Satisfiability Modulo Recursive Programs

Philippe Suter*, Ali Sinan Köksal, and Viktor Kuncak

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{firstname.lastname}@epfl.ch

Abstract. We present a semi-decision procedure for checking satisfiability of expressive correctness properties of recursive first-order functional programs. In our approach, both properties and programs are expressed in the same language, a subset of Scala. We implemented our procedure and integrated it with the Z3 SMT solver and the Scala compiler. Our procedure is sound for counterexamples and for proofs of terminating functions. It is terminating and thus complete for many important classes of specifications, including all satisfiable formulas and all formulas where recursive functions satisfy certain syntactic restrictions. Using our system, Leon, we verified detailed correctness properties for functional data structure implementations, as well as syntax tree manipulations. We have found our system to be fast for both finding counterexamples and finding correctness proofs, and to scale to larger programs than alternative techniques.

1 Introduction

This paper explores the problem of reasoning about functional programs. We reduce this problem to solving constraints representing precisely program semantics. Our starting point are SMT solving tools [22, 3, 10], which proved to be important drivers of advances in verification of software and hardware [2,12]. SMT solvers are efficient for deciding quantifier-free formulas in the language of useful theories, such as linear integer arithmetic, algebraic data types, and finite sets. Nonetheless, the operations available in the existing theories are limited, which prevents verification against more detailed specifications, as well as automated discovery of more complex invariants needed for verification. To increase the power of SMT-based reasoning, we extend the expressive power of formulas and allow them to contain user-defined recursive functions. By insisting on computable (as opposed to arbitrarily axiomatizable) functions, we obtain *familiarity* to developers, as well as *efficiency* and *completeness* of reasoning.

Our technique extends SMT solvers with recursive function definitions, so it can be used for all tasks where SMT solvers are used, including verification of functional and imperative programs, synthesis, and test generation. In this paper we introduce this technique and its concrete implementation as a verifier, named Leon, for a functional subset of Scala [25]. Leon enables the developer to state the properties as pure Scala functions and compile them using the standard Scala

* Philippe Suter was supported by the Swiss NSF Grant 200021_120433.

compiler. Developers never leave the familiar world of Scala; they use the existing library for dynamically checking executable contracts [24] to describe the desired properties. As a result, run-time contract violations can be found using testing, which would be difficult if there was a significant gap between annotation and implementation language [32]. Because we can use other functions to specify properties of the functions of interest, we obtain a very expressive specification language. We can use abstraction functions to specify abstract views of data structures. We can naturally specify properties such as commutativity and idempotence of operations, which require multiple function invocations and are not easy to express in type systems and many other specification approaches.

Leon generates verification conditions that enforce 1) that the functions meet their contracts, 2) that the preconditions at all function invocations are met, and that 3) the pattern-matching is complete for given preconditions. Note that, to define an external property of a set of functions, the developer can write a boolean-valued test function that invokes the functions of interest, and state a contract that the function always returns **true**. Leon searches in parallel for proofs and counterexamples for all generated verification conditions.

Contributions. The core technical result of our paper is an algorithm for combined proof and counterexample search, as well as a theoretical and experimental analysis of its effectiveness. We summarize our contributions as follows:

- We introduce a procedure for satisfiability modulo computable functions, which integrates DPLL(T) solving with unfolding of function definitions and validation of candidate models, which returns a model or UNSAT.
- We establish that our satisfiability procedure is:
  1. sound for models: every model it returns makes the formula true;
  2. terminating for all formulas that are satisfiable;
  3. sound for proofs: if it reports UNSAT, then there are no models;
  4. terminating for all sufficiently surjective abstractions [28];
  5. satisfying the above properties if the declared functions are always terminating; more generally, UNSAT implies no "terminating models", moreover, each returned model leads to **true**.
- We describe the implementation of our system, named Leon, as a plugin for the Scala compiler, which uses only existing constructs in Scala for specifying functions and properties. The system integrates with the SMT solver Z3.
- We present our results in verifying over 60 functions manipulating integers, sets, and algebraic data types, with detailed invariants of complex data structures such as red-black trees and amortized heap, and user code such as syntax tree manipulation. Leon verified detailed correctness properties about the content of data as well as completeness of pattern-matching expressions.

We have found that Leon was fast for finding both counterexamples and proofs for verification conditions. We thus believe that the algorithm holds great promise for practical verification of complex properties of computer systems. Leon and all benchmarks are available from `http://lara.epfl.ch`.

2 Examples

We now illustrate how Leon can be used to prove interesting properties about functional programs. Consider the following recursive datatype, written in Scala syntax [25], that represent formulas of propositional logic:

```
sealed abstract class Formula
case class And(lhs: Formula, rhs: Formula) extends Formula
case class Or(lhs: Formula, rhs: Formula) extends Formula
case class Implies(lhs: Formula, rhs: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class PropVar(id: Int) extends Formula
```

We can write a recursive function that simplifies a formula by rewriting implications into disjunctions as follows:

```
def simplify(f: Formula): Formula = (f match {
  case And(lhs, rhs) ⇒ And(simplify(lhs), simplify(rhs))
  case Or(lhs, rhs) ⇒ Or(simplify(lhs), simplify(rhs))
  case Implies(lhs, rhs) ⇒ Or(Not(simplify(lhs)), simplify(rhs)) // note the replacement
  case Not(f) ⇒ Not(simplify(f))
  case PropVar(_) ⇒ f
}) ensuring(isSimplified(_))
```

Note that **ensuring** is an infix command in Scala, taking the entire function body as the left argument and taking a lambda function as the right argument. The expression e **ensuring** p indicates that p(e) should hold for all values of the function parameters. To denote an anonymous function, $\lambda x.B$, in Scala we write x⇒B. When there is only one occurrence of x in B, we can denote this occurrence by _ and omit the binder, so both (_+1) and x⇒x+1 denote the increment function.

