Abstract

We present a verification procedure for pure higher-order functional Scala programs with parametric types. We show that our procedure is sound for proofs, as well as sound and complete for counter-examples. The procedure reduces the analysis of higher-order programs to checking satisfiability of a sequence of quantifier-free formulas over theories such as algebraic data types, integer linear arithmetic, and uninterpreted function symbols, thus enabling the use of efficient satisfiability modulo theory (SMT) solvers.

Our solution supports arbitrary function types and arbitrarily nested anonymous functions (which can be stored in data structures, passed as arguments, returned, and applied). Among the contributions of this work is supporting even those cases when anonymous functions cannot be statically traced back to their definition, ensuring completeness of the approach for finding counter-examples. We provide a proof of soundness and counter-example completeness for our system as well as initial evaluation in the Leon verifier.

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

Keywords software verification; higher-order functions; satisfiability modulo theories

1. Introduction

Functional languages are well suited for verification due to their clear semantics [6]. Recent work [3, 19] has shown that recursive programs over unbounded data types can be precisely handled using unfolding-based approaches. However, one of the main features of functional languages, namely higher-order functions, is still difficult to support in modern automated program verifiers. A common approach is to focus on sound approaches while sacrificing completeness for counter-examples [4, 14, 20] or focus on finite domains [7, 9]. While universal quantification offers a natural encoding of first-order functions, encoding closures typically requires universally quantifying over SMT arrays, a feature for which modern SMT solvers offer limited support and few guarantees.

Our approach extends existing work on solving constraints from first-order recursive programs that relies on unfolding function definitions [19]. Supporting closures blurs the boundary between programs and data, complicating the reduction of functional programs to tractable verification conditions. For instance, representing the application of a closure may need to take into account closures that have potentially not been discovered yet.

Our solution adds support for higher-order constructs by encoding them in a sequence of first order quantifier-free formulas that are efficiently supported by the underlying SMT solvers. We introduce a form of controlled dynamic dispatch for closure applications. However, since not all viable targets may have yet been discovered at the time of encoding a particular closure application, this dynamic dispatch needs to expand as unfoldings discover new compatible definitions. This encoding supports even those cases when anonymous functions cannot be statically traced back to their definition: function values can be passed arbitrarily through parameters, used to construct new function values, and stored inside unbounded data structures.

In the presence of terminating programs, our technique is sound both when it reports that the program is correct, and when it reports a counter-example. Moreover, it is complete (guaranteed to terminate) when there exists a counter-example, which is a non-obvious feature for a system that
verifies higher-order functions. We find this aspect of our system very important because most of the time when developing a verified program is spent correcting errors in code or specification.

**Contributions.** We make the following contributions:

- We present a procedure for verifying higher-order functional programs with decidable theories including algebraic data types and integer linear arithmetic. Our procedure uses a new encoding of first-class functions, with expressive and precise representation of functions stored inside data structures.
- Our procedure is sound for proofs and counter-examples, and complete for the later. We provide a detailed proof of counter-example completeness.
- We present the implementation of the procedure within the Leon verifier (http://leon.epfl.ch) as well as its evaluation on a number of Scala programs that make use of higher-order functions. Our results show that, in most cases, the verification remains tractable in the presence of higher-order functions.

### 2. Examples of Verification with Higher-Order Functions

We illustrate the capabilities of the Leon verification system for finding errors and proving correctness of programs with higher-order functions. Our input language is a purely functional subset of the Scala programming language with recursive algebraic data types. We rely on the first phases of the Scala compiler to consistently resolve symbols, types, and implicits.

**Expression transformations.** Our first example in Figure 1 defines simple arithmetic expressions and manipulates them using three higher-order functions: a generic transformation function, a function checking the existence of a sub-expression, and a simplification function. The post-condition of a function is given using the infix `ensuring` operator by constraining the result value as described in [11]. Here, we ensure that the result of `simplifyEquals` no longer contains equality checks with additions of literals.

Leon checks for correctness by building a constraint corresponding to the presence of a counter-example, that is, a constraint checking for the existence of a valid input to `simplifyEquals` such that its result violates the post-condition. Since these constraints generally contain both function calls and higher-order constructs, we encode them in a sequence of quantifier-free formulas in which we progressively unfold the bodies of functions and closures. Based on the result of the solver checks, the procedure determines validity of the specified property or outputs a counter-example. In our example, Leon finds the following counter-example:

```
expr => Equals(Add(Literal(0), Literal(0)), Add(Literal(0), Literal(0)))
```

![Figure 1. Expression tree transformation](image)

This concrete counter-example allows the developer to understand the error: `simplifyEquals` does not handle the case where both operands of `Equals` are additions of literals. We can correct this error by folding additions of literals, adding
case Add(Literal(i), Literal(j)) => Some(Literal(i + j))
to the cases of `simplifyEquals`. This new version is proved correct by Leon (for all of the infinitely many expression trees) in less than a second.

**Generic sorting.** We consider in Figure 2 the problem of sorting a generic list with a parametric ordering. We define an ordering on elements by a closure that maps each element to \( \mathbb{Z} \), ensuring a well-founded ordering. This definition enables us to verify the sorting algorithm modularly, independently of the concrete list type or the ordering.

We check that our version of merge sort keeps the same content, expressed as a set of elements, and that the resulting list is indeed sorted. Leon successfully verifies our implementation in under a second.

### 3. Verifying Higher-Order Programs

To set up the context of our contribution, we start by presenting the existing technique for verifying first-order recursive functions in Leon, then build on it to present techniques for higher-order functions.
case class Ordering[T](f: T ⇒ BigInt)

def isSorted[T](list: List[T])(implicit o: Ordering[T]): Boolean =
list ... for arbitrary recursive functions. In
addition, [17, 18] proves termination of verification for a
class of functions.

3.1 Verifying Recursive First-Order Programs

defined portions. When over-approximating, the complete
branches leading to function calls that have not been defined
b... The boolean variables
constraint as well as two unfoldings of the recursive
size
in Figure 3 and its verification condition, negated:

<table>
<thead>
<tr>
<th>Figure 2. Generic sorting function</th>
</tr>
</thead>
<tbody>
<tr>
<td>def merge[T](l1: List[T], l2: List[T])</td>
</tr>
<tr>
<td>(implicit o: Ordering[T]): List[T] =</td>
</tr>
<tr>
<td>(l1, l2) match {</td>
</tr>
<tr>
<td>case (Cons(h1, Cons(h2, xs))) ⇒</td>
</tr>
<tr>
<td>if (o.f(h1)) Cons(h1, merge(t1, l2))</td>
</tr>
<tr>
<td>else Cons(h2, merge(l1, t2))</td>
</tr>
<tr>
<td>case Nil ⇒ (l1, l2)</td>
</tr>
<tr>
<td>} ensuring { res ⇒</td>
</tr>
<tr>
<td>contents(res._1) ++ contents(res._2) = contents(list) }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figure 3. Duplication of the head of a list</th>
</tr>
</thead>
<tbody>
<tr>
<td>def size[T](l: List[T]): BigInt = (l match {</td>
</tr>
<tr>
<td>case Nil ⇒ 0</td>
</tr>
<tr>
<td>case Cons(h, t) ⇒ 1 + size(t)</td>
</tr>
<tr>
<td>}) ensuring (r ≥ 0)</td>
</tr>
</tbody>
</table>

Our procedure for first-order programs alternates be-
tween model construction (i.e. counter-example discovery) and proofs, by building a sequence of under- and over-
approximations of our verification constraints. These approx-
imations are represented by a decision tree where branching
expressions are instrumented to allow control over which
branches to avoid.

We illustrate this process using the function dup defined in
Figure 3 and its verification condition, negated:

\[ l = Cons(h, t) \land r = Cons(h, l) \land size(r) \leq 1 \]

Figure 4 shows the decision tree corresponding to this initial
constraint as well as two unfoldings of the recursive size
function. The boolean variables \( b_0, b_1, \ldots \) serve as controls
to explicitly exclude program branches from the search.

When under-approximating the constraint, we avoid all
branches leading to function calls that have not been defined
yet. This ensures that potential models only rely on well-
deﬁned portions. When over-approximating, the complete
tree is used. Since function symbols are uninterpreted, calls
that have not been explicitly constrained are treated as return-
ing arbitrary values, which is a sound over-approximation in
our purely functional language.

