# On Satisfiability Modulo Computable Functions
## EPFL-REPORT-161285

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 formulas in the language of algebraic data types and integer linear arithmetic extended with user-defined terminating recursive functions. Our procedure is designed to integrate into a DPLL($T$) solver loop, using blocking clauses to control function definition unfolding. The procedure can check the faithfulness of candidate counterexamples using code execution. It is sound for proofs and counterexamples. Moreover, it is terminating and thus complete for many important classes of specifications: for satisfiable specifications, for specifications whose recursive functions are sufficiently surjective, and for functions annotated with inductive postconditions. We have implemented our system in Scala, building on top of the Z3 API and Z3's plugin mechanism. Our results show our approach to be superior in practice to the alternative of encoding recursive functions as quantified axioms. Using our system, we verified detailed correctness properties for functional data structure implementations, as well as Scala syntax tree manipulations. We have found our system to be fast for both finding counterexamples and finding proofs for inductively annotated specifications. Furthermore, it can quickly enumerate many test cases satisfying a given functional precondition, which can then be used to test both functional and imperative code. Thanks to our tool, many SMT solver clients, including verifiers and synthesizers, can benefit from the expressive power of recursive function definitions within formulas.

## 1 Introduction and Background

SMT solving tools [20, 4, 8] are important drivers of advances in verification of large software and hardware systems [3, 10]. SMT solvers are efficient for deciding quantifier-free formulas in the language of useful theories, such as linear integer arithmetic and algebraic data types. Nonetheless, the operations available in the existing theories are limited, which prevents verification against more detailed specifications and the discovery of more complex invariants needed to prove safety of systems. To increase the power of SMT-based reasoning, we extend the expressive power of formulas and allow them to contain user-defined

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

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 in many cases.

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 [17, 19]. Recent systems based on functional programs include VeriFun [26] and AProVE [12]. Moreover, computable specifications form important parts of many case studies in proof assistants Coq [5] and Isabelle [21]. 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 [15]. When it comes to reasoning within these systems, the systems offer varying degree 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. Analogously, the HMC verifier [16] and DSolve [24] can automatically discover inductive invariants. Our system does not attempt to do so and instead focuses on counterexamples and simple cases of inductive proofs. Counterexample generation has been introduced into the Isabelle system through tools such as Nitpick [6], but such techniques fail to explore the efficiency of theory solvers over domains such as integers, finite sets, and algebraic data types.

**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 to determine whether universally quantified axioms hold for all of the infinitely many values of the domain. In our case, terminating executable functions are a particularly important and well-behaved class of quantified axioms, so we can check the consistency of a candidate assignment by executing them. Moreover, the unfolding of function definitions gives us an analogue of a quantifier instantiation strategy along with completeness guarantees.

A high degree of automation in our system comes in part from using state-of-the-art SMT solver Z3 [20] to reason about quantifier-free formula modulo theories such as algebraic data types, and linear arithmetic, as well as to perform case splitting along with automated lemma learning. Our algorithm integrates into the DPLL(T) loop of Z3, both through a top-level wrapper and a call-back supported in the Z3 theory plugin.

The algorithm performs demand-driven unfolding of function definitions and checks the validity of candidate counterexamples using both the forms of the formulas and code execution. Our algorithm thus both finds proofs and generates counterexamples in the forms of test cases. We have designed our algorithm to be complete for counterexamples and for assume-guarantee reasoning, but we have subsequently discovered that it is complete also for useful previously identified decidable fragments [25]; this completeness has been confirmed by the behavior of our system in practice.

**Experience with the resulting system.** Our system acts as an extension of the Z3 prover with recursive functions, and can therefore be used for verification of functional and imperative programs, synthesis, test generation, and related tasks. We have evaluated it on the verification of functional programs, as well as test generation.

Our verifier for functional programs allows the developer to state the functions in a subset of Scala and compile them using the standard Scala compiler. This means that the developer can manually run tests on sample inputs. The developer uses the existing library for dynamically checking executable contracts [22] to describe the desired properties. As a result, run-time contract violations of contracts can be found using testing.