We can write an executable function isSimplified that checks whether a given formula contains an implication as a subformula, and use it in a contract. The **ensuring** statement in the example is a postcondition written in Scala notation [24], stating that the function isSimplified evaluates to true on the result. We define isSimplified recursively as follows:

```
def isSimplified(f: Formula): Boolean = f match {
  case And(lhs, rhs) ⇒ isSimplified(lhs) && isSimplified(rhs)
  case Or(lhs, rhs) ⇒ isSimplified(lhs) && isSimplified(rhs)
  case Implies(_,_) ⇒ false
  case Not(f) ⇒ isSimplified(f)
  case PropVar(_) ⇒ true
}
```

Note that a developer would also typically write such an executable specification function for testing purposes. Using our procedure for satisfiability modulo computable functions, Leon can prove that the postcondition of simplify is satisfied for *every* input formula f.

Such subsets of values denoted by algebraic data types are known as refinement types [13]. Refinement types that are defined using functions such as isSimplified are in fact sufficiently surjective abstractions [28], which implies that our system is a decision procedure for such constraints (see Section 3.1). This is confirmed with our experiments—our tool is predictably fast on such examples.

Suppose now that we wish to prove that simplifying a simplified formula does not change it further. In other words, we want to prove that the property simplify(simplify(f)) == simplify(f) holds for all formulas f. Because our programming and specification languages are identical, we can write such universally quantified statements as functions that return a boolean and whose postcondition is that they always return true. In this case, we would write:

```
def simplifyIsStable(f: Formula) : Boolean = {simplify(simplify(f)) == simplify(f)} holds
```

Because such specifications are common, we use the notation **holds** instead of the more verbose postcondition stating that the returned result should be an identity function with boolean argument **ensuring**(res⇒res). Our verification system proves this property instantly.

Another application for our technique is verifying that pattern-matching expressions are defined for all cases. Pattern-matching is a very powerful construct commonly found in functional programming languages. Typically, evaluating a pattern-matching expression on a value not covered by any case raises a runtime error. Because checking that a **match** expression never fails is difficult in non-trivial cases (for instance, in the presence of guards), compilers in general cannot statically enforce this property. For instance, consider the following function that computes the set of variables in a propositional logic formula, assuming that the formula has been simplified:

```
def vars(f: Formula): Set[Int] = {
  require(isSimplified(f))
  f match {
    case And(lhs, rhs) ⇒ vars(lhs) ++ vars(rhs)
    case Or(lhs, rhs) ⇒ vars(lhs) ++ vars(rhs)
    case Not(f) ⇒ vars(f)
    case PropVar(i) ⇒ Set[Int](i) }}
```

Here ++ denotes the set union operation in Scala. Although it is implied by the precondition that all cases are covered, the Scala compiler on this example will issue the warning:

```
Logic.scala: warning: match is not exhaustive!
missing combination         Implies
```

Previously, researchers have developed specialized analyses for checking such exhaustiveness properties [9, 11]. Our system generates verification conditions for checking the exhaustiveness of all pattern-matching expressions, and then uses the same procedure to prove or disprove them as for the other verification conditions. It quickly proves that this particular example is exhaustive by unrolling the definition of isSimplified sufficiently many times to conclude that t can never be an Implies term. Note that our system will also prove that all recursive calls to vars satisfy its precondition; it performs sound assume-guarantee reasoning.

Consider now the following function, that supposedly computes a variation of the negation normal form of a formula f:

```
def nnf(formula: Formula): Formula = formula match {
  case Not(Not(f)) ⇒ nnf(f)
  case And(lhs, rhs) ⇒ And(nnf(lhs), nnf(rhs))
  case Not(And(lhs, rhs)) ⇒ Or(nnf(Not(lhs)), nnf(Not(rhs)))
  ...
  case Implies(lhs, rhs) ⇒ Implies(nnf(lhs), nnf(rhs))
}
```

From the supposed roles of the functions simplify and nnf, one could conjecture that the operations are commutative. Because of the treatment of implications in the above definition of nnf, though, this is not the case. We can disprove this property by finding a counterexample to

```
def wrongCommutative(f: Formula) : Boolean = {
  nnf(simplify(f)) == simplify(nnf(f))} holds
```

On this input, Leon reports

```
Error: Counter-example found:
f -> Implies(Not(And(PropVar(48), PropVar(47))),
             And(PropVar(46), PropVar(45)))
```

A consequence of our algorithm is that Leon never reports false positives (see Section 3.1). In this particular case, the counterexample clearly shows that there is a problem with the treatment of implications whose left-hand side contains a negation. Counterexamples such as this one are typically short and Leon finds them quickly.