If results are inconclusive with a given deduction tree
(that is, the under-approximation is Unsat and the over-
approximation is Sat), we increase the precision of the
over-approximations as well as the coverage of the under-
approximation by unfolding function calls left undefined. The
unfolding replaces function application with function body, and
also assumes that the postcondition of the function holds
(enabling reasoning by -induction on function execution).
Any fair unfolding strategy gives same high-level guarantees;
we currently use a breadth unfolding first-search strategy,
which unfolds each function call occurrence. Our encoding
enables us to perform unfolding by “pushing” new constraints,
making use of the incremental solving capabilities of modern
SMT solvers.

In our example, the first under-approximation \( F_0 \land \neg b_0 \)
is trivially Unsat and the over-approximation \( F_0 \) is Sat. We thus unfold the call size(r) by pushing new constraints
according to the instrumented definition of size(r), and obtain \( F_1 \) equal to:

\[
F_0 \land (b_1 \lor b_2) \\
\land ((b_1 \land r = Cons(h_1, t_1)) \Rightarrow S_1) \\
\land ((b_2 \land r = Nil) \Rightarrow S_2)
\]

Given that \( S_1 \) contains an unconstrained function call, the
under-approximation avoids it by enforcing \( \neg b_1 \). Since \( F_1 \land 
\neg b_1 \) is Unsat and \( F_1 \) is Sat, we unfold size(t_1) and obtain \( F_2 \).
Here again, \( F_2 \land \neg b_2 \) is Unsat and \( F_2 \) is Sat. After a third
unfolding, the over-approximation \( F_3 \) is Unsat, attesting of
the absence of counter-examples and thus of the validity of
the verification condition. This approach has three interesting
properties: it guarantees that 1) counter-examples found using
the under-approximation are valid, that 2) proofs obtained
with the over-approximation hold for the original program
(assuming functions are terminating) and that 3) by unfolding,
we cover longer executions and thus a larger subset of the
space of all inputs. This ensures that any counter-example
with a finite execution trace will eventually be discovered.
These properties hold for arbitrary recursive functions. In
addition, [17, 18] proves termination of verification for a
class of functions.
\[ b_0 \land l = \text{Cons}(h, t) \land r = \text{Cons}(h, l) \land \text{size}(r) \leq 1 \quad \ldots \quad F_0 \]

**Unfolding 1**

\[ S_1 \quad b_1 \land r = \text{Cons}(h_1, t_1) \quad b_2 \land r = \text{Nil} \]

\[ \text{size}(r) \geq 0 \land \text{size}(r) = 1 + \text{size}(t_1) \quad \text{size}(r) = 0 \quad \ldots \quad S_2 \]

**Unfolding 2**

\[ S_3 \quad b_3 \land t_1 = \text{Cons}(h_2, t_2) \quad b_4 \land t_1 = \text{Nil} \]

\[ \text{size}(t_1) \geq 0 \land \text{size}(t_1) = 1 + \text{size}(t_2) \quad \text{size}(t_1) = 0 \quad \ldots \quad S_4 \]

**Figure 4.** Decision tree for the verification condition of `dup` with two unfoldings and instrumented branching conditions.

### 3.2 Encoding Closure Applications

In contrast to named functions, the code executed by closure calls cannot in general be statically located. Additionally, although anonymous functions are not directly recursive, a program may define an arbitrary number of closures during its execution. It is thus not possible to lift closures as a finite set of named functions. The dynamic nature of closure applications requires a dedicated encoding, for which we need to progressively consider closure definitions discovered as the analysis unfolding tree grows.

We define a closure as a function body together with an environment. Due to the tree-like nature of our unfolding procedure, the environment can be grounded modulo some global free variables in the initial formula, so closed context need not be handled explicitly, much like in substitution-based semantics for lambda calculus. We call \( \Lambda \) the set of all closures and associate to \( \lambda \in \Lambda \) its arguments \( \lambda_{\text{arg},1}, \ldots, \lambda_{\text{arg},n} \) and body \( \lambda_{\text{body}} \). Closures are not supported by SMT solvers, so we use an encoding domain \( U \) with infinite cardinality (\(|\Lambda|\) is infinite) that supports equality.

Given a bijective mapping \( \mathcal{L} : \Lambda \rightarrow U \) from closures to their identifying values and \( \Lambda_t = \{\lambda_1, \ldots, \lambda_m\} \subseteq \Lambda \) the set of all closures encountered so far in the decision tree \( t \), we can perform guarded unfolding for the application of a closure \( f \) by inlining all possible bodies guarded by equality between \( f \) and the current closure. Namely:

\[
\begin{align*}
\text{f}(x^n) &= \begin{cases} 
\lambda_{1\text{body}} [\lambda_{1\text{arg}} \rightarrow x^n] & \text{if } f = \mathcal{L}(\lambda) \\
\vdots & \vdots \\
\lambda_{m\text{body}} [\lambda_{m\text{arg}} \rightarrow x^n] & \text{if } f = \mathcal{L}(\lambda_m) \\
\text{uninterpreted} & \text{otherwise}
\end{cases}
\end{align*}
\]

\( f(x^n) \) is left unconstrained if the closure associated to \( f \) has not yet been defined in \( t \), yet our fair unfolding ensures that any closure definition that the program produces is eventually considered. The use of unique closure identifiers makes our approach flexible and allows arbitrary use of closures in data structures. Much like a precise \( k \)-CFA [15, 16] for unbounded \( k \), this representation encodes exact propagation of closure identifiers up to a currently considered execution depth. Unlike some alternatives, the encoding can be represented in the simply typed language without subtyping, which is used by SMT solvers. It also works well with our handling of generics that instantiates them at function unfolding time.

### 3.3 Blocking Decision Tree Branches

The guarded unfolding as described above preserves soundness of proofs, but not of counter-examples. The uninterpreted else case needs to be explicitly excluded when looking for models of the under-approximation. To ensure validity, we must prune the decision tree like in the first-order case to disallow branches for which the necessary unfoldings have not yet taken place. We define

\[
b_f = \bigvee_{\lambda \in \Lambda_t} f = \mathcal{L}(\lambda)
\]

and enforce \( b_f \) in the under-approximation. Furthermore, the previously stated property that each closure defined by the program is eventually covered in the decision tree provides us with a high-level argument to the completeness of counter-examples of our procedure. We provide a formalized proof of this argument in Section 4.

### 3.4 Optimizations

The unfolding and guarding procedures we described can be quite expensive when \( \Lambda_t \) becomes large. In practice, there are recurrent patterns that can be handled in an optimized manner while maintaining the above procedure as a fallback to guarantee completeness. An immediate optimization is to only consider closures whose types are compatible with the call.

**Definition tracking along simple paths.** Thanks to the lack of operators on function-typed expressions, concrete function-typed arguments are quite often statically known closures. If we consider the function

\[
def \text{apply1}(f : \text{Int} \Rightarrow \text{Int}) : \text{Int} = f(1)
\]

and the invocation \( \text{apply1}(x \Rightarrow x + 2) \), during unfolding \( f \) can be bound to \( x \Rightarrow x + 2 \) which immediately gives us \( f(1) = 1 + 3 \), thus avoiding an expensive guarded unfolding over all possible \( \lambda \in \Lambda_t \). This technique can be extended to track arbitrary (finitely complex) paths from closure application back to its definition and we implemented it for function-typed arguments as well as immediately returned closures.

To simplify this tracking, we perform some equivalence-preserving transformations to the input programs. For example, let us consider the definition

\[
def \text{applyPair}(p : (\text{Int} \Rightarrow \text{Int}, \text{Int})) : \text{Int} = p._1(p._2)
\]
As \( p \) is no function-typed argument of \( \text{applyPair} \), the path tracking rules described above do not apply. However, through a simple program transform of definition and all invocation points (which are statically known), we get

\[
\text{def applyPair}(f: \text{Int} \Rightarrow \text{Int}, p: \text{Int}): \text{Int} = f(x)
\]

and our simple path tracking rules can be instantiated. These techniques give our approach many opportunities to avoid the combinatorial explosion we get in the fallback case, while maintaining the same soundness and completeness properties of the procedure.

