSub = toSub3
```

In this transformation, we made use of the mapping information in the body for the recursive call. Note that a loop invariant would be translated to a pre and post-condition of the recursive function. We also substituted left and toSub in the body of the recursive function. In the final step, we combine all top level statements and substitute the new variables in the returned expression:

```
def sqrt(n : Int) : Int = {
  val toSub4 = 1
  val left5 = n
  def rec(left3: Int, toSub2: Int) = if(left3 ≥ 0) {
    val left2 =
      if(toSub3 % 2 == 1) {
        val left1 = left3 − toSub2
        left1
      } else {
        left3
      }
    val toSub1 = tuSub2 + 1
    rec(left2, toSub1)
  } else {
    (left3, toSub2)
  }
  val (left4, toSub3) = rec(left5, toSub4)
  (toSub3 / 2) − 1
}
```

5.2 Transformation Rules

Figure 7 shows the formal rules to rewrite imperative code into equivalent functional code. The rules define a function $e \rightsquigarrow \langle T \mid \sigma \rangle$, which constructs from an expression e a term constructor T and a variable substitution function σ.

We give the main rules for each fundamental transformation. This is a mathematical formalization of the intuition of the previous section, we defined a scope of definitions as well as maintained a mapping from program variables to fresh names. Note that each time we introduce subscripted versions of variables, we are assuming they adopt fresh names.

We write term constructors as terms with exactly one instance of a special value $\Box$ (a "hole"). If e is an expression and T a term constructor, we write $T[\mathsf{e}]$ the expression obtained by applying the constructor T to e ("plugging the hole"). We also use this notation to apply a term constructor to another constructor, in which case the result is a new term constructor. Similarly, we apply variables substitutions to variables, variable tuples, expressions and term constructors alike, producing as an output the kind passed as input.

As an illustration, if $T \equiv \Box + \mathsf{y}$, $\mathsf{e} \equiv \mathsf{x} + 1$, and $\sigma \equiv \{\mathsf{x} \mapsto \mathsf{z}\}$, then we have for instance:

$$T[e] \equiv \mathsf{x} + 1 + \mathsf{y} \qquad T[T] \equiv \Box + \mathsf{y} + \mathsf{y}$$

$$\sigma(e) \equiv \mathsf{z} + 1 \qquad \sigma(T) \equiv \Box + \mathsf{y}$$

We denote the point-wise update of a substitution function by $\sigma_1 \uplus \sigma_2$. This should be interpreted as "σ_2 or else σ_1". That is, in case the same variable is mapped by both σ_1 and σ_2, the mapping in σ_2 overrides the one in σ.

For ease of presentation, we assume that blocks of statements are terminated with a pure expression r from the core language, which corresponds to the value computed in the block. So, given the initial body of the block b and the following derivation:

$$\mathsf{b} \rightsquigarrow \langle s \mid \sigma \rangle$$

we can define the function expression equivalent to $\mathsf{b}; \mathsf{r}$ by:

$$T[\sigma(\mathsf{r})]$$

This simplification allows us to ignore the fact that each of those expressions with side effect actually returns a value,

$$\frac{}{\mathsf{x} = \mathsf{e} \rightsquigarrow \langle \textbf{val}\ \mathsf{x}_1 = \mathsf{e}; \square \mid \{\mathsf{x} \mapsto \mathsf{x}_1\}\rangle} \qquad \frac{}{\textbf{var}\ \mathsf{x} = \mathsf{e} \rightsquigarrow \langle \textbf{val}\ \mathsf{x}_1 = \mathsf{e}; \square \mid \{\mathsf{x} \mapsto \mathsf{x}_1\}\rangle} \qquad \frac{\mathsf{e1} \rightsquigarrow \langle T_1 \mid \sigma_1\rangle \qquad \mathsf{e2} \rightsquigarrow \langle T_2 \mid \sigma_2\rangle}{\mathsf{e1}; \mathsf{e2} \rightsquigarrow \langle T_1[\sigma_1(T_2)] \mid \sigma_1 \uplus \sigma_2\rangle}$$

$$\frac{\mathsf{t} \rightsquigarrow \langle T_1 \mid \sigma_1\rangle \qquad \mathsf{e} \rightsquigarrow \langle T_2 \mid \sigma_2\rangle \qquad \mathrm{dom}(\sigma_1 \uplus \sigma_2) = \overline{\mathsf{x}}}{\textbf{if}(\mathsf{c})\ \mathsf{t}\ \textbf{else}\ \mathsf{e} \rightsquigarrow \langle \textbf{val}\ \overline{\mathsf{x}}_1 = \textbf{if}(\mathsf{c})\ T_1[\sigma_1(\overline{\mathsf{x}})]\ \textbf{else}\ T_2[\sigma_2(\overline{\mathsf{x}})]; \square \mid \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_1\}\rangle} \qquad \frac{}{() \rightsquigarrow \langle \square \mid \emptyset\rangle}$$