As a final example of the expressive power of our system, we consider the question of showing that an implementation of a collection implements the proper interface. Consider the implementation of a set as red-black trees. (We omit the datatype definition in the interest of space.) To specify the operation on the trees in terms of the set interface that they are supposed to implement, we define an *abstraction function* that computes from a tree the set it represents:

```
def content(t : Tree) : Set[Int] = t match {
  case Empty() ⇒ Set.empty
  case Node(_, l, v, r) ⇒ content(l) ++ Set(v) ++ content(r) }
```

Note that this is again a function one would write for testing purposes. The specification of insertion using this abstraction becomes very natural, despite the relative complexity of the operations:

```
def ins(x: Int, t: Tree): Tree = (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))
```

We also wrote functions that check whether a tree is balanced and whether it satisfies the coloring properties. We used these checks to specify insertion and balancing operations. Leon proved all these properties of red-black tree operations. We present more such results in Section 5.

3 Our Satisfiability Procedure

In this section, we describe our algorithm for checking the satisfiability of formulas modulo recursive functions. We start with a description of the supported class of formulas. Let $\mathcal{L}$ be a *base theory* (logic) with the following properties:

- $\mathcal{L}$ is closed under propositional combination and supports boolean variables
- $\mathcal{L}$ supports uninterpreted function symbols
- there exists a complete decision procedure for $\mathcal{L}$

Note that the logics supported by DPLL(T) SMT solvers naturally have these properties.[1]

Let $\mathcal{L}^{\Pi}$ be the extension of $\mathcal{L}$ with *interpreted* function symbols defined in a program Π. The interpretation is given by an expression in $\mathcal{L}^{\Pi}$ (the *implementation*). To facilitate proofs and the description of program invariants, functions in Π can also be annotated with a *pre-* and *postcondition*. We denote the implementation, pre- and postcondition of a function f in Π by impl^{Π}_{f}, prec^{Π}_{f} and post^{Π}_{f} respectively. The free variables of these expressions are the arguments of f denoted args^{Π}_{f}, as well as, for the postcondition, a special variable ρ that denotes the result of the computation.

```
def solve(φ, Π) {
  (φ, B) = unrollStep(φ, Π, ∅)
  while(true) {
    decide(φ ∧ ⋀_{b∈B} b) match {
      case "SAT" ⇒ return "SAT"
      case "UNSAT" ⇒ decide(φ) match {
        case "UNSAT" ⇒ return "UNSAT"
        case "SAT" ⇒ (φ, B) = unrollStep(φ, Π, B) }}}}
```

Fig. 1. Pseudo-code of the solving algorithm. The decision procedure for the base theory is invoked through the calls to decide.

Figure 1 shows the pseudo-code of our algorithm. It is defined in terms of two subroutines, decide, which invokes the decision procedure for $\mathcal{L}$, and unrollStep, whose description follows. Note that the algorithm maintains, along with a formula ϕ, a set B of boolean literals. We call these *control literals* and their role is described below.

At a high-level, the role of unrollStep is to give a partial interpretation to function invocations, which are treated as uninterpreted in $\mathcal{L}$. This is achieved in two steps: 1) introduce definitions for one unfolding of the (uninterpreted) function invocations in ϕ and 2) generate an assignment of control literals that

[1] In Leon, $\mathcal{L}$ is the multi-sorted combination of uninterpreted function symbols with integer linear arithmetic and user-defined algebraic datatypes and finite sets.

$$
\begin{array}{ll}
\mathsf{size}(\mathsf{lst}) = \mathsf{lst}\ \textbf{match}\ \{ & \psi \equiv \mathsf{size}(\mathsf{lst}) = t_{\mathsf{f}} \\
\quad \textbf{case}\ \mathsf{Nil} \Rightarrow 0 & \quad \wedge\ b_{\mathsf{f}} \Leftrightarrow \mathsf{lst} = \mathsf{Nil} \\
\quad \textbf{case}\ \mathsf{Cons}(\_,\ \mathsf{xs}) \Rightarrow 1 + \mathsf{size}(\mathsf{xs}) & \quad \wedge\ b_{\mathsf{f}} \Rightarrow t_{\mathsf{f}} = 0 \\
\} & \quad \wedge\ \neg b_{\mathsf{f}} \Rightarrow t_{\mathsf{f}} = 1 + \mathsf{size}(\mathsf{lst.tail})
\end{array}
$$

Fig. 2. Function definition and its translation into clauses with control literals.

guard newly introduced function invocations. As an example, consider a formula ϕ that contains the term size(lst), where lst is a list and size is the usual recursive definition of its length. Figure 2 shows on the left the definition of size(lst) and on the right its encoding into clauses with fresh variables.

This encoding into clauses is obtained by recursively introducing, for each if-then-else term, a boolean variable representing the truth value of the branching condition, and another variable representing the value of the if-then-else term.

In addition to conjoining ψ to ϕ, unrollStep would produce the set of literals $\{b_{\mathsf{f}}\}$. The set should be understood as follows: the decision procedure for the base theory $\mathcal{L}$, which treats size as an uninterpreted function symbol, if it reports SAT, can only be trusted when b_{f} is set to true. Indeed, if b_{f} is false, the value used for t_{f} and hence for the term size(lst) may depend on size(lst.tail), which is undefined (because its definition has not yet been introduced). A subsequent call to unrollStep on $\phi \wedge \psi$ would introduce the definition of size(lst.tail). When unrolled functions have a precondition, the definitions introduced for their body and postcondition are guarded by the precondition. This is done to prevent an inconsistent function definition with a precondition equivalent to **false** from making the formula unsatisfiable.