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.

The system generates verification conditions expressing the validity of functions against their contracts. It also generates implicit checks that ensure that preconditions are met at all function invocations. Finally, it generates conditions that ensure completeness of pattern matching expressions. Finally, it proves these verification conditions using our algorithm for integrating function unfolding and model validation. We have applied our system to a number of examples including data structures such as red-black trees, a library of list manipulating functions, as well as expression manipulation functions. We have found that the system was fast both in finding counterexamples and proofs for verification conditions.

We also examined the performance of our system for generating test cases. We have found that using a simple model enumeration technique we were able to generate large number of test cases that satisfy specifications involving recursive functions.

We therefore believe that the algorithm holds great promise for practical verification of complex properties of computer systems.

**Contributions.** We summarize the contributions of our paper 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.
- We observe that our satisfiability procedure is:
  1. sound for proofs: if it reports 'unsat', then there are no assignments to free variables for which the formula is true;
  2. sound for models: every model it returns in the 'sat' case is a true model of the formula;
  3. terminating for all formulas that are satisfiable;
  4. terminating in case of unsatisfiable formulas arising from functions annotated with inductive postconditions (as in assume-guarantee reasoning);
  5. terminating for the important family of all sufficiently surjective abstractions [25].
- 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 through a combination of a top-level loop and a callback function.
- We present our results in verifying over 40 functions manipulating integers, sets, and algebraic data types. The system was able to verify correctness properties expressed as postconditions as well as the completeness of pattern matching expressions. The system is also able to generate test cases based on specifications involving recursive functions.

## 2 Examples

We now illustrate how our verification system based on our procedure for satisfiability modulo computable functions, which we call Leon, can be used to prove interesting properties about functional programs. Consider the following recursive datatype, written in Scala syntax [23], 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 Literal(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))
  case Not(f) ⇒ Not(simplify(f))
  case Literal(_) ⇒ f
}) ensuring(isSimplified(_))
```

We could prove that simplified formulas never contain implications by using for instance the approach of set constraints [1]. Alternatively, using our technique we can simply write a recursive function isSimplified that *computes* 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 [22], 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 Literal(_) ⇒ true
}
```

(Note that we would also typically write such an executable specification function for testing purposes.) Using our procedure for satisfiability modulo computable functions, our system can prove that the postcondition of simplify is satisfied for any input formula f.

Such refinements of types are known as refinement types [11]. Refinement types that are defined using functions such as isSimplified are in fact sufficiently surjective abstractions [25], which implies that our system is a decision procedure for such constraints (see Section 4). This is confirmed with our experiments—our tool is predictably fast on such examples.

Suppose now that we want 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 ensuring(res ⇒ res).) Our verification system proves this property instantly, by unrolling the definitions and postconditions of simplify and isSimplified sufficiently many times.

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 nontrivial 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 Literal(i) ⇒ Set[Int](i)
  }
}
```

(++ 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 ad-hoc analyses for checking such exhaustiveness properties [7, 9]. 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, a requirement to perform 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 And(lhs, rhs) ⇒ And(nnf(lhs), nnf(rhs))
  case Or(lhs, rhs) ⇒ Or(nnf(lhs), nnf(rhs))
  case Implies(lhs, rhs) ⇒ Implies(nnf(lhs), nnf(rhs))
  case Not(And(lhs, rhs)) ⇒ Or(nnf(Not(lhs)), nnf(Not(rhs)))
  case Not(Or(lhs, rhs)) ⇒ And(nnf(Not(lhs)), nnf(Not(rhs)))
  case Not(Implies(lhs, rhs)) ⇒ And(nnf(lhs), nnf(Not(rhs)))
  case Not(Not(f)) ⇒ nnf(f)
  case Not(Literal(_)) ⇒ formula
  case Literal(_) ⇒ formula
}
```

From the supposed roles of the functions simplify and nnf, one would imagine 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(Literal(48),
Literal(47))), And(Literal(46), Literal(45)))
```

A consequence of our algorithm is that Leon never reports false positives (see Section 4). In this particular example, the counter-example 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 discovered rapidly.

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, defined as follows:

```
sealed abstract class Tree
case class Empty() extends Tree
case class Node(color: Color, left: Tree, value: Int, right: Tree) extends Tree
sealed abstract class Color
case class Red() extends Color
case class Black() extends Color
```

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 anyway want to write for testing purposes. The specification of insertion and of tree balancing 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))

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

Leon proves these properties of red-black tree operations. Similarly, one can write an abstraction function that computes whether a tree satisfies a given invariant (such as being balanced, or having the right coloring property), and prove that the invariants are maintained by all operations. More such results are described in Section 6.

## 3 Our Satisfiability Procedure

In this section, we describe our algorithm for checking the satisfiability of formulas modulo computable functions. At a high-level, it can be described as an abstraction-refinement approach; initially, function symbols are treated as entirely uninterpreted, and their interpretation is introduced gradually, for deeper and deeper unrollings. In other words, the formula can be viewed as a partial but growing program that computes a truth value. We use boolean variables to represent the branching conditions, and at any point in the execution of our algorithm, we ensure that paths that lead to a function call that has not yet been unrolled are blocked. As a result, when the SMT solver reports a satisfying assignment, we know it to be valid, because it can correspond to an execution of the formula that returned false using only known parts of the program; in other words, the SMT solver had all the information it needed to evaluate the counter-example using the proper semantics for all required functions.

Figure 1 describes our approach in pseudo-code. In this figure, $\phi$ is the formula for which we want to determine satisfiability, $\Pi$ is the program containing the function definitions, $\mathsf{impl}_f^{\Pi}$ is the expression corresponding to the body of $f$ in $\Pi$, $\mathsf{post}_f^{\Pi}$ is the (boolean) expression corresponding to its precondition (or $\mathsf{true}$ if not defined), and similarly for $\mathsf{prec}_f^{\Pi}$. The free variables of $\mathsf{impl}_f^{\Pi}$ are denoted by $\mathsf{args}_f^{\Pi}$. The free variables of $\mathsf{post}_f^{\Pi}$ are $\mathsf{args}_f^{\Pi}$ plus a special variable $\rho$ that denotes the value returned by the function.

We can make the following observations:

- The prioritization of pairs added to the queue *Queue* is not specified in our description, and neither is the bound on the loop in the function DEQUEUE. Indeed, the specifics can vary, as long as the priority queue is guaranteed to be *fair*, that is that every pair is popped eventually. A valid strategy is for instance to always empty the queue in DEQUEUE.
- While the description of SOLVE suggests that we need to query the solver twice in a given loop iteration, we can in practice use the solver's capability to output *unsat cores* to detect with a single query whether the blocking literals $\bigwedge_{b \in B} b$ played any role in the unsatisfiability.
- Similarly, when adding the clauses obtained from DEQUEUE, we can use the solver's incremental reasoning and push the new constraints directly, rather than building a new formula and requerying it.