**One-time function encoding.** SMT solvers such as Z3 provide library APIs to inject clauses directly into the solver without passing through the SMT-LIB interface. One performance gain of these APIs is that substitution can be performed directly in the solver’s formula domain. In other words, it is possible to pre-translate program elements into the formula domain and substitute variables with other values later on. We make use of this feature by statically determining all invocation and application points in function definitions and storing these in a pre-translated function template. During unfolding, formal arguments are simply substituted with concrete ones in the formula domain and the next required unfoldings are collected based on the previously accumulated call points.

**Closure equality.** In addition to performance concerns, our system also improves the detection of cases when no counter-examples exist. When building inductive proofs, the procedure heavily relies on the hypothesis holding in the inductive case. The potential for inductive hypothesis identification is greatly improved by introducing a notion of closure equality. This is encoded by syntactic checks along with closed world equality constraints. Despite its incompleteness, we have found our check to be quite useful in proofs of inductive properties.

### 4. Completeness and Soundness

We now describe our procedure in a more formal sense and provide a proof of its counter-example soundness and completeness. The completeness for counter-examples then also implies soundness for proofs. We will concentrate here on finding a valid model to arbitrary expressions: if we have a procedure that is guaranteed to find such models when they exist, then we are complete for counter-examples.

#### 4.1 Defining the Domains

We start by defining \( H \) in Figure 5, a purely functional subset of Scala. We call \( H_f \) the set of named functions in \( H \) and for \( f \in H_f \), let \( f_{\text{arg},1}, \ldots, f_{\text{arg},n} \) denote the arguments of \( f \) and \( f_{\text{body}} \) its body. Likewise, we call \( H_c \) the set of closures in \( H \) and for \( \lambda \in H_\lambda \), we define \( \lambda_{\text{arg},i} \) and \( \lambda_{\text{body}} \) by analogy to \( f \in H_f \). To avoid confusion, we will refer hereafter to function invocations when discussing named function calls (i.e. \( f(\overline{x}) \) for \( f \in H_f \)) and function applications when discussing other calls (i.e. \( g(\overline{x}) \) where \( g \) evaluates to \( \lambda \in H_\lambda \)). Note that callers in function applications can never be recursive as they are anonymous.

We define \( H_{\text{var}} \) the set of variables and \( H_{\text{val}} = \{\text{true}, \text{false}\} \cup H_\lambda \) the set of values in \( H \). We also define \( H_{\text{ground}} \) as the set of ground terms in \( H \), namely \( \eta \in H \) such that \( \text{FV}(\eta) = \emptyset \) where \( \text{FV}(\eta \in H) \) is the set of free variables in the program \( \eta \). Finally, we define \( H_{\text{type}} \) the set of types in \( H \), and for a function \( f \in H_f \), let \( f_{\text{arg},1}, \ldots, f_{\text{arg},n} \) denote the types associated to the arguments of \( f \) and \( f_T \) the return type. We also define \( \lambda_{\text{arg},i} \) and \( \lambda_T \) in a similar manner for \( \lambda \in H_\lambda \). We then define the usual typing relation \( H : H_{\text{type}} \) on \( H \) and can therefore define \( H_{c:T} \) the set of variables in \( H \) that type to \( T \) along \( H_{f:T} \) and \( H_{\lambda:T} \) named functions and lambdas typing to \( T \) (note that \( T \) is a function type here).

We further associate a set of evaluation rules to \( H_{\text{ground}} : T \) with call-by-value for functions which give us the evaluation relation \( H_{\text{ground}} \rightarrow H_{\text{val}} \) as defined in Figure 6. Note that for any \( \eta \in H \), such that \( H : T \in H_{\text{type}} \), given a mapping \( m_T \) such that each \( \forall v : T \in \text{FV(}\eta), v \in m_T \land m_{\text{val}[v]} : T \), \( \eta|_{m_H} \in H_{\text{ground}} : T \) is obtained by substitution and \( \eta|_{m_H} \rightarrow g \in H_{\text{val}} \) is well defined.

Our procedure transforms programs into corresponding formulas, so we also give a definition of the logic we work with. Our procedure is orthogonal to built-in theory operations (such as \( + \)), so we use uninterpreted function symbols. Let \( H \) be the theories of boolean terms along with a theory of uninterpreted values. Note that the only operator available for uninterpreted values is equality comparison. We call \( H_{\text{var}} \) the set of variables in \( H \) and \( H_{v:T} \) the set associated to theory \( T \) (for boolean and \( U \) for uninterpreted). We also define \( H_{f:T} \) the set of uninterpreted function symbols associated with theory \( T \) where \( T \) is a tuple of types in \( \{B, U\} \).

We can give a more formal definition of \( L \) introduced in 3.2 as \( L : H_\lambda \leftrightarrow H_{\text{c:T}} \) a bijection between closures and uninterpreted variables in \( H \). We also define a bijection \( V : H_{\text{var}} \leftrightarrow H_{\text{var}} \) between variables of \( H \) and \( H \). Given both these two functions, one can trivially build a correspondence between free variable mapping \( m_H : H_{\text{var}} \rightarrow H_{\text{val}} \) and model \( m_H : H_{\text{val}} \rightarrow \{\text{True, False}\} \cup H_{U:T} \) (note that \( H_{U:T} \) can be considered as values since uninterpreted values do not have fixed interpretation).

Finally, we still require the means to encode functional properties of function calls. Uninterpreted function symbols offer exactly this property, so let us define the class of type-parametric mappings \( F_T : H_{f:T} \rightarrow H_{f:T} \) and a mapping \( \gamma : H_{\text{type}} \rightarrow H_{f:T} \). We use \( F_T \) to encode named function calls and \( \gamma \) to perform dynamic dispatch on closures.

#### 4.2 Defining the Transformation

Given the above domain definitions, we define a transformation \( C \) from a program \( \eta \in H \) to a formula \( c \in H \) such that \( c \) is instrumented in a way that lets us render arbitrary branches of the underlying decision tree inconsequential to overall satisfiability. This instrumentation is performed using control variables that imply all propositions introduced in
\[ \neg \text{false} \rightarrow \text{true} \quad \neg \text{true} \rightarrow \text{false} \quad \text{if}(\text{true}) \rightarrow \text{else} \rightarrow \text{e}_t \quad \text{if}(\text{false}) \rightarrow \text{else} \rightarrow \text{e}_e \]

\[
\begin{align*}
\text{if}(e_c) & \; \rightarrow \; e_t & \quad \text{if}(e'_c) & \; \rightarrow \; e'_t & \quad e_j \in P_{val}, 1 \leq j \leq i-1 & \rightarrow e_i \rightarrow e'_i \\
\text{if}(e'_c) & \; \rightarrow \; e_e & \quad e_j \in P_{val}, 1 \leq j \leq n & \rightarrow f(e_1, ..., e_i, ..., e_n) \\
\end{align*}
\]

\[\lambda \in \mathcal{H}_\lambda \quad e_j \in H_{val}, 1 \leq j \leq i-1 \quad e_i \rightarrow e'_i \]

\[\lambda(e_1, ..., e_i, ..., e_n) \rightarrow \lambda(e_1, ..., e'_i, ..., e_n) \]

\[
\begin{align*}
\text{if}(f_{arg,1} : \text{Boolean}, ..., f_{arg,n} : \text{Boolean}) = \lambda_{body}(\text{f}_{body})^* e & \rightarrow e
\end{align*}
\]

---

**Figure 6.** Evaluation rules for \(H_{ground} \rightarrow H_{val}\)

\[
H \ ::= \langle \text{Definition}\rangle^* \text{Expr} \\
\text{Definition} \ ::= \text{def} \(f(H_{var} : \text{Type}, (H_{var} : \text{Type})^*) : \text{Type} = \text{Expr} \\
\text{Type} \ ::= \text{Boolean} | (\text{Type}, (\text{Type})^*) \Rightarrow \text{Type} \\
\text{Expr} \ ::= H_{var} | H_{val} | \neg \text{Expr} \\
| \text{if}(\text{Expr}) \; \text{Expr} \; \text{else} \; \text{Expr} \\
| (H_{var} : \text{Type}, (H_{var} : \text{Type})^*) \Rightarrow \text{Expr} \\
| \text{Expr}(\text{Expr}, (\text{Expr})^*) \\
| f(\text{Expr}, (\text{Expr})^*)
\]

**Figure 5.** Abstract syntax of \(H\)

A branch and our recursive transformation therefore takes both a program and the current control variable as inputs, so \(C : H \times H_{w:B} \rightarrow R\) for \(R\) described in the following.