The formula in $\mathcal{L}$ without the control literals can be seen as an *under-approximation* of the formula with the semantics of the program defining $\mathcal{L}^{\Pi}$, in that it accepts all the same models plus some models in which the interpretation of some invocations is incorrect, and the formula with the control literals is an *over-approximation*, in the sense that it accepts only the models that do not rely on the guarded invocations. This explains why the UNSAT answer can be trusted in the first case and the SAT case in the latter.

In Figure 1, the third argument of calls to unrollStep denotes the set of control literals introduced at previous steps. An invocation of unrollStep will insert definitions for *all* or only *some* function terms that are guarded by such control literals, and the returned set will contain all literals that were not released as well as the newly introduced ones. From an abstract point of view, when a new control literal is created, it is inserted in a global priority queue with all the function invocations that it guards. An important requirement is that the dequeuing must be *fair*: any control literal that is enqueued must eventually be dequeued. This fairness condition guarantees that our algorithm is complete for satisfiable formulas (see Section 3.1). Using a first-in first-out policy, for instance, is enough to guarantee fairness and thus completeness. Finally, note that unrollStep not only adds definitions for the implementation of the function calls, but also for their postcondition when it exists. We discuss the issue of reliably using these facts in Section 3.1.

Implementation notes. While the description of solve suggests that we need to query the solver twice in each loop iteration, we can in practice use the solver's capability to output *unsat cores* to detect with a single query whether the conjunction of control literals $\bigwedge_{b \in B} b$ played any role in the unsatisfiability. Similarly, when adding the new constraints obtained from unrollStep, we can use the solver's incremental reasoning and push the new constraints directly, rather than building a new formula and issuing a new query. SMT solvers can thus exploit at any time facts learned in previous iterations.

Finally, although we noted that we cannot in general trust the underlying solver when it reports SAT for the formula ϕ without control literals, it could still be that the assignment the solver guessed for the uninterpreted function symbols is valid. Because testing an assignment is fast (it amounts to executing the specification), we can therefore sometimes report SAT early and save time.

3.1 Properties of Our Procedure

The properties of our procedure rely on the following two assumptions.

Termination: All precondition computations terminate for all values. Each function in the program Π terminates on each input for which the precondition holds, and similarly for each postcondition. Tools such as [14,1] or techniques developed for ACL2 [20] could be used to establish this property.

Base theory solver soundness: The underlying solver is complete and sound for the (quantifier-free) formulas in the base theory. The completeness means that each model that the solver reports should be a model for the conjunction of all constraints passed to the solver. Similarly, soundness means that whenever the solver reports unsatisfiability, false can be logically concluded modulo the solver's theories from these constraints.

We use the above assumptions throughout this section. Note, however, that even without the termination assumption, a counterexample reported by Leon is never a counterexample that generates a terminating execution of the property resulting in the value true, so it is a counterexample worth inspecting.

Soundness for Proofs. Our algorithm reports unsatisfiability if and only if the underlying solver could prove unsatisfiable the problem given to it *without* the control literals. Because the control literals are not present, some function applications are left uninterpreted, and the conclusion that the problem is unsatisfiable therefore applies to *any* interpretation of the remaining function application terms, and in particular to the one conforming to the correct semantics.

From the assumption that the underlying solver only produces sound proofs, it suffices to show that all the axioms communicated to the solver as a result of the unrollStep invocations are obtained from sound derivations. These are correct by definition: they are logical consequences obtained by the definition of functions, and these definitions are conservative when the functions are terminating.

An important consideration when discussing soundness of the *post* axioms is that any proof obtained with our procedure can be considered valid only when the following properties about the functions of Π have been proved:[2]

1. for each function f of Π, the following formula must hold:

$$\mathsf{prec}_f^{\Pi} \implies \mathsf{post}_f^{\Pi}\left[\mathsf{impl}_f^{\Pi}/\rho\right]$$

2. for each call in f to a function f_2 (possibly f itself), the precondition $\mathsf{prec}_{f_2}^{\Pi}$ must be implied by the path condition
3. for each pattern-matching expression, the patterns must be shown to cover all possible inputs under the path condition.

The above conditions guarantee the absence of runtime errors, and they also allow us to prove the overall correctness by induction on the call stack, as is standard in assume-guarantee reasoning for sequential procedures without side effects [15, Chapter 12].

The first condition shows that all postconditions are logical implications of the function implementations under the assumption that the preconditions hold. The second condition shows that all functions are called with arguments satisfying the preconditions. Because all functions terminate, it follows that we can safely assume that postconditions always hold for all function calls. This justifies the soundness of axioms *post* in the presence of ϕ and Π.

Soundness for Models. Our algorithm reports satisfiability when the solver reports that the unrolled problem augmented with the control literals is satisfiable. By construction of the set of control literals, it follows that the solver can only have used values for function invocations whose definition it knows. As a consequence, every model reported by the solver for the problem augmented with the control literals is always a true model of the original formula. We mentioned in Section 3 that we can also check other satisfying assignments produced by the solver. In this second case, we use an evaluator that complies with the semantics of the program, and therefore the validated models are true models as well.