```
$Queue \leftarrow empty$
**function** SOLVE($\phi$,$\Pi$)
    $(initialClauses, B) \leftarrow$ CLAUSIFY($\phi, \Pi$)
    $\phi \leftarrow \bigwedge initialClauses$
    **while** true **do**
        $result_1 \leftarrow Solver.solve(\phi \wedge \bigwedge_{b \in B} \neg b)$
        **if** $result_1 =$ sat **then**
            **return** sat
        **else**
            $result_2 \leftarrow Solver.solve(\phi)$
            **if** $result_2 =$ unsat **then**
                **return** unsat
            **else**
                $(newClauses, newB) =$ DEQUEUE($\Pi$)
                $\phi \leftarrow \phi \wedge \bigwedge newClauses$
                $B \leftarrow newB$
            **end if**
        **end if**
    **end while**
**end function**
**function** DEQUEUE($\Pi$)
    $B \leftarrow \emptyset$
    $clauses \leftarrow \emptyset$
    **while** ... **do**
        $(b, F) \leftarrow Queue.pop$
        **for** $f(\overline{args}) \leftarrow F$ **do**
            $\psi_1 \leftarrow (f\overline{args}) = \mathsf{impl}_f^{\Pi} \left[\overline{args}/\mathsf{args}_f^{\Pi}\right])$
            $\psi_2 \leftarrow (\mathsf{post}_f^{\Pi} \left[\overline{args}/\mathsf{args}_f^{\Pi}\right] [f(\overline{args})/\rho])$
            $(clauses_1, B_1) \leftarrow$ CLAUSIFY($\psi_1, \Pi$)
            $(clauses_2, B_2) \leftarrow$ CLAUSIFY($\psi_2, \Pi$)
            $clauses \leftarrow clauses \cup clauses_1 \cup clauses_2$
            $B \leftarrow B \cup B_1 \cup B_2$
        **end for**
    **end while**
    **return** $(clauses, B)$
**end function**
**function** CLAUSIFY($\psi, \Pi$)
    Returns a pair $(clauses, B)$ where $\bigwedge clauses$ is equivalent to $\psi$. As a side-effect,
pushes into $Queue$ all pairs $(b, F)$, where $F$ is the non-empty set of recursive function
application terms appearing in $clauses$ and that are guarded by $b$.
**end function**
```

**Fig. 1.** Pseudo-code of our Procedure for Satisfiability Modulo Computable Functions

- It can happen in any iteration that the solver concludes that, for a literal $b \in B$, $\phi \implies b$. In that case, we can safely remove the pair $(b, F)$ from the queue (if it was present), because we know for sure that the path represented by $b$ will never be used in a satisfying assignment. While this does not necessarily influence the solver's search space, it can speed up the unrolling process, as some function invocations can be ignored.

## 4 Properties of Our Procedure

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

**Termination:** Each functions in the program $\Pi$ terminates on each input. Tools such as [13, 2] could be used to establish this property.

**SMT Solver Soundness:** The underlying SMT solver is complete and sound for the quantifier-free formulas that it receives. 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 SMT solver reports unsatisfiability, $\mathsf{false}$ can be logically concluded modulo the solver's theories from these constraints.

We use the above assumptions throughout this section.

**Soundness for Proofs.** Our algorithm reports unsatisfiability if and only if the underlying SMT solver could prove the problem given to it unsatisfiable *without* the blocking literals. Because the blocking 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 SMT solver only produces sound proofs, it suffices to show that all the axioms *body* and *post* communicated to the solver in our procedure are obtained from sound derivations.

The *body* axioms 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 $\Pi$ have been proved:[1]

1. for each function $f$ of $\Pi$, the following formula must hold:

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

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

2. for each call in $f$ to a function $f_2$ (possibly $f$ itself), the precondition $\mathsf{prec}_f^{\Pi}$ 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 [14, 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 $\phi$ and $\Pi$.

**Soundness for Models.** Our algorithm reports satisfiability when the solver reports that the unrolled problem augmented with the blocking literals is satisfiable. By construction of the set of blocking 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 SMT solver for the problem augmented with the blocking 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. To understand why, 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 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. By our Termination assumption, this computation tree is finite. Consequently, the tree contains finitely many unfoldings 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 pair that is pushed into the queue is popped eventually, 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 SMT solver and the consistency of a satisfying specification, it means that the SMT solver reports a counterexample (either $\boldsymbol{a}$ itself or another counterexample).

**Termination for Valid Inductively Annotated Programs.