Counter-Example Complete Verification for Higher-Order Functions

Nicolas Voirol Etienne Kneuss Viktor Kuncak [*]

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

{firstname.lastname}@epfl.ch

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

[*] This work is supported in part by the European Research Council (ERC) Project *Implicit Programming* and Swiss National Science Foundation Grant "Constraint Solving Infrastructure for Program Analysis".

SCALA'15, June 13, 2015, Portland, OR, USA
ACM. 978-1-4503-3626-0/15/06
http://dx.doi.org/10.1145/2774975.2774978

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.

```
sealed abstract class Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Equals(e1: Expr, e2: Expr) extends Expr
case class Literal(i: Int) extends Expr

def transform(f: Expr ⇒ Option[Expr])(e: Expr): Expr = {
  val rec = (x: Expr) ⇒ transform(f)(x)
  val newExpr = e match {
    case Add(e1, e2) ⇒ Add(rec(e1), rec(e2))
    case Equals(e1, e2) ⇒ Equals(rec(e1), rec(e2))
    case Literal(i) ⇒ Literal(i)
  }
  f(newExpr).getOrElse(newExpr) }

def exists(f: Expr ⇒ Boolean)(expr: Expr): Boolean = {
  val rec = (x: Expr) ⇒ exists(f)(x)
  f(expr) || (expr match {
    case Add(e1, e2) ⇒ rec(e1) || rec(e2)
    case Equals(e1, e2) ⇒ rec(e1) || rec(e2)
    case Literal(i) ⇒ false }) }

def simplifyEquals(expr: Expr) = (transform {
  case Equals(Add(Literal(i), Literal(j)), e2) ⇒
    Some(Equals(Literal(i + j), e2))
  case Equals(e1, Add(Literal(i), Literal(j))) ⇒
    Some(Equals(e1, Literal(i + j)))
  case _ ⇒ None[Expr]()
} (expr)) ensuring (res ⇒ !exists {
  case Equals(_, Add(Literal(i), Literal(j))) ⇒ true
  case Equals(Add(Literal(i), Literal(j)), _) ⇒ true
  case _ ⇒ false
} (res))
```

Figure 1. Expression tree transformation

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 subexpression, 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)))
```

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 match {
    case Cons(h1, t1 @ Cons(h2, xs)) ⇒
      o.f(h1) ≤ o.f(h2) && isSorted(t1)
    case _ ⇒ true }

def split[T](list: List[T]): (List[T], List[T]) = (list match {
  case Cons(h1, Cons(h2, xs)) ⇒
    val (t1,t2) = split(xs)
    (Cons(h1, t1), Cons(h2, t2))
  case _ ⇒ (list, Nil())
}) ensuring { res ⇒
  contents(res._1) ++ contents(res._2) == contents(list) }

def merge[T](l1: List[T], l2: List[T])
            (implicit o: Ordering[T]): List[T] = {
  require(isSorted(l1) && isSorted(l2))
  (l1, l2) match {
    case (Cons(h1, t1), Cons(h2, t2)) ⇒
      if (o.f(h1) < o.f(h2)) Cons(h1, merge(t1, l2))
      else Cons(h2, merge(l1, t2))
    case _ ⇒ l1 ++ l2
  }} ensuring { res ⇒ isSorted(res) &&
    (contents(res) == contents(l1) ++ contents(l2)) }

def sort[T](list: List[T])(implicit o: Ordering[T]): List[T] =
  (list match {
    case Cons(h1, t1 @ Cons(h2, t2)) ⇒
      val (l1, l2) = split(list)
      merge(sort(l1), sort(l2))
    case _ ⇒ list
  }) ensuring (res ⇒ isSorted(res) &&
    contents(list) == contents(res))
```

Figure 2. Generic sorting function

```
def size[T](l: List[T]): BigInt = (l match {
  case Nil ⇒ 0
  case Cons(h, t) ⇒ 1 + size(t)
}) ensuring (_ ≥ 0)

def dup[T](l: Cons[T]): List[T] = ({
  Cons(l.head, l)
}) ensuring (r ⇒ size(r) > 1)
```

Figure 3. Duplication of the head of a list

3.1 Verifying Recursive First-Order Programs

Our procedure for first-order programs alternates between model construction (*i.e.* counter-example discovery) and proofs, by building a sequence of under- and over-approximations of our verification constraints. These approximations 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:

$$l = Cons(h, t) \wedge r = Cons(h, l) \wedge size(r) \leq 1$$

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.

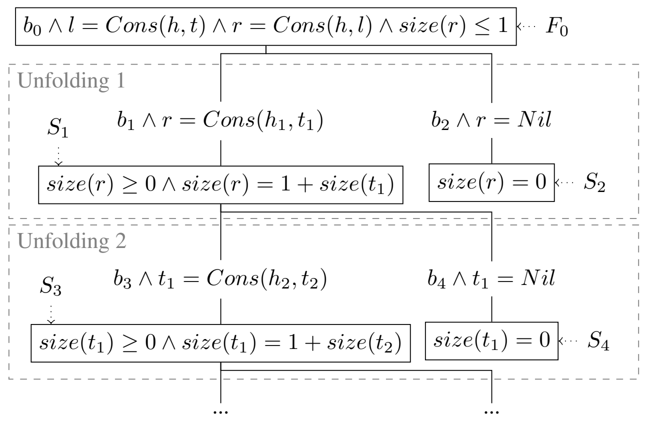
When under-approximating the constraint, we avoid all branches leading to function calls that have not been defined yet. This ensures that potential models only rely on well-defined portions. When over-approximating, the complete tree is used. Since function symbols are uninterpreted, calls that have not been explicitly constrained are treated as returning 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 k-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 $F_0 \wedge \neg b_0$ is trivially Unsat and the over-approximation F_0 is Sat. We thus unfold the call size(r) by pushing new constraints corresponding to the instrumented definition of size(r), and obtain F_1 equal to:

$$\begin{aligned} F_0 \quad & \wedge \ (b_1 \vee b_2) \\ & \wedge \ ((b_1 \wedge r = Cons(h_1, t_1)) \Rightarrow S_1) \\ & \wedge \