$$\frac{\mathsf{e} \rightsquigarrow \langle T_1 \mid \sigma_1\rangle \qquad \sigma_1 = \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_1\} \qquad \sigma_2 = \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_2\} \qquad T_2 = \sigma_2(T_1)}{\textbf{while}(\mathsf{c})\ \mathsf{e} \rightsquigarrow \langle \textbf{def}\ \mathsf{loop}(\overline{\mathsf{x}}_2) = \{\ \textbf{if}(\sigma_2(\mathsf{c}))\ T_2[\mathsf{loop}(\overline{\mathsf{x}}_1)]\ \textbf{else}\ \overline{\mathsf{x}}_2\}; \textbf{val}\ \overline{\mathsf{x}}_3 = \mathsf{loop}(\overline{\mathsf{x}}); \square \mid \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_3\}\rangle}$$

Figure 7. Transformation rules to rewrite imperative constructs into functional ones.

and could be the last one of a function. This is particularly true for the **if** expression which can return an expression additionally to its effects. The rules can be generalized to handle such situation by using a fourth element in the relation denoting the actual returned value if the expression was returned from a function or assigned to some variable. Note that in our system we have implemented this more general behaviour.

We have also assumed that expressions such as right hand sides of assignments and test conditions are pure expressions that do not need to be transformed. However, it is also possible to generalize the rules to handle such expressions when they are not pure, but omit this discussion. Again, in our implementation we support this more general transformation. Note that pattern matching is simply a generalized conditional expression in Leon, we do not present the rule here but Leon implements complete translation rules for pattern matching. We assume that **if**(c) t is rewritten to **if**(c) t **else** () with () corresponding to the Unit literal.

5.3 Function Hoisting

Nested functions can read immutable variables from the enclosing scope, for example the formal parameters or a let-binding from an outer function. We note that since the previously described transformation rules have already run at this point, the program and in particular nested functions are free of side-effects.

The function hosting phase starts by propagating the precondition of the enclosing function to the nested function. We also track path conditions until the definition. This outer precondition is indeed guaranteed to hold within the nested function. We then *close* nested functions, which consists in augmenting the signature of functions with all variables read from the enclosing scope. Function invocations are also updated accordingly to include these additional arguments. As a result, nested functions become self-contained and can be hoisted to the top level.

We note that this transformation causes nested functions to be treated modularly, similarly to functions that were not nested originally. It thus prevents Leon from exploiting the fact that these functions could only be called from a finite number of program points.

This is illustrated in the following example:

```
def f(x: Int) = {
  require(x > 0)
  def g(y: Int) = {
    y * 2
  } ensuring(_ > y)
  g(x)
}
```

which becomes, after hoisting:

```
def g(y: Int, x: Int) = {
  require(x > 0)
  y * 2
} ensuring(_ > y)

def f(x: Int) = {
  require(x > 0)
  g(x, x)
}
```

Even though g is originally only called with positive values, this fact is not propagated to the new precondition. Leon thus reports a spurious counter-example in the form of $\mathsf{y} = -1$.

5.4 Arrays

We support functional arrays in the core solver by mapping them to the theory of maps over the domain of integers. In order to support the .size operation, arrays are encoded as a pair of an integer, for the length, and of a map representing the contents of the array. This is necessary since maps have an implicit domain that spans the set of all integers. Maintaining this symbolic information for the size lets us generate verification conditions for accesses, thus allowing us to prove that array accesses are safe.