** When arguing soundness for proofs, we have observed that our algorithm subsumes assume-guarantee reasoning for functional programs in which functions are annotated with contracts. For the same reason, if assume-guarantee reasoning succeeds to prove a formula, then our algorithm succeeds as well. Note that our algorithm supports arbitrary expressions relating any number of functions. Moreover, like $k$-induction, it succeeds also in the cases where the annotations of functions are not inductive, but become inductive after some number of function unfoldings.

**Termination for Sufficiently Surjective Abstraction.** Our procedure always terminates and is therefore a decision procedure in the case of a recursive function that is sufficiently surjective [25]. We have already established termination in the case of 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 [25]. The unrolling depth depends on the particular sufficiently surjective abstraction, which is why [25] 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. It is also a first implementation that we know that is simultaneously complete for all these decidable problems.

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 a 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 number of functions into a finite domain. Among the functions that map into a finite domain are predicates expressing refinement types [11], which explains why Leon was successful in verifying complex pattern-matching exhaustiveness constraints.

## 5 Our 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 (vals and no vars in Scala terminology).

**Underspecified functions.** Leon also supports functions whose implementation is not expressed in the strict subset; in such cases, it emits a warning that the results are provided under the assumption that the implementation of the unknown functions is correct. This can be useful for instance to specify only the pre- or postcondition of a function without its implementation.

**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. When we successfully prove that all match expressions are exhaustive, we effectively rule out the possibility that this value affects the semantics of the expression. Note that we only generate verification conditions for pattern-matching expressions that are not obviously complete; if the match contains a catch-all, or if it is clear from the syntax that all alternatives of the matched data typed are covered, we do not need to use an error value.

**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 [20] 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 treatment 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 is not a problem in practice though, as we are guaranteed that all programs that go through the front-end are well-typed, and thus all selectors are applied only to valid terms.

**Test case enumeration.** Leon can also be used to generate test cases. This feature is particularly useful for functions that are not implemented in the core functional subset of Scala, but are annotated. The generation works by finding a series of satisfying assignments for the precondition of the function we wish to test, using our procedure for satisfiability modulo computable functions. We have found that even for relatively complex preconditions (such as for instance the constraint that an input tree should be a binary search tree) we could generate hundreds of assignments within seconds. We believe this technique could be successfully integrated into tools such as QuickCheck, which may experience difficulties to satisfy a complex precondition by randomly generating candidate test cases.

## 6 Experimental Evaluation

We evaluated our verification system by 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 2, where 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, 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 Pentium 4 processors running at 3 GHz and 2 GB of RAM.[2]

The ListOperations benchmark contains various common operations on lists, and we prove for instance that a tail-recursive version of size is functionally equivalent to a simpler version, that append is associative, or that content, which computes the set of elements in a list, distributes over append. For associative lists, we first prove 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. We then prove the *read-over-write* property, which states that looking up the value associated with a key gives the most recently updated value. In our system, we express this 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 this property automatically, sufficiently unrolling all three invocations of the recursive functions. We prove standard properties of insertion sort such as the fact that the output of the function sort is sorted, and has the same size and content as the input list. The function buggySortedIns is similar to sortedIns, which 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