In order to later progressively unfold the actual result of function calls, we accumulate invocation and application information during the transformation. Specifically, we need \(t \in T = H_{w:B} \times H \times P_f \times H^*\) for invocations (see case 6 in \(C\)) and both \(p \in \Pi = H_{w:B} \times H \times H \times H^*\) (case 7) and \(\lambda \in \Sigma = H_\lambda\) (case 3). The tuples \(t \in T\) and \(p \in \Pi\) therefore both consist in four parts, namely:

- the instrumentation variable associated to the call,
- an uninterpreted function call that provides a place holder for the concrete call result,
- an identifier for the caller which consists in a static function reference for function invocations and a value in the formula domain for applications,
- a list of arguments (in the formula domain).

The details of the unfolding procedure will be presented in section 4.3. These considerations imply that \(R\) must depend on \(2^2 \times 2^2 \times 2^2\).

Finally, our transformation must naturally return a formula encoding of the input program. In order to perform instrumentation, we separate this output into two parts, the current formula-domain result and a conjunct of implications that represents the decision tree (see conditional encoding case 5 in \(C\)). Note that the former can have any type in the considered theories and the later is boolean. We can now define \(C : H \times H_{w:B} \rightarrow H \times H \times 2^2 \times 2^2 \times 2^2\) such that

\[
\begin{align*}
0. & \quad C((f \in H_1)^* e, b) = C(e, b) \\
1. & \quad C(v \in H_{var}, b) = (\mathcal{V}(v), \emptyset, \emptyset, \emptyset) \\
2. & \quad C(\text{true/false}, b) = (\text{True}/\text{False}, \emptyset, \emptyset, \emptyset, \emptyset) \\
3. & \quad C(\lambda \in H_\lambda, b) = (\mathcal{L}(\lambda), e, \emptyset, \emptyset, \emptyset) \\
(a) & \quad e = \bigwedge_{\lambda_i \in \{\text{previous} \lambda\}} \mathcal{L}(\lambda_i) \\
& \quad \land \bigwedge_{v \in \mathcal{F}(\eta)} \mathcal{L}(\lambda) \neq \mathcal{V}(v) \text{ for } \eta \text{ original program} \\
4. & \quad C(\neg e, b) = (\neg c, e, \tau, \pi, \sigma) \text{ where } (e, c, \tau, \pi, \sigma) = C(e, b) \\
5. & \quad C(\text{if (COND) THEN ELSE b}) = (r, e, \tau, \pi, \sigma) \text{ where } \begin{array}{l}
\text{given } b_t, b_e \in H_{w:B} \text{ fresh variables and } \\
(c, e, \tau, \pi, \sigma)_{c,e} = C(\text{COND, THEN, ELSE}, [b, b_t, b_e]), \text{ let } \\
(a) & \quad r = \mathcal{V}(r_H \in H_{\tau T}) \text{ where } r_H \text{ is a fresh variable and } \\
& \quad \text{THEN : T and ELSE : T} \\
(b) & \quad e = e_c \land e_t \land e_e \\
& \quad \land b \implies (c_e \implies b_t \land \neg c_e \implies b_e) \\
& \quad \land b \implies (b_t \lor b_e) \land \neg (b_t \lor \neg b_e) \\
& \quad \land b_t \implies (r = e_t) \\
& \quad \land b_e \implies (r = e_e) \\
& \quad (c) & \quad \tau = \tau_c \cup \tau_t \cup \tau_e, \pi = \pi_c \cup \pi_t \cup \pi_e \text{ and } \sigma = \sigma_c \cup \sigma_t \cup \sigma_e \\
6. & \quad C(f(\text{ARG}_1, ..., \text{ARG}_n), b) = (v, e, \tau, \pi, \sigma) \text{ where } \\
& \quad \begin{array}{l}
\text{given } (c_1, e_1, \tau_1, \pi_1, \sigma_1) = C(\text{ARG}_1, b) \text{ for } 1 \leq i \leq n, \text{ let } \\
(a) & \quad v = f_{f_{arg} f_{arg}}(f_c(1, ..., c_n)) \\
(b) & \quad \tau = \{f_{arg}, f_c(1, ..., c_n)\} \cup \bigcup_{i=1}^{n} \tau_i \\
(c) & \quad e = \bigwedge_{c_1}^{n} c_1, \pi = \bigcup_{i=1}^{n} \pi_1, \text{ and } \sigma = \bigcup_{i=1}^{n} \sigma_1 \\
7. & \quad C(\text{(ARG}_1, ..., \text{ARG}_n), b) = (v, e, \tau, \pi, \sigma) \text{ where } \\
& \quad \begin{array}{l}
\text{given } (c_1, e_1, \tau_1, \pi_1, \sigma_1) = C(\text{ARG}_1, b) \text{ for } 1 \leq i \leq n, \text{ and } \\
(\text{c}_0, \text{e}_0, \text{c}_0, \text{e}_0, \text{c}_0) = C(\text{b}, \text{b}), \text{ let } \\
(a) & \quad v = \gamma(T)(c_0, c_1, ..., c_n) \text{ where } C : T
\end{array}
\end{array}
\]

23
(b) \( \pi = \{(b, v, c_0, \{c_1, ..., c_n\}) \cup \bigcup_{i=0}^n \pi_i \) \\
(c) \( e = \lambda_{\text{body}}^{n \in \pi} \) \\
We further define the functions \( C_H : H \rightarrow H \times 2^T \times 2^I \times 2^T \) and \( C_L : H \rightarrow H : \text{given } \eta \in H, \text{ let } b_{\text{start}} \) be a fresh variable and compute \((c, e, \tau, \pi, \sigma) = C(\eta, b_{\text{start}})\). Let \( r = c \wedge e \wedge b_{\text{start}} \) in \( C_H(\eta) = (r, \tau, \pi, \sigma) \) and \( C_H(\eta) = r \).

### 4.3 Unfolding Function Calls

The transformation we just described handles function calls by replacing their results with an uninterpreted function result that can take on arbitrary values. In order to bind these uninterpreted function calls to concrete bodies, we consider the definition of function call evaluation to establish the equivalence of evaluation before and after unfolding the body of a function.

Given \( \eta \in H \) and free variable mapping \( m_H \), for \( c_1 = f(\text{ARG}_1, ..., \text{ARG}_n) \subseteq \eta \), let us define \( e_f = f_{\text{body}}[f_{\text{arg}}^{n} \rightarrow \text{ARG}^{n}] \) and \( \eta_f = \eta[c_1 \rightarrow e_f] \). Also, for \( c_2 = C(\text{ARG}_1, ..., \text{ARG}_n) \subseteq \eta \) with \( \eta[m_H] \rightarrow \lambda \), we define \( e_\lambda = \lambda_{\text{body}}[\lambda_{\text{arg}}^{n} \rightarrow \text{ARG}^{n}] \) and \( \eta_\lambda = \eta[c_2 \rightarrow e_\lambda] \). These unfoldings preserve evaluation and give us for \( g \in H \).

\[ \eta[m_H] \rightarrow g \iff \eta_{f,\lambda}[m_H] \rightarrow g. \]

We now want to define unfolding for formulas in \( H \). Given \((c, \tau, \pi, \sigma) = C_H(\eta) \), we define function invocation unfolding for \( t = (b, v, f, c^n) \in \tau \). Let \((c_f, r_f, \tau_f, \pi_f, \sigma_f) = C(\text{fbody}, b)[f_{\text{arg}}^{n} \rightarrow \text{ARG}^{n}] \) and \( I_f(t) = r_f \wedge (b \mapsto v = c_f) \) in \( c_f = c \wedge I_f(t) \) the unfolding of \( t \) in \( c_f \). We know from the definition of \( C \) that \( b \mapsto P(v) \) in \( c_f \) for some proposition \( P \), so \( c_f \) is equivalent to \( c[v \mapsto c_f] \wedge r_f \). Therefore, for any model \( m_H \), we have

\[ m_H \models c_f \iff m_H \models c. \]