Imperative arrays are supported through another transformation phase. We rewrite (imperative) array updates as assignments and functional updates. The imperative transformation phase described in the previous paragraphs then handles those assignments as any other assignments.

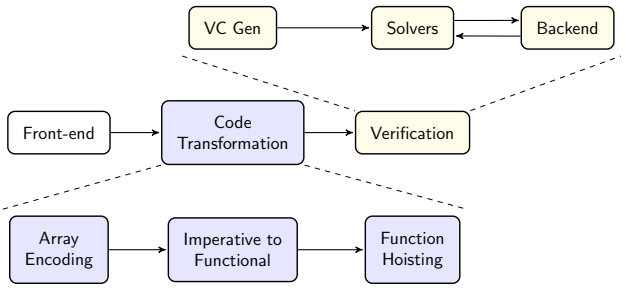


Figure 8. Overall architecture of Leon.

6. Leon Tool Architecture and Features

In this section we describe the implementation of the different parts that make up the pipeline of Leon. The overall architecture is displayed in Figure 8.

Front end. The front end to Leon relies on the early phases of the official Scala compiler —up to and including refchecks. We connected them to a custom phase that filters the Scala abstract syntax trees, rejects anything not supported by Leon, and finally produces Leon abstract syntax trees. This architecture allows us to rely entirely on the reference implementation for parsing, type inference, and type checking.

Core solver. The core solver, described in Section 4, relies on the Z3 SMT solver [10]. Communication between Leon and Z3 is done through the Scala^{Z3} native interface [24]. As more clauses are introduced to represent function unfoldings, new constraints are pushed to the underlying solver. We have found it crucial for performance to implement this loop using as low-level functions as possible; by using substitutions over Z3 trees directly as opposed to translating back-and-forth into Leon trees, we have lowered solving times by on average 30% and sometimes up to 60% on comparable hardware compared to our previous effort [35].

Code generator. Several components in Leon need or benefit from accessing an evaluator; a function that computes the truth value of ground terms. In particular, the core solver uses ground evaluation in three cases:

- Whenever a function term is ground, instead of unfolding it using the clausal representation, we invoke the evaluator and push a simple equality to the context instead. This limits the number of boolean control literals, and generally simplifies the context.
- Whenever an over-approximation for a constraint is established to be satisfiable, we cannot in general trust the result to be valid (see Section 4). In such situations, we evaluate the constraint with the obtained model to check if, by chance, it is in fact valid.
- As an additional precaution against bugs in the solver, we validate all models through evaluation.

To ensure fast evaluation, Leon compiles all functions using on-the-fly Java bytecode generation. Upon invocation, the evaluator uses reflection to translate the arguments into the Java runtime representation and to invoke the corresponding method. The results are then translated back into Leon trees.

Termination checker. Proving that all functions terminate for all inputs is a prerequisite to sound analysis. While termination was previously simply *assumed*, the latest version of Leon includes a basic termination checker, which works by identifying decreasing arguments in recursive calls. Our implementation is far from the state of the art, but is an important step towards a fully integrated verification system for a subset of Scala.

Web interface. The most convenient way to use Leon is via its public web interface[2]. It provides an editor with continuous compilation similar to modern IDEs. The web server is implemented using the Play framework[3]. Leon runs inside a per-user actor on the server side, and communicates with the client through web-sockets.

The interface also performs continuous verification: it displays an overview of the verification results and updates it asynchronously as the program evolves. Upon modification, the server computes a conservative subset of affected functions, and re-runs verification on them. We identify four different verification statuses: *valid*, *invalid*, *timeout*, and *conditionally-valid*. This last status is assigned to functions which were proved correct modularly but invoke (directly or transitively) an invalid function.

For invalid functions, we include a counter-example in the verification feedback. The web interface shows them for selected functions. Screen captures of the web interface can be found in the appendix.

7. Evaluation

We used Leon to prove correctness properties about purely functional as well as imperative data structures. Additionally, we proved full functional correctness of textbook sorting algorithms (insertion sort and merge sort). To give some examples: we proved that insertion into red-black trees preserves balancing, coloring properties, and implements the proper abstract set interface.