Termination for Satisfiable Formulas. Our procedure has the remarkable property that it finds a model whenever the model for a formula exists. We define a model as an assignment to the free variables of the formula such that evaluating it under that assignment terminates with the value **true**. An assignment that leads to a diverging evaluation is not considered to be a proper model. To understand why our procedure always finds models when they exist, consider a counterexample for the specification. This counterexample is an assignment of integers and algebraic data types to variables of a function $f(\boldsymbol{x})$ being proved. This evaluation specifies concrete inputs $\boldsymbol{a}$ for f such that evaluating $f(\boldsymbol{a})$ yields

[2] When proving or disproving a formula ϕ modulo the functions of Π, it is in fact sufficient that the three properties hold only for all functions in ϕ and those that can be called (transitively) from them or from their contracts.

a value for which the postcondition of f evaluates to false (the case of preconditions or pattern-matching is analogous). Consider the computation tree arising from (call-by-value) evaluation of f and its postcondition. This computation tree is finite. Consequently, the tree contains finitely many unrollings of function invocations. Let us call K the maximum depth of that tree. Consider now the execution of the algorithm in Figure 1; because we assume that any function invocation that is guarded by a control literal is eventually accessible, we can safely conclude that every function application in the original formula will eventually be unrolled. By applying this reasoning inductively, we conclude that eventually, all function applications up to nesting level $K + 1$ will be unrolled. This means that the computation tree corresponding to $f(\boldsymbol{a})$ has also been explored. By the completeness of the solver for the base theory and the consistency of a satisfying specification, it means that the solver reports a counterexample (either $\boldsymbol{a}$ itself or another counterexample).

Termination for Sufficiently Surjective Abstraction. Our procedure always terminates and is therefore a decision procedure in the case of a recursive function that are *sufficiently surjective catamoprhisms* [28]. In fact, it also serves as the first implementation of the technique [28]. A catamorphism is a fold function from a tree data type to some domain, which uses a simple recursive pattern: compute the value on subtrees, then combine these values using an expressions from a decidable logic. The sufficient surjectivity for a function f is a condition implying, informally, that the size of the set $\{x \mid f(x) = y\}$ can be made sufficiently large for "sufficiently large" elements y. Leon shows that the technique inspired by [28] is fast in practice. Moreover, by interleaving unrolling and satisfiability checking, it addresses in practice the open problem of determining the maximal amount of unrolling needed for a user-defined function.

We have already established termination in the case of formula satisfiability. In the case of an unsatisfiable formula, the termination follows because the unsatisfiability can be detected by unrolling the function definitions a finite number of times [28]. The unrolling depth depends on the particular sufficiently surjective abstraction, which is why [28] presents only a family of decision procedures and not a decision procedure for all sufficiently surjective abstractions. In contrast, our approach is one uniform procedure that behaves as a decision procedure for the *entire* family, because it unrolls functions in a fair way. [3]

Among the examples of such recursive functions for which our procedure is a decision procedure are functions of algebraic data types such as size, height, or content (expressed as a set, multiset, or a list). Further examples include properties such as sortedness of a list or a tree, or a combination of any finite

[3] Using the terminology of [28, p.207], consider the case of a sufficiently surjective catamorphism with the associated parametric formula M_p and set of shapes S_p, and an unsatisfiable formula containing a term $\alpha(t)$. Sufficiently unrolling α will result in coverage of all possible shapes of t that belong to S_p. If no satisfying assignment for t can be found with these shapes, then a formula at least as strong as $M_p(\alpha(t))$ will be implied, so a complete solver for the base theory will thus detect unsatisfiability because [28] detects unsatisfiability when using $M_p(\alpha(t))$.

number of functions into a finite domain. Through experiments with Leon, we have also discovered a new and very useful instance of constraints for which our tool is complete: the predicates expressing refinement types [13], which we specify as, e.g., the isSimplified function in Section 2. These functions map data structures into a finite domain–booleans, so they are sufficiently surjective. This explains why Leon is complete, and why it was so successful in verifying complex pattern-matching exhaustiveness constraints on syntax tree transformations.

Non-terminating Functions. We conclude this section with some remarks on the behavior of our procedure in the presence of functions that do not terminate on all their inputs. We are interested in showing that if for an input formula the procedure returns UNSAT, then there are indeed no models whose evaluation terminates with **true**. (The property that all models are true models is not affected by non-terminating functions.) Note that it may still be the case that the procedure returns UNSAT when there is an input for which the evaluation doesn't terminate.

To see why the property doesn't immediately follow from the all-terminating case, consider the definition: **def** $\mathsf{f}(\mathsf{x} : \mathsf{Int}) = \mathsf{f}(\mathsf{x}) + 1$. Unrolling that function could introduce a contradiction $\mathsf{f}(\mathsf{x}) = \mathsf{f}(\mathsf{x}) + 1$ and make the formula immediately unsatisfiable, thus potentially masking a true satisfying assignment. However, because all introduced definitions are guarded by a control literal, the contradiction will only prevent those literals from being true that correspond to executions leading to the infinite loop.

4 The Leon Verification System

We now present some of the characteristics of the implementation of Leon, our verification system that has at its core an implementation of the procedure presented in the previous sections. Leon takes as an input a program written in a purely functional subset of Scala and produces verification conditions for all specified postconditions, calls to functions with preconditions, and match expressions in the program.