For function applications, \( p = (b, v, c_0, c^n) \in \pi \), the situation is slightly more complex. Indeed, the concrete function we would wish to unfold for \( v \) cannot be easily deduced from \( c_0 \). This issue is dealt with by selecting an arbitrary \( \lambda \in \sigma \) and guarding the unfolding with equality between \( c_0 \) and \( \lambda(\lambda) \). Let \( b_p = b \wedge (c_0 = \lambda(\lambda)) \), \((c_p, r_p, \tau_p, \pi_p, \sigma_p) = C(\lambda_{\text{body}}, b_p)[\lambda(\lambda) \rightarrow c^n] \) and \( I_p(p, \lambda) = \tau_p \wedge (b_p \mapsto v = c_0) \) in \( c_p = c \wedge I_p(p, \lambda) \), the unfolding of \( p \) in \( c_f \) conditional on \( c_0 = \lambda(\lambda) \). Note that when we require equality between \( c_0 \) and \( \lambda(\lambda) \), this is modulo a given model \( m_H \), so the full statement would be \( m_H \models c_0 = \lambda(\lambda) \). Our definition of \( C \) guarantees a top-level conjunct in \( c_f \) that states \( \lambda(\lambda) \neq \lambda(\lambda_i) \) for any \( \lambda_i \neq \lambda \) and \( \lambda(\lambda) \neq v \) for \( v \in \text{FV}(\eta) \), so any model \( m_H \models c_0 \) will provide a valid equality check between \( c_0 \) and \( \lambda(\lambda) \). Again, for any model \( m_H \), we have

\[ m_H \models c_p \iff m_H \models c. \]

It is interesting to note that this definition of unfolding function applications extends to any caller variable including \( \nu \) such that \( \nu \in \text{FV}(\eta) \) for \( \eta \) the original program and models for these free functions can be trivially reconstructed given models for the relevant \( \gamma(T) \) and \( \mathcal{V}(\nu) \).

Given the above formula unfolding procedures, we define \( I_f(c, t) = (c_1, \tau_1, \pi_1, \sigma_1) \) and \( I_p(c, p, \lambda) = (c_p, \tau_p, \pi_p, \sigma_p) \).

### 4.4 Interpretation Independence

It is now useful to note a property about the transformation \( C \) that will be used in the following proofs. For \( \eta \in H \) with \( m_H \) such that \( \eta[m_H] \in H_{\text{ground}} \), for each node \( \eta_i \subseteq \eta \) such that \( \eta_i[m_H] \rightarrow \eta_i' \) is inferred during evaluation of \( \eta[m_H] \), then \( \eta_i \) fully determines its associated \( b_i \) from the transformation \( C \). Indeed, this follows trivially from the recursive definitions of evaluation and \( C \) that both visit all nodes in \( \eta \). We say \( b_i \) is the corresponding blocker of \( \eta_i \).

In our definition of \( C \), function invocations and applications are handled by replacing them with a fresh variable in the resulting formula. We call these calls uninterpreted and it is clear that for a formula \( c = C_H(\eta) \) with model \( m_H \models c \) and associated \( m_H \), if \( m_H \) depends on such calls then \( c(m_H) \) may not accurately reflect \( \eta(m_H) \). Indeed, pure function calls are deterministic and can’t take on arbitrary values (given fixed arguments). However, once a call has been unfolded following the previous definitions in 4.3, the model may depend on the associated result value as it is no longer uninterpreted. These considerations lead us to the definition of interpretation-independent models that do not rely on unknown function call results.

**Definition 1.** [interpretation-independence] Given \( \eta \in H \) with \((c, \tau, \pi, \sigma) = C_H(\eta) \) and model \( m_H \models c \), we define \( v_p = \{v \mid (b, v, f, c^n) \in \tau \} \) and \( v_\pi = \{v \mid (b, v, c_0, c^n) \in \pi \} \) as the sets of potentially uninterpreted call results. Let \( TLC(c) \) be the set of top-level conjuncts in \( c \) in

\[ v_t = \{v \mid I_f((b, v, f, c^n) \in \tau) \in TLC(c)\} \]

\[ v_{p,\lambda} = \{v \mid I_p((b, v, f, c^n) \in \pi, \lambda \in \sigma) \in TLC(c) \} \wedge m_H \models c_0 = \lambda(\lambda) \]

We call \( m_H \) interpretation-independent if \( \forall m \neq m_H \) such that \( m[v_i] = m_H[v_i] \) for all \( v_i \in UF(c) - (v_\tau - v_{p,\lambda}) \) where \( UF(c) \) is the set of uninterpreted function calls in \( c \), then \( m \models c \). Note that all elements in \( UF(c) \) correspond to a function call in \( \eta \) as \( C \) only introduces uninterpreted function calls in cases 6 and 7.

The above definition allows us to prove our first theorem, namely that formulas with interpretation-independent models prove to be accurate reflections of programs (i.e. sufficient under-approximations).

**Theorem 2.** For \( \eta \in H \) with \( \eta : T \) for some \( T \in H_{\text{type}} \) and \( m_H \models C_H(\eta) \), if \( m_H \) is interpretation-independent, then corresponding \( m_H \) is such that \( \eta[m_H] \rightarrow \text{true} \).

**Proof.** We will start by defining a helper function \( C_\lambda \) for \( \eta_i \subseteq \eta \) and associated \( b_i \) where \( C_\lambda(\eta_i, b_i) = c \wedge e \) given
Given \( c, \pi, \psi \) and \( q \in \psi \), let \( (c_{i+1}, \pi_q, \psi_q, \sigma_q) = \mathcal{I}_\Lambda(q_i) \) in \( \tau_{i+1} = \pi \cup q, \sigma_{i+1} = \pi \cup \sigma_q \) and let \( \psi_{i+1} = (\psi - \{q\}) \cup Y'(\pi_q, \sigma_q) \cup Y'(\pi, \sigma) \). Note that \( \pi_q \) and \( \sigma_q \) are strictly increasing with respect to set inclusion, and \( \bigcup_{j=0}^i \psi_j \supseteq Y'\left(\pi_q, \sigma_q\right) \). In other words, \( \psi_i \) is the cartesian product of \( \pi_q \) and \( \sigma_q \), minus the \( q_i \) selected at each iteration.

Now observe that for each \( q_i = (b, v, \lambda, c_0, c^n) \), if \( m_{i+1} \models c_i \exists \) such that \( m_i \models \mathcal{L}(\lambda) = c_0 \), then we have \( \mathcal{I}_\Lambda(q_i) \) as a top-level conjunct in \( c_i \) and interpretation-independence with respect to \( v \) is ensured. Let us now define an equivalence relation \( \pi_q \) on \( \Psi \) such that \( q_1, \pi_q(q_2) \) iff \( q_1 \) and \( q_2 \) share a common source in \( \Pi \). Formally, \( (q_1, q_2) \in \pi_q \iff \mathcal{P}_{\{b, v, c_0, c^n\}}(q_1) = \mathcal{P}_{\{b, v, c_0, c^n\}}(q_2) \).