Our results are summarized in Table 1. The benchmarks were run on a computer equipped with two Intel Core 2 Duo CPU running at 2.53GHz and 4.0 GB of RAM. We used Z3 version 4.2. The column V/I indicates the number of valid and invalid postconditions. The column #VCs refers to additional verification conditions such as preconditions, match exhaustiveness and loop invariants. All benchmarks are available and can be run from the web interface.

[2] `http://lara.epfl.ch/leon/`

[3] `http://www.playframework.com/`

Benchmark	LoC	V/I	#VCs	Time (s)
Imperative				
ListOperations	146	6/1	16	0.62
AssociativeList	98	3/1	9	0.80
AmortizedQueue	128	10/1	21	2.57
SumAndMax	36	2/0	2	0.21
Arithmetic	84	4/1	8	0.58
Functional				
ListOperations	107	12/0	11	0.43
AssociativeList	50	4/0	5	0.43
AmortizedQueue	114	13/0	18	1.56
SumAndMax	45	4/0	7	0.23
RedBlackTree	117	7/1	10	1.87
PropositionalLogic	81	6/1	9	0.72
SearchLinkedList	38	3/0	2	0.21
Sorting	175	13/0	17	0.48

Table 1. Summary of evaluation results.

8. Related Work

Many interactive systems that mix the concept of computable functions with logic reasoning have been developed, ACL2 [20] being one of the historical leader. Such systems have practical applications in industrial hardware and software verification [21]. ACL2 requires manual assistance because it is usually required to break down a theorem into many small lemmas that are individually proven. Other more recent systems for functional programming include VeriFun [36] and AProVE [14]. Isabelle [30] and Coq [7] are proof assistant systems that provide a sort of programming language in higher order logic to express theorem and help proving them. This logic is expressive enough to define some computable functions in a similar way as it would be done in functional programming. It is also possible to automatically generate code for such systems.

A trait common to these systems is that the outcome is relatively difficult to predict. These systems provide very expressive input languages that make it very hard to automatically solved in general. Many of these systems are also very good at automating the proof of some valid properties, mostly by a smart usage of induction, while our system is complete for finding counter-examples. We think that our approach is more suited for practical programmers, that may not be verification experts but that would be able to make sense out of counter-example.

Several tools exist for the verification of contracts and invariants in imperative programs. One such tool is Dafny [25]. Dafny supports an imperative language as well as many object-oriented features. It is thus able to reason about class invariant and mutable fields, which Leon does not support so far. Dafny translates its input program to an intermediate language, Boogie [4], from which verifications conditions are then generated. The generation of verification conditions is done via the standard weakest precondition semantics [11, 29]. Our approach, on the other hand, translates the imperative code into functional code and does not make use of predicate transformers. Additional features of our translation, as well as support for disciplined non-determinism are presented in [8].

From early days, certain programming languages have been designed with verification in mind. Such programming languages usually have built-in features to express specifications that can be verified automatically by the compiler itself. These languages include Spec# [5], GYPSY [2] and Euclid [26]. Eiffel [27] popularized design by contract, where contracts are preconditions and postconditions of functions as language annotations. On the other hand, we have found that Scala's contract functions, defined in the library, work just as well as built-in language contracts and encourage experimenting with further specification constructs [31].

We expect that the idea of using encoding into functional constraints will continue to prove practical for more complex constructs. Such techniques have been used even for translation into simpler constraints, including finite-state programs [3], set constraints [1], and Horn clauses [16, 17]. Many of these constraints can be expressed as Leon programs; we plan to explore this connection in the future.

9. Conclusions

We presented Leon, a verification system for a subset of Scala. Leon reasons on both functional programs and certain imperative constructs. It translates imperative constructs into functional code. Our verification procedure then validates the functional constraints. The verification algorithm supports recursive programs on top of decidable theories and is a semi-decision procedure for satisfiability; it is complete for finding counter-examples to program correctness. Experiments show that Leon is fast for practical use, providing quick feedback whether the given programs and specifications are correct or incorrect. The completeness for counter-examples and the use of the same implementation and specification language makes Leon a practical tool that can be used by developers without special training.