Front-end. We wrote a plugin for the official Scala compiler to use as the front-end of Leon. The immediate advantage of this approach is that all programs are parsed and type-checked before they are passed to Leon. This also allows users to write expressive programs concisely, thanks to type-inference and the flexible syntax of Scala. The subset we support allows for definitions of recursive datatypes, as shown in examples throughout this paper, as well as arbitrarily complex pattern-matching expressions over such types. The other admitted types are integers and sets, which we found to be particularly useful for specifying properties with respect to an abstract interface. In our function definitions, we allow only immutable variables for simplicity (**val**s and no **var**s in Scala terminology).

Conversion of pattern-matching. We transform all pattern-matching expressions into equivalent expressions built with if-then-else terms. For this purpose, we use predicates that test whether their argument is of a given subtype (this is equivalent to the method .isInstanceOf[T] in Scala). The translation is relatively straightforward, and preserves the semantics of pattern-matching. In particular, it preserves the property that cases are tested in their definition order. To encode the error that can be triggered if no case matches the value, we return for the default case a fresh, uninterpreted value. This value is therefore guarded by the conjunction of the negation of all matching predicates. Recall that we separately prove that all match expressions are exhaustive. When these proofs succeed, we effectively rule out the possibility that the unconstrained error value affects the semantics of the expression.

Proofs by induction. To simplify the statement and proof of some inductive properties, we defined an annotation @induct, that indicates to Leon that it should attempt to prove a property by induction on the arguments. This works only when proving a property over a variable that is of a recursive type; in these cases, we decompose the proof that the postcondition is always satisfied into subproofs for the alternatives of the datatype. For instance, when proving by induction that a property holds for all binary trees, we generate a verification condition for the case where the tree is a leaf, then for the case where it is a node, assuming that the property holds for both subtrees.

Communicating with the solver. We used Z3 [22] as the SMT solver at the core of our solving procedure. As described in Section 3, we use Z3's support for incremental reasoning to avoid solving a new problem at each iteration of our top-level loop.

Interpretation of selectors as total functions. We should note that the interpretation of selector functions in Z3 is different than in Scala, since they are considered to be total, but uninterpreted when applied to values of the wrong type. For instance, the formula Nil.head = 5 is considered in Z3 to be satisfiable, while taking the head of an empty list has no meaning in Scala (if not a runtime error). This discrepancy does not affect the correctness of Leon, though, as the type-checking algorithm run by the Scala compiler succeeds only when it can guarantee that the selectors are applied only to properly typed terms.

5 Experimental Evaluation

We are very excited about the speed and the expressive power of properties that Leon can prove; this feeling is probably best understood by trying out the Leon distribution. As an illustration, we here report results of Leon on proving correctness properties for a number of functional programs, and discovering counterexamples when functions did not meet their specification. A summary of our evaluation can be seen in Figure 3. In this figure, LOC denotes the number of lines of code, #p. denotes the number of verification conditions for function invocations with preconditions, #m. denotes the number of conditions for showing exhaustiveness of pattern-matchings, V/I denotes whether the verification conditions were valid or invalid, U denotes the maximum depth for unrolling function

Benchmark *(LOC)*	#p.	#m.	V/I	U	Time	function	#p.	#m.	V/I	U	Time
ListOperations *(107)*											
size	0	1	V	1	0.12	sizeTailRecAcc	1	1	V	1	0.01
sizesAreEquiv	0	0	V	2	<0.01	sizeAndContent	0	0	V	1	<0.01
reverse	0	0	V	2	0.02	reverse0	0	1	V	2	0.04
append	0	1	V	1	0.03	nilAppend	0	0	V	1	0.03
appendAssoc	0	0	V	1	0.03	sizeAppend	0	0	V	1	0.04
concat	0	0	V	1	0.04	concat0	0	2	V	2	0.29
zip	1	2	V	2	0.09	sizeTailRec	1	0	V	1	<0.01
content	0	1	V	0	<0.01						
AssociativeList *(50)*											
update	0	1	V	1	0.03	updateElem	0	2	V	1	0.05
readOverWrite	0	1	V	1	0.10	domain	0	1	V	0	0.05
find	0	1	V	1	<0.01						
InsertionSort *(99)*											
size	0	1	V	1	0.06	sortedIns	1	1	V	2	0.24
buggySortedIns	1	1	I	1	0.08	sort	1	1	V	1	0.03
contents	0	1	V	0	<0.01	isSorted	0	1	V	1	<0.01
RedBlackTree *(117)*											
ins	2	1	V	3	2.88	makeBlack	0	0	V	1	0.02
add	2	0	V	2	0.19	buggyAdd	1	0	I	3	0.26
balance	0	1	V	3	0.13	buggyBalance	0	1	I	1	0.12
content	0	1	V	0	<0.01	size	0	1	V	1	0.11
redNHaveBlckC.	0	1	V	1	<0.01	redDHaveBlckC.	0	1	V	0	<0.01
blackHeight	0	1	V	1	<0.01						
PropositionalLogic *(86)*											
simplify	0	1	V	2	0.84	nnf	0	1	V	1	0.37
wrongCommutative	0	0	I	3	0.44	simplifyBreaksNNF	0	0	I	1	0.28
nnfIsStable	0	0	V	1	0.17	simplifyIsStable	0	0	V	1	0.12
isSimplified	0	1	V	0	<0.01	isNNF	0	1	V	1	<0.01
vars	6	1	V	1	0.13						
SumAndMax *(46)*											
max	2	1	V	1	0.13	sum	0	1	V	0	<0.01
allPos	0	1	V	0	<0.01	size	0	1	V	1	<0.01
prop0	1	0	V	1	0.02	property	1	0	V	1	0.11
SearchLinkedList *(48)*											
size	0	1	V	1	0.11	contains	0	1	V	0	<0.01
firstZero	0	1	V	1	0.03	firstZeroAtPos	0	1	V	0	<0.01
goal	0	0	V	1	0.01						
AmortizedQueue *(124)*											
size	0	1	V	1	0.14	content	0	1	V	0	<0.01
asList	0	1	V	0	<0.01	concat	0	1	V	1	0.04
isAmortized	0	1	V	0	<0.01	isEmpty	0	1	V	0	<0.01
reverse	0	1	V	3	0.20	amortizedQueue	0	0	V	2	0.05
enqueue	0	1	V	1	<0.01	front	0	1	V	3	0.01
tail	0	1	V	3	0.15	propFront	1	1	V	3	0.07
enqueueAndFront	1	0	V	4	0.21	enqDeqThrice	5	0	V	5	2.48