[2] All the benchmarks are available from `http://lara.epfl.ch/~psuter/cav2011/`.

| Benchmark *(LOC)* | #p. | #m. | V/I | Time | function | #p. | #m. | V/I | Time |
|---|---|---|---|---|---|---|---|---|---|
| ListOperations *(122)* | | | | | | | | | |
| size | 0 | 1 | V | 0.39 | sizeTailRecAcc | 1 | 1 | V | 0.05 |
| sizesAreEquiv | 0 | 0 | V | 0.01 | sizeAndContent | 0 | 0 | V | 0.01 |
| reverse | 0 | 0 | V | 0.07 | reverse0 | 0 | 1 | V | 0.04 |
| append | 0 | 1 | V | 0.05 | nilAppend | 0 | 0 | V | 0.05 |
| appendAssoc | 0 | 0 | V | 0.11 | sizeAppend | 0 | 0 | V | 0.05 |
| concat | 0 | 0 | V | 0.04 | concat0 | 0 | 2 | V | 0.25 |
| zip | 1 | 2 | V | 0.15 | sizeTailRec | 1 | 0 | V | 0.02 |
| content | 0 | 1 | V | < 0.01 | | | | | |
| AssociativeList *(60)* | | | | | | | | | |
| update | 0 | 1 | V | 0.20 | updateElem | 0 | 2 | V | 0.06 |
| readOverWrite | 0 | 1 | V | 0.30 | domain | 0 | 1 | V | 0.35 |
| find | 0 | 1 | V | < 0.01 | | | | | |
| InsertionSort *(86)* | | | | | | | | | |
| size | 0 | 1 | V | 0.31 | sortedIns | 1 | 1 | V | 0.27 |
| buggySortedIns | 1 | 1 | I | 0.15 | sort | 1 | 1 | V | 0.13 |
| contents | 0 | 1 | V | < 0.01 | isSorted | 0 | 1 | V | < 0.01 |
| RedBlackTree *(112)* | | | | | | | | | |
| ins | 4 | 1 | V | 2.02 | makeBlack | 0 | 0 | V | 0.02 |
| add | 2 | 0 | V | 0.07 | buggyAdd | 1 | 0 | I | 0.32 |
| balance | 0 | 1 | V | 0.15 | buggyBalance | 0 | 1 | I | 0.09 |
| content | 0 | 1 | V | < 0.01 | size | 0 | 1 | V | < 0.01 |
| redNHaveBlckC. | 0 | 1 | V | 0.31 | redDHaveBlckC. | 0 | 1 | V | < 0.01 |
| PropositionalLogic *(86)* | | | | | | | | | |
| simplify | 0 | 1 | V | 0.77 | nnf | 0 | 1 | V | 1.49 |
| wrongCommutative | 5 | 1 | I | 0.70 | simplifyBreaksNNF | 0 | 0 | I | 0.49 |
| nnfIsStable | 0 | 0 | V | 0.38 | simplifyIsStable | 0 | 0 | V | 0.20 |
| isSimplified | 0 | 1 | V | < 0.01 | isNNF | 0 | 1 | V | 0.01 |
| vars | 6 | 1 | V | 0.14 | | | | | |

**Fig. 2.** Summary of evaluation results

RedBlackTrees benchmark, we proved that the tree implements a set interface and that balancing, additionnally to producing a balanced tree, preserves the "red nodes have no black children" property and the contents of the tree. We also introduced two bugs (forgetting to paint a node black and missing a case in balancing) and Leon found a concise counterexample in both cases. Finally, the PropositionalLogic benchmark contains functions manipulating abstract syntax trees of boolean formulas. We proved for instance that applying a negation normal form transformation twice is equivalent to applying it once, and showed that our nnf and simplify operations don't commute (because our simplification function introduces a negation to eliminate implications).

## References

1. Aiken, A.: Introduction to set constraint-based program analysis. Science of Computer Programming 35, 79–111 (1999)
2. 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)
3. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: CAV (2010)
4. Barrett, C., Tinelli, C.: CVC3. In: CAV. LNCS, vol. 4590 (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. Dotta, M., Suter, P., Kuncak, V.: On static analysis for expressive pattern matching. Tech. Rep. LARA-REPORT-2008-004, EPFL (2008)
8. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV. LNCS, vol. 4144 (2006)
9. Ferrara, P.: Static type analysis of pattern matching by abstract interpretation. In: Formal Techniques for Distributed Systems. pp. 186–200. Springer (2010)
10. Franzen, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. In: FMCAD (2010)
11. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proc. ACM PLDI (1991)
12. Giesl, J., Schneider-Kamp, P., Thiemann, R.: Automatic termination proofs in the dependency pair framework. In: IJCAR (2006)
13. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: RTA. pp. 210–220 (2004)
14. Gries, D.: The Science of Programming. Springer (1981)
15. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. In: Theorem Proving in Higher Order Logics: Emerging Trends Proceedings (2007)
16. Jhala, R., Majumdar, R., Rybalchenko, A.: Refinement type inference via abstract interpretation. CoRR abs/1004.2884 (2010)
17. Kaufmann, M., Manolios, P., Moore, J.S. (eds.): Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
18. Kaufmann, M., Manolios, P., Moore, J.S. (eds.): Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
19. Moore, J.S.: Theorem proving for verification - the early days. In: Keynote talk at FLoC. Edinburgh (July 2010)
20. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer-Verlag (2002)
22. Odersky, M.: Contracts in Scala. In: International Conference on Runtime Verification. Springer LNCS (2010)
23. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: a comprehensive step-by-step guide. Artima Press (2008)
24. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
25. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL (2010)
26. Walther, C., Schweitzer, S.: About VeriFun. In: CADE (2003)