We have introduced several techniques that improved the performance of Leon, including efficient unfolding of bodies of recursive calls by appropriate communication with the Z3 SMT solver. The main strength of Leon among different verification tools is the ability to predictably find counterexamples, as well the ability to prove correctness properties that do not require complex inductive reasoning. We believe that the current version of Leon, at the very least, has potential in modeling algorithms and systems using functional Scala as the modeling language, as well as a potential in teaching formal methods. Thanks to the use of modular per-function verification methods, Leon can, in principle, scale to arbitrarily large Scala programs.

References

[1] A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Programming*, 35:79–111, 1999.

[2] A. L. Ambler. GYPSY: A language for specification and implementation of verifiable programs. In *Language Design for Reliable Software*, pages 1–10, 1977.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. 2001.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[5] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[7] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[8] R. W. Blanc. Verification of Imperative Programs in Scala. Master's thesis, EPFL, 2012.

[9] E. Börger and R. Stärk. *Abstract State Machines*. 2003.

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

[11] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, N.J, 1976.

[12] B. Dutertre and L. M. de Moura. The Yices SMT solver, 2006.

[13] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.

[14] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA*, pages 210–220, 2004.

[15] M. Gordon and H. Collavizza. Forward with Hoare. In A. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 102–121. Springer, 2010.

[16] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[17] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.

[18] K. Havelund. Closing the gap between specification and programming: VDM++ and scala. In *Higher-Order Workshop on Automated Runtime Verification and Debugging*, 2011.

[19] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1986.

[20] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[21] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[22] E. Kneuss, V. Kuncak, I. Kuraj, and P. Suter. On integrating deductive synthesis and verification systems. Technical Report EPFL-REPORT-186043, EPFL, 2013.

[23] A. S. Köksal. Constraint programming in Scala. Master's thesis, EPFL, 2011.

[24] A. S. Köksal, V. Kuncak, and P. Suter. Scala to the power of Z3: Integrating SMT and programming. In *CADE*, pages 400–406, 2011.

[25] K. R. M. Leino. Developing verified programs with Dafny. In *HILT*, pages 9–10, 2012.

[26] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Inf.*, 10:1–26, 1978.

[27] B. Meyer. *Eiffel: the language*. Prentice-Hall, 1991.

[28] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI*, pages 333–344, 1998.

[29] G. Nelson. A generalization of Dijkstra's calculus. *TOPLAS*, 11(4):517–561, 1989.

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

[31] M. Odersky. Contracts for scala. In *RV*, pages 51–57, 2010.

[32] C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

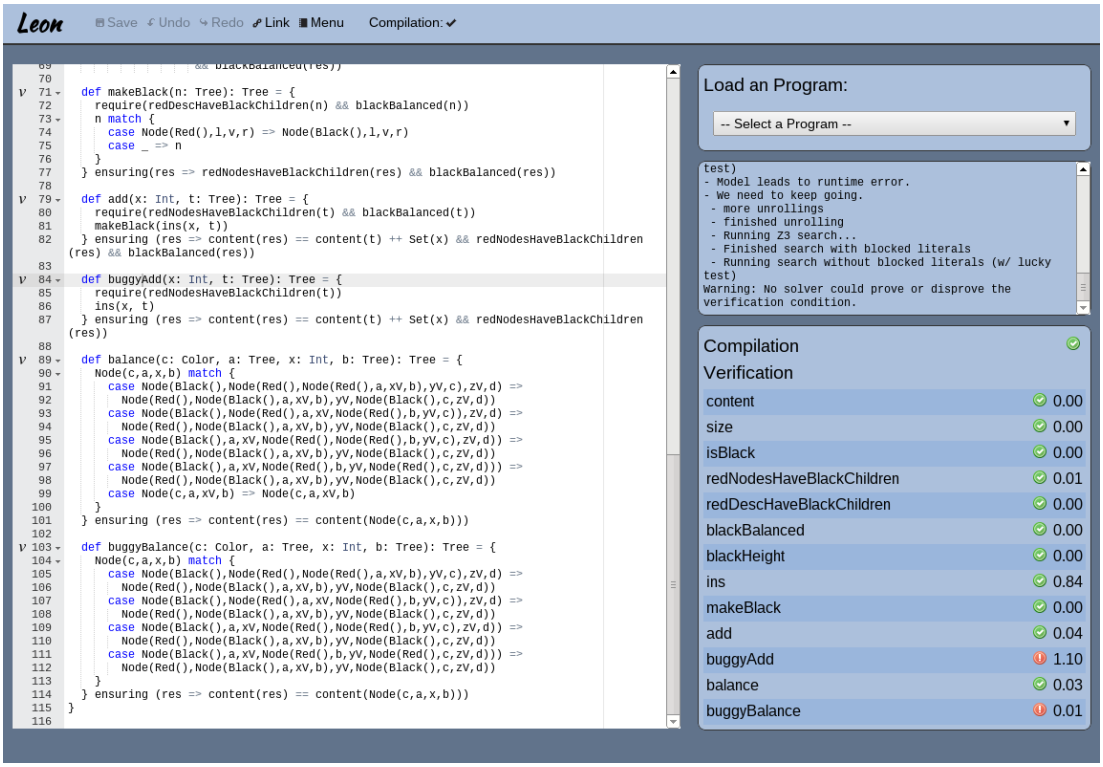
[33] P. Suter. *Programming with Specifications*. PhD thesis, EPFL, 2012.