We call \( Q_\pi(Q \in 2^\Psi) = \left\{ [q]_{\pi_q} \mid q \in \psi \right\} \) the set of equivalence classes in \( \psi \) with respect to \( \pi_q \). For \( q_1, q_2 \in Q_\pi(Q) \), all elements share a common \( (b, v, c_0, c^n) \), so we can view \( q_1 \) as \((b, v, \Lambda, c_0, c^n)\) where \( \Lambda = \{ \mathcal{P}_\lambda(q) \mid q \in q_1 \} \). If we look at \( q_{\Lambda}(i) = \{ q_j \mid 0 \leq j \leq i \} \), for \( q_1 = (b, v, \Lambda, c_0, c^n) \) is found on the right-hand side of an implication from \( b \) and \( c_1 \). These observations lead to the following constraint on \( b \) given \( \Lambda \) and \( c_0 \)

\[
B_q(q_{\pi_\psi}) = \neg \left( \bigvee_{\lambda \in \Lambda} c_0 = \mathcal{L}(\lambda) \right) \implies \neg b
\]

Furthermore, we can extend this constraint to all unfoldings as

\[
B Q(i) = \bigwedge_{q \in Q_\pi \left( q_{\Lambda}(i) \right) } B_q(q_{\pi_\psi})
\]

Finally, let \( B_{\Lambda}(\psi) = \{ b \mid (b, v, c_0, c^n) \in \psi \} - \{ b \mid (b, v, \Lambda, c_0, c^n) \in Q_\pi \left( q_{\Lambda}(i) \right) \} \) and in \( B_\Lambda(\psi) = B_q(i) \land \bigwedge_{b \in B_{\Lambda}(\psi)} B_q(i) \). Assuming a \( B_{\Lambda, \sigma} \) defined by analogy to \( B_\pi \), we have the following lemma:

**Lemma 4.** If \( c_i, \pi_i, \sigma_i \), and \( \psi_i \) are built from \((c_0, \pi_0, \sigma_0, c_0) = \mathcal{C}_H(\eta \in H) \) and \( \psi_0 = \Psi(\pi(\pi, \sigma)) \), then \( B_\Lambda(\psi_i) \) \( \implies B_{\pi, \sigma}(Y(\pi, \sigma), v_\pi, v_{\pi, \lambda}) \) where \((v_\pi, v_{\pi, \lambda}) \) depend on \( c_i \).

**Defining the process.** We discussed an iterative process satisfying certain properties above, let us now define it completely. Let \( U(\eta) = u_0, u_1, u_2, \ldots \) be a sequence where \( u_0 = (c, \tau, \pi, \sigma, Y(\pi, \sigma)) \) and given \( u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \), we compute \( u_{i+1} = (c_{i+1}, \tau_{i+1}, \pi_{i+1}, \sigma_{i+1}, \psi_{i+1}) \) as

- if \([i \text{ is even}]\) select \( t \in \tau_i \) and define \( c_{i+1} \) and \( \tau_{i+1} \) as discussed in the function invocation case. The remaining items are obtained as \( \tau_{i+1} = \pi_i \cup \pi_{i+1} = \sigma_i \cup \sigma_{i+1} \) and \( \psi_{i+1} = \psi_i \cup Y(\pi_{i+1}, \sigma_{i+1}) \cup Y(\pi, \sigma_{i+1}) \cup Y(\pi_{i+1}, \sigma_{i+1}) \).
- if \([i \text{ is odd}]\) select \( q \in \psi_i \) and define \( c_{i+1}, \tau_{i+1}, \sigma_{i+1} \) and \( \psi_{i+1} \) as in the function application case, and let \( \tau_{i+1} = \tau_i \cup \tau_q \).
Theorem 5. For \( \eta \in H \) with \( u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \in U(\eta) \), if \( m_i \models c_i \land B_f(\tau_i) \land B_\lambda(\psi_i) \), then \( m_i \) is interpretation-independent.

Proof. By noting that alternating unfoldings preserves validity, follows from Lemmas 3 and 4. \( \square \)

4.6 Eventual Unblocking

We have discussed an iterative process that progressively unfolds function calls and provides formulas with interpretation-independent models that prove accurate reflections of an evaluation input. We now wish to show that beyond soundness, our procedure is complete and is therefore guaranteed to find such an input if it exists. Note that our selection strategy for \( t_i \in \tau_i \) and \( q_i \in \psi_i \) in the previous section was left open. We now constraint it to first-in-first-out selection to provide breadth-first exploration of the remaining unfoldings. This requirement allows us to state that eventually, any blocker \( b \) will be unlocked as long as the concerned functions are terminating. Let us first define the set of blockers for \( u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \) given \( B(B(u)) = \{ b : (b, v, f, e^n) \in \pi_i \} \cup B_{\text{Unsat}}(\psi_i) \), which leads to the final theorem.

Theorem 6. For \( \eta \in H \) with \( \eta : \text{Boolean} \) such that for all \( f(c_1, \ldots, c_n) \subseteq \eta \), \( f \) is terminating and \( \exists m. \eta(m) \rightarrow \text{true} \), there is a \( u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \in U(\eta) \) for which \( \exists m_H. m_H \models c_i \land B_f(\tau_i) \land B_\lambda(\psi_i) \), and by converting \( m\eta \) to \( m_H \), we have \( \eta(m_H) \rightarrow \text{true} \).

In other words, for any negated verification property \( \eta \in H \) that has a counter-example, there comes a point \( u_i \) in our unfolding procedure \( U(\eta) \) where a model for \( c_i \) exists and this constitutes a counter-example to the considered verification property, ergo we have soundness and completeness.

The proof of Theorem 6 as well as the remaining theorems and lemmas is in the Appendix.

4.7 Soundness for Proofs

Up to now, we abstracted away the over-approximations (see 3.1) in our formalizations, but completeness depends on these as well. Note, however, that, for \( \eta \in H \) with \( \eta : \text{Boolean} \) and \( u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \in U(\eta) \), if \( c_i \) is Unsat, then clearly \( c_i \land B_f(\tau_i) \land B_\lambda(\psi_i) \) is Unsat, and furthermore, for any \( j > i \), we have that \( c_j \land B_f(\tau_j) \land B_\lambda(\psi_j) \) is Unsat as well since \( c_j \) is obtained by adding top-level conjuncts to \( c_{j-1} \). These observations let us conclude that performing Unsat checks on \( c_i \) provide us simply with early guarantees that no counter-example can be reported in the future, so it does not change the set of cases when a counter-example is reported. This translates counter-example soundness and completeness to the procedure with both under- and over-approximation checks. The procedure stops as soon as it finds a counter-example or detects Unsat. If a counter-example exists, it is eventually found. If Unsat is reported, we know that no counter-example is reported, and, by completeness, no counter-example exist. This establishes soundness for proofs (Unsat answers) as well.

5. Evaluation

We have implemented our technique within the Leon verifier. Our implementation is available in the master branch of the public Leon repository.\(^1\) The results of our initial evaluation are presented in Table 1. Our set of benchmarks covers the verification of different program properties involving higher-order functions. We mostly focus on recursive data-structures for which the framework is particularly well adapted, but also showcase various other verification tasks that illustrate the flexibility of the tool. The set of list operations we verify mainly consists in different correspondence properties between higher-order operators mixed in with a few equivalent first-order recursive definitions. We also verify associativity of certain operators such as map and flatMap as well as fold reassociativity.

\(^1\)https://github.com/epfl-lara/leon

<table>
<thead>
<tr>
<th>Operation</th>
<th>LoC</th>
<th>V</th>
<th>I</th>
<th>U</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>List.forall</td>
<td>105</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>0.44</td>
</tr>
<tr>
<td>List.exists</td>
<td>20</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0.17</td>
</tr>
<tr>
<td>List.map</td>
<td>60</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0.31</td>
</tr>
<tr>
<td>List.sort</td>
<td>51</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0.11</td>
</tr>
<tr>
<td>List.flatMap</td>
<td>48</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0.24</td>
</tr>
<tr>
<td>List.foldRight</td>
<td>101</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0.94</td>
</tr>
<tr>
<td>CommutativeFold</td>
<td>141</td>
<td>18</td>
<td>4</td>
<td>0</td>
<td>0.42</td>
</tr>
<tr>
<td>ListOps</td>
<td>111</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0.33</td>
</tr>
<tr>
<td>OptionMonad</td>
<td>49</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0.13</td>
</tr>
<tr>
<td>DeMorganSets</td>
<td>23</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.07</td>
</tr>
<tr>
<td>AssocSets</td>
<td>23</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.07</td>
</tr>
<tr>
<td>SetOps</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td>Closures</td>
<td>50</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0.20</td>
</tr>
<tr>
<td>Continuations</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.07</td>
</tr>
<tr>
<td>Switch</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0.07</td>
</tr>
<tr>
<td>Transformations</td>
<td>49</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0.63</td>
</tr>
<tr>
<td>ParBalanceFold</td>
<td>206</td>
<td>33</td>
<td>0</td>
<td>2</td>
<td>0.45</td>
</tr>
<tr>
<td>FiniteQuantifiers</td>
<td>39</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>157.00</td>
</tr>
<tr>
<td>Total</td>
<td>1082</td>
<td>149</td>
<td>15</td>
<td>2</td>
<td>161.69</td>
</tr>
<tr>
<td>Total (non-degenerate)</td>
<td>847</td>
<td>115</td>
<td>15</td>
<td>0</td>
<td>4.69</td>
</tr>
</tbody>
</table>
cation conditions, including match exhaustiveness checks and call-site precondition checks; not all of these verification conditions end up referring to higher-order functions. One should also note here that call-site precondition verification must be deferred when passing named functions to higher-order functions. This can be performed during unfolding or preconditions can simply be desugared as implying the postcondition.