Fig. 3. Summary of evaluation results

definitions, and Time denotes the total running time in seconds to verify all conditions for a function. The benchmarks were run on a computer equipped with two Intel Core 2 processors running at 2.66 GHz and 3.2 GB of RAM, using a very recent version of Z3 at the time of running the experiments (June 12, 2011). We verified over 60 functions, with over 600 lines of compactly written code and properties that often relate multiple function invocations. This includes a red-black tree set implementation including the height invariant (which most reported benchmarks for automated systems omit); amortized queue data structures, and examples with syntax tree refinement that show Leon to be useful for checking user code, and not only for data structures.[4]

The ListOperations benchmark contains a number of common operations on lists. Leon proves, e.g., that a tail-recursive version of size is functionally equivalent to a simpler version, that append is associative, and that content, which computes the set of elements in a list, distributes over append. For association lists, Leon proves that updating a list l1 with all mappings from another list l2 yields a new associative list whose domain is the union of the domains of l1 and l2. It proves the *read-over-write* property, which states that looking up the value associated with a key gives the most recently updated value. We express this property simply as:

```
def readOverWrite(l : List, e : Pair, k : Int) : Boolean = (e match {
  case Pair(key, value) ⇒
    find(updateElem(l, e), k) == (if (k == key) Some(value) else find(l, k))
}) holds
```

Leon proves properties of insertion sort such as the fact that the output of the function sort is sorted, and that it has the same size and content as the input list. The function buggySortedIns is similar to sortedIns, and is responsible for inserting an element into an already sorted list, except that the precondition that the list should be sorted is missing. On the RedBlackTrees benchmark, Leon proves that the tree implements a set interface and that balancing preserves the "red nodes have no black children" and "every simple path from the root to a leaf has the same number of black nodes" properties as well as the contents of the tree. In addition to proving correctness, we also seeded two bugs (forgetting to paint a node black and missing a case in balancing); Leon found a concise counterexample in each case. The PropositionalLogic benchmark contains functions manipulating abstract syntax trees of boolean formulas. Leon proves that, e.g., applying a negation normal form transformation twice is equivalent to applying it once.

Further benchmarks are taken from the Verified Software Competition [30]: For example, in the AmortizedQueue benchmark Leon proves that operations on an amortized queue implemented as two lists maintains the invariant that the size of the "front" list is always larger than or equal to the size of the "rear" list, and that the function front implements an abstract queue interface given as a sequence.

We also point out that, apart from the parameterless @induct hint for certain functions, there are no other hint mechanisms used in Leon: the programmer

[4] All benchmarks and the sources of Leon are available from `http://lara.epfl.ch`.

simply writes the code, and boolean-valued functions that describe the desired properties (as they would do for testing purposes). We thus believe that Leon is easy and simple to use, even for programmers that are not verification experts.

6 Related Work

We next compare our approach to the most closely related techniques.

Interactive verification systems. The practicality of computable functions as an executable logic has been demonstrated through a long line of systems, notably ACL2 [18] and its predecessors. These systems have been applied to a number of industrial-scale case studies in hardware and software verification [18, 21]. Recent systems based on functional programs include VeriFun [31] and AProVE [14]. Moreover, computable specifications form important parts of many case studies in proof assistants Coq [5] and Isabelle [23]. These systems support more expressive logics, with higher-order quantification, but provide facilities for defining executable functions and generating the corresponding executable code in functional programming languages [16]. When it comes to reasoning within these systems, they offer varying degrees of automation. What is common is the difficulty of predicting when a verification attempt will succeed. This is in part due to possible simplification loops associated with the rewrite rules and tactics of these provers. Moreover, for performance and user-interaction reasons, interactive proofs often fail to fully utilize aggressive case splitting that is at the heart of modern SMT solvers.