[34] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, 2010.

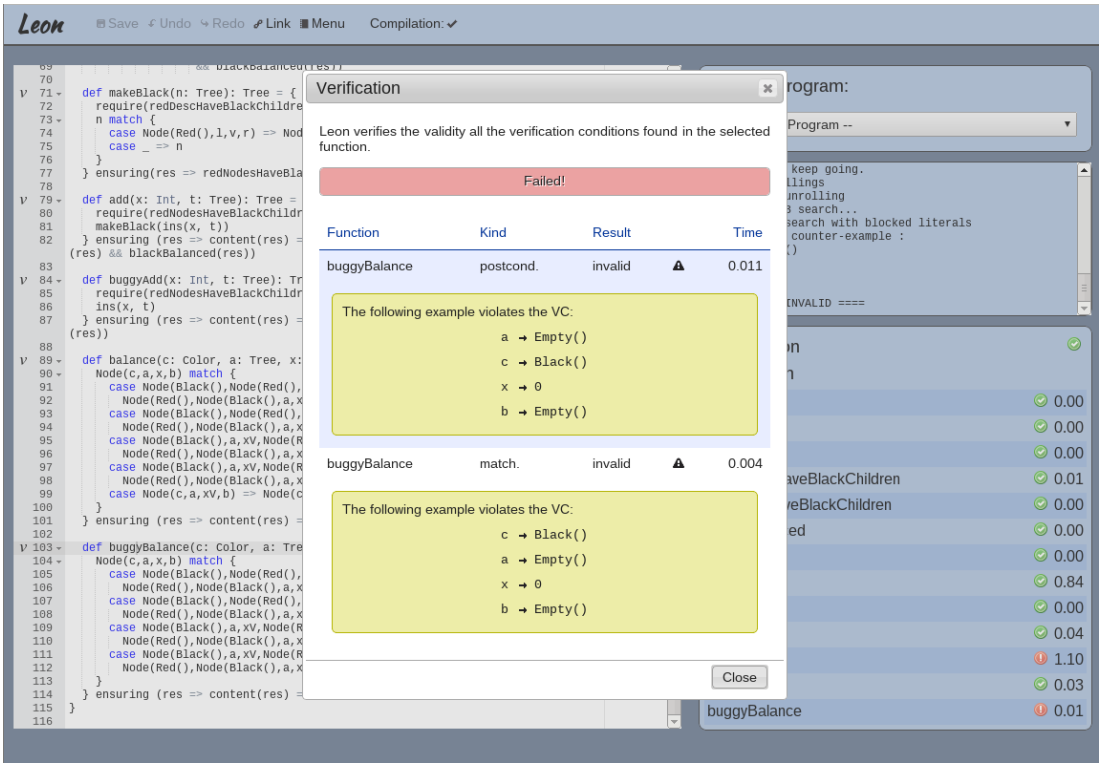
[35] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.

[36] C. Walther and S. Schweitzer. About VeriFun. In *CADE*, pages 322–327, 2003.

A. Appendix — Screenshots



Web Interface: the pane on the right displays continuous verification results. It refreshes and automatically updates the results as the program evolves.



Detailed verification results for one function with counter-examples.