We have focused in this work on counter-example finding, for which our system is complete. That said, the results also show that there are also many valid specifications involving higher-order functions that our system can prove. We have found many useful properties that can be expressed and proved correct, despite the fact that our specification language does not support quantifiers in specifications. Because we have not yet integrated more sophisticated inductive reasoning of CVC4 [13], some of the properties are written containing proof hints to specify the necessary recursion schema. These hints are specified directly in the input language as recursive function calls, and they do not require special handling by the framework. The ability to specify hints and automate induction is outside of the scope of the present paper.

We find the running time of our tool to be usable for interactive development of verified software with higher-order functions. There are degenerate cases where the running time is extremely poor as one can see in the FiniteQuantifiers case in Table 1. This benchmark uses finite lists as universal and existential quantification domains and closures that themselves perform finite quantification checks are specified as quantified formulas. This leads to combinatorial explosion and the tool ends up performing search in a large call-tree (just under $10^{32}$ nodes) with regular (and extremely large) solver queries.

Each proved property can be used as a basis for further proofs, thus providing good scalability to large but modular projects. Even in the presence of invalid specifications for which counter-examples will be reported, verification of valid properties does not suffer a performance hit, so the tool can easily be integrated into a development workflow, where the validity of verification conditions is not known in advance.

6. Related Work

Automated first-order program verification already boasts impressive results and has resulted in industrial-grade frameworks such as ACL2 [6] and Spec# [1]. When dealing with pure functional languages, we can leverage their mathematical structure and have sound inductive proofs in a counter-example complete procedure [3, 19]. However, reasoning about higher-order functions is hard [15, 16], and the field still in its infancy with tool support lacking.

**Dependent refinement types** provide a powerful avenue for higher-order functional verification and have been applied in Liquid Types [14] as well as Liquid Haskell [20] that extended the technique to call-by-name evaluation. Refinement types enable predicate specification on program types which can then be used constructively for sound verification. Furthermore, type invariants can be further strengthened through counter-example-guided abstraction refinement and the system can therefore report witnesses to invalid specifications in some cases. However, counter-examples are not the focus of these systems and they boast no completeness-results in that direction.

**Model checking higher-order recursion schemas** is another main techniques used in higher-order function verification [7]. This approach reduces the verification problem to an equivalent one of model checking through source analysis by turning the input program into a (possibly) infinite tree where each path represents an event sequence in the program execution. Once a model has been built, it can be checked using HORS to determine validity. Type refinements can also be leveraged during model creation and many refinement techniques can be applied in this setting as well. Recursion schemas are not well suited for handling infinite domains, but this limitation has proven to be a worthwhile research direction and has been (partially) addressed in later work [8, 12].

**Higher-order logic provers.** Among the most powerful generalization of our approach are techniques employed in the LEO II prover [2], which guarantee completeness for proofs for certain semantics of higher-order logic, and can also detect non-theorems. While we were not able to make direct experimental comparisons, additional encoding would be needed to describe the data type and integer theories we use within the higher-order logic supported by LEO II. We expect that the generality of these approaches will translate into lower performance for finding counter-examples for our benchmarks. Another related avenue are powerful interactive proof-assistants such as Isabelle/HOL [10] or Coq [5]. These frameworks are also capable of reasoning about universal quantification and do so in a somewhat more predictable manner, but typically require interaction. Counterexample finders such as Alloy* [9] and Nitpick [4] can handle propositions in higher-order logics. These tools offer a high level of automation and boast impressive theoretical results with sound handling of universal and existential quantification. However, completeness in Alloy* is limited to bounded domains. Nitpick supports unbounded domains, but we are not aware of its completeness guarantees.

**Reasoning using first-order quantifiers** enables encoding higher-order functions, but completeness guarantees are missing with current first-order theorem provers and SMT solvers. Dafny verifier has a limited support for higher-order functions https://dafny.codeplex.com. However, the nature of the support for quantifiers precludes their use in a system that aims for completeness result such as ours.
7. Conclusions and Analysis

The techniques we presented offer complete counter-example discovery for pure higher-order recursive functional programs using quantifier-free logic. The procedure constructs a binary decision tree with blocked branches and iteratively extends/unblocks paths until a valid model is found. This procedure can be viewed as an iteratively increasing under-approximation. The extension to the initial procedure with higher-order functions retains the same philosophy of eventual validity, thus maintaining completeness. Interestingly, the technique also enables proofs for a variety of programs using higher-order functions. Furthermore, the examples we have where proofs fail do not seem restricted by our extension, but by the first-order reasoning procedure that fails to discover invariants for complex inductive steps. Finally, the guarded unfolding technique we presented could open the way to reasoning about other programming language features such as objects with subtyping.

Acknowledgments

We thank Philippe Suter for his contributions to Leon and for discussions about verification of higher-order functions.

References


Appendix: Proofs

List of complete proofs omitted when discussing counterexample completeness.

Theorem 2. For η ∈ H with η : T for some T ∈ Htype and m_H |= C_H(η), if m_H is interpretation-independent, then corresponding m_H is such that m_H[η] → true.

Proof. We will start by defining a helper function C_λ for η_i ⊆ η and associated b_i where C_λ(η_i, b_i) = c ∧ e given (c, e, τ, σ) = C(η_i, b_i). Note that C_λ(η_i, b_i) depends on all conjunctions generated in C for the pair (η_i, b_i).

Let us prove by induction that for η_i ⊆ η with associated b_i, if m_H |= b_i then

m_H |= C_λ(η_i, b_i) ⇒ η_i[m_H] → true (4)

m_H |= ∼C_λ(η_i, b_i) ⇒ η_i[m_H] → false (5)

m_H |= C_λ(η_i, b_i) = C(λ) ⇒ η_i[m_H] → λ ∈ H_λ (6)

If η_i ∼ (Definition)* Expr, then the induction step is trivial. The same holds for η_i ∼ g ∈ H_val and η_i ∼ v ∈ H_var. If η_i ∼ ∼η_j ∈ H, then the definition of C tells us that b_j associated to η_j is the same as b_i. Therefore, m_H |= C_λ(η_j, b_j) implies both η_j[m_H] → true (by induction) and m_H |= ∼C_λ(η_i, b_i), which gives us m_H[η_i] → false. Consequently, we have proposition 5, and 4 by symmetry. Note that we can safely ignore 6 since η is well-typed.

Let us now consider η_i ∼ f(η_A1, ..., η_An). Given interpretation-independence, we know that either the uninterpreted result is non-critical to the model, or the corresponding unfolding t has already taken place. The first case is identical to η_i ∼ v ∈ H_var and the hypothesis holds. In the second case, all sub-term b_i’s are the same as b_i so the induction hypothesis holds for η_A1, ..., η_An. Let
us augment models $m_H$ and $m_H'$ to $m'_H$ and $m''_H$ by respectively adding bindings for $f_{arg,1}$ and $\forall f_{arg}$. We described unfolding equivalence in $H$, so assuming by symmetry that $m'_H = C_\lambda(f_{body}(b_1)), f_{body}(m_H') \rightarrow \text{true}$ and these observations imply both $m_H \models I_{\lambda}(\eta_1, b_1)$ and $\eta[m_H] \rightarrow \text{true}$. The $\eta \sim \eta_{\lambda(\eta_{A1}, ..., \eta_{An})}$ case is similar but when dealing with the unfolded case for $(p, \lambda)$, we must also consider $\eta_{C}(m_H' \rightarrow \lambda_k$ where $\lambda_k \neq \lambda$. If this is the case, $m_H \models L(\lambda_k) \neq L(\lambda)$ and therefore $m_H \models \neg b_p$ from $I_{\lambda}(p, \lambda)$, so we fall back to the $\eta \sim v \in H_{\text{var}}$ case and preserve validity.