Inductive generalizations vs. counterexamples. Existing interactive systems such as ACL2 are stronger in automating induction, whereas our approach is complete for finding counterexamples. We believe that the focus on counterexamples will make our approach very appealing to programmers that are not theorem proving experts. The HMC verifier [17] and DSolve [26] can automatically discover inductive invariants, so they have more automation, but it appears that certain properties involving multiple user-defined functions are not expressible in these systems. Recent results also demonstrate inference techniques for higher-order functional programs [19, 17]. These approaches hold great promise for the future, but the programs on which those systems were evaluated are smaller than our benchmarks. Leon focuses on first-order programs and is particularly effective for finding counterexamples. Our experience suggests that Leon is more scalable than the alternative systems that can deal with this expressive properties. Counterexample generation has been introduced into Isabelle through tools like Nitpick [6]. Further experimental comparisons would be desirable, but these techniques do not use theory solvers and appear slower than Leon on complex functional programs. Counterexample generation has been recently incorporated into ACL2 Sedan [7]. This techniques is tied to the sophisticated ALC2 proof mechanism and uses proof failures to find counterexamples. Although it appears very useful, it does not have our completeness guarantees.

Counterexample finding systems for imperative code. Researchers have explored the idea of iterative function and loop unfolding in a number of contexts. Among well-known tools is CBMC [8]; techniques to handle procedures include [29, 4, 27]. The use of imperative languages in these systems makes their design more complex and limits the flexibility of the counterexample search. Thanks to a direct encoding into SMT and the absence of side-effects, Leon can prove more easily properties that would be harder to prove using imperative semantics. As a result, we were able to automatically prove detailed functional correctness properties as opposed to only checking for errors such as null dereferences. Moreover, both [29] and [27] focus on error finding, while we were also able to prove several non-trivial properties correct, using counterexample finding to debug our code and specifications during the development.

Satisfiability modulo theory solvers. SMT solvers behave as (complete) decision procedures on certain classes of quantifier-free formulas containing theory operations and uninterpreted functions. However, they do not support user-defined functions, such as functions given by recursive definitions. An attempt to introduce them using quantifiers leads to formulas on which the prover behaves unpredictably for unsatisfiable instances, and is not able to determine whether a candidate model is a real one. This is because the prover has no way to determine whether universally quantified axioms hold for all of the infinitely many values of the domain. Leon uses terminating executable functions, whose definitions are a well-behaved and important class of quantified axioms, so it can check the consistency of a candidate assignment. A high degree of automation and performance in Leon comes in part from using state-of-the-art SMT solver Z3 [22] to reason about quantifier-free formula modulo theories, as well as to perform case splitting along with automated lemma learning. Other SMT solvers, such as CVC3 [3] could also be used.

Acknowledgments. We thank Nikolaj Bjørner and Leonardo de Moura for their help with Z3. We thank Mirco Dotta and Swen Jacobs for preliminary versions of some of the benchmarks. We thank Panagiotis Manolios and J Strother Moore for discussions about ACL2.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: Formal Methods for Components and Objects. pp. 113–132 (2007)
2. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: CAV (2010)
3. Barrett, C., Tinelli, C.: CVC3. In: CAV. LNCS, vol. 4590 (2007)
4. Basler, G., Kroening, D., Weissenbacher, G.: A complete bounded model checking algorithm for pushdown systems. In: Haifa Verification Conference (2007)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
6. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: ITP (2010)

7. Chamarthi, H.R., Dillinger, P.C., Manolios, P., Vroon, D.: The acl2 sedan theorem proving system. In: TACAS. pp. 291–295 (2011)
8. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: TACAS. pp. 168–176 (2004)
9. Dotta, M., Suter, P., Kuncak, V.: On static analysis for expressive pattern matching. Tech. Rep. LARA-REPORT-2008-004, EPFL (2008)
10. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV. LNCS, vol. 4144 (2006)
11. Ferrara, P.: Static type analysis of pattern matching by abstract interpretation. In: Formal Techniques for Distributed Systems. pp. 186–200. Springer (2010)
12. Franzen, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. In: FMCAD (2010)
13. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proc. ACM PLDI (1991)
14. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: RTA. pp. 210–220 (2004)
15. Gries, D.: The Science of Programming. Springer (1981)
16. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. In: Theorem Proving in Higher Order Logics: Emerging Trends Proceedings (2007)
17. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: Verifying functional programs using abstract interpreters. In: Computer Aided Verification (CAV) (2011)
18. Kaufmann, M., Manolios, P., Moore, J.S. (eds.): Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
19. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: POPL (2010)
20. Manolios, P., Turon, A.: All-termination(T). In: TACAS. pp. 398–412 (2009)
21. Moore, J.S.: Theorem proving for verification - the early days. In: Keynote talk at FLoC. Edinburgh (July 2010)
22. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
23. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer-Verlag (2002)
24. Odersky, M.: Contracts in Scala. In: International Conference on Runtime Verification. Springer LNCS (2010)
25. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: a comprehensive step-by-step guide. Artima Press (2008)
26. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
27. Sinha, N.: Modular bug detection with inertial refinement. In: FMCAD (2010)
28. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL (2010)
29. Taghdiri, M.: Inferring specifications to detect errors in code. In: ASE'04 (2004)
30. VSComp: The Verified Software Competition: `http://www.macs.hw.ac.uk/vstte10/Competition.html` (2010)
31. Walther, C., Schweitzer, S.: About VeriFun. In: CADE (2003)
32. Zee, K., Kuncak, V., Taylor, M., Rinard, M.: Runtime checking for program verification. In: Workshop on Runtime Verification. LNCS, vol. 4839 (2007)