It remains to consider $\eta \sim \text{if}(\eta_k) \eta_k$ else $\eta_k$. We can assume by symmetry that $m_H \models C_\lambda(\eta_k, b_k)$ and $m_H \models C_\lambda(\eta_k, b_k)$ and therefore $m_H \models C_\lambda(\eta_k, b_k)$. The definition of $C$ again tells us that $b_k$ associated to $\eta_k$ is the same as $b_k$ and the induction hypothesis implies that $\eta_{C}(m_H) \rightarrow \text{true}$. We also know given the definitions of $\epsilon$ in the case of $C$ that $m_H \models b_k = b_k \mapsto C_\lambda(\eta_k, b_k) \mapsto b_k$ and therefore $m_H \models b_k$. Again, the induction hypothesis tells us that $\eta_{C}(m_H) \rightarrow \text{true}$, and evaluation rules on $H$ give us $\eta_{C}(m_H') \rightarrow \text{true}$.

To complete the proof, it suffices to note that $m_H \models b_{\text{start}}$ and $m_H' \models C_\lambda(\eta, b_{\text{start}})$ by construction and we therefore have $\eta_{m_H'} \rightarrow \text{true}$. □

Lemma 3. If $(c_i, \tau_i)$ are built from $(c_0, \tau_0, \pi_0, \sigma_0) = C_\lambda(\eta \in H)$, then $B_{\lambda}(\tau_i) \mapsto B_{\lambda}(\tau_{all}, v_r, v_t)$ where $(v_r, v_t)$ depend on $c_i$ and $\tau_{all} = \bigcup_{i=0}^n \tau_i$ is the union of all $\tau$ generated during unfolding.

Proof. Let us start by defining $t_{uf}(i) = \{t_j | 0 \leq j < i\}$, $v_{uf}(i) = \{v \in (b, v, f, c^o) \in t_{uf}(i)\}$ and $V_i = \{v \in (b, v, f, c^o) \in (\tau_i \cup t_{uf}(i))\}$. We know by construction that $v_r \subseteq V_i$ and given the definition of unfolding, $v_{uf}(i) \subseteq v_t$ which gives us $V_i - v_{uf}(i) \subseteq v_r - v_t$.

Lemma 4. If $c_i$, $\tau_i$, $\pi$ and $\psi_i$ are built from $(c_0, \tau_0, \pi_0, \sigma_0) = C_\lambda(\eta \in H)$ and $\psi_0 = Y(\pi_0, \sigma_0)$, then $B_{\lambda}(\psi_i) \mapsto B_{\lambda}(Y(\pi, \sigma), v_r, v_{p, \lambda})$ where $(v_r, v_{p, \lambda})$ depend on $c_i$.

Proof. For any $v \in v_r$, we either have (1) an associated $(b, v, \Lambda_i, c^o) \in Q_{\pi}(q_{uf}(i))$ or (2) a $(b, v, f, c^o) \in \psi_i$. Note that we consider these two cases as distinct, realizing the second only if the first falls through.

1. Given the definition of unfolding, we have $I_{\lambda}(p, \lambda)$ a top-level conjunct in $c_i$ for all $\lambda \in \Lambda$. Hence, either $m_i \models c_0 = L(\lambda)$ for one of these $\lambda$’s or we have $m_i \models \neg b_0$, both options leading to interpretation-independence.

2. We have that $b \in B_{\lambda_{ef}}(\psi_i)$ by definition and interpretation-independence is therefore also guaranteed.

Theorem 6. For $\eta \in H$ with $\eta : \text{Boolean}$ such that for all $f(e_1, ..., e_n) \subseteq \eta$, $f$ is terminating and $\exists m. \eta[m] \rightarrow \text{true}$, there is a $u = (c_1, \tau_1, \pi_1, \sigma_1, \psi_1) \in U(\eta)$ for which $\exists m_H. m_H \models c_1 \wedge B_{\lambda}(\tau_1) \wedge B_{\lambda}(\psi_1)$, and by converting $m_H$ to $m', m_H \models \eta[m_H] \rightarrow \text{true}$.

Proof. We will begin by proving that for any $b$ from $\tau_i$ or $\pi_i$, there exists a $j > i$ such that $b \in B(\tau_j, \psi_j)$ where $B(\tau_j, \psi_j) = B_j(\tau_j) \cup B_{\lambda}(\psi_j)$. Let us argue by contradiction that there exists an infinite chain in $L(\eta)$ of $u_i, u_{i+1}, u_{i+2}, ...$ with $0 \leq i$ such that $b \in B(\tau_k, \psi_k)$ for all $k \geq i$.

We start by looking at which conditions are necessary for $b$ to belong to $B(\tau_{i+1}, \psi_{i+1})$ given $b \in B(\tau_i, \psi_i)$. We define $E_b$ to be the set of all expressions in $H$ such that if the body associated to $t_i$ or $q_i$ (depending on whether $i$ is even or odd) is in $E_b$, then $b \in B(\tau_{i+1}, \psi_{i+1})$. Given the definitions of $u_{i+1}$ and $C$, we can easily see that

$$E_b := f(e_1, ..., e_n), \quad |\lambda(e_1, ..., e_n), \quad |\text{if}(E_b) e_1 \text{ else } e_2 \quad |\neg E_b$$

We therefore have that an infinite chain of $u_k$ where $b \in B(\tau_k, \psi_k)$ must correspond to an infinite chain of alternating $t_k/q_k$ where the body of the function associated to each $t_k/q_k$ is in $E_b$. However, if such an infinite chain exists, then we have non-termination and our contradiction.

Let us now consider the $B_Q(i)$ clause. For $q = (b, v, \Lambda_i, c^o) \in Q_{\pi}(q_{uf}(i))$, only $\Lambda_i$ depends on $i$ and it is increasing in $i$ since any later $q_j$ with $j > i$ such that $q \pi \tau_j$ will imply $\Lambda_i \cup \{\Lambda_j(q_j)\} \subseteq \Lambda_{j+1}$. Also, due to the fair selection of $q_j$, for any $\lambda \in H_{\lambda}$ encountered during evaluation of $\eta[m]$, $\lambda \in \Lambda_k$ for some $k > 0$.

The model $m$ is given, so $\eta[m]$ is a valid input to the evaluator. We can therefore define the sets $I$ of all nodes $e = \text{if}(\text{COND}[m])$ THEN $\text{|m|} \text{ ELSE } |\text{m|}$ and $C$ of all nodes $e \in \lambda(e_1, ..., e_n)$ where $\lambda \in H_\lambda$. Finally, let $I_b$ be the union of all $\{b_i, \lambda_i\}$ generated at corresponding points $C(e, b)$ with $e$ in $I$ along with $b_{\text{start}}$ and $C_\lambda$ be the set of all caller $\lambda$’s in $C$. Note that all functions encountered are terminating so $I, I_b, C$ and $C_\lambda$ are finite.

We have just seen that for all $b \in I_b$, there exists a $k_b \in \mathbb{N}$ such that $b \notin B(\tau_k, \psi_k)$. Also, for all $\lambda \in C_\lambda$ there exists a $k_\lambda \in \mathbb{N}$ such that for all $(b, v, \Lambda_{k^*}, c^o) \in Q_{\pi}(q_{uf}(k_\lambda))$, $\lambda \in \Lambda_{k^*}$. Based on these, we can define

$$k = \max\{\max(k_b, \max(k_\lambda) \text{ for } b \in I_b, \lambda \in C_\lambda)$$

and let $m_H \models c_k \wedge B_{\lambda_k} \wedge B_{\lambda}(\psi_k)$. Since $m$ exists and all extra variables introduced by $C$ are free, $m_H$ is guaranteed to exist, and Theorems 2 and 5 ensure $\eta[m_H] \rightarrow \text{true}$ for $m_H$ associated to $m_H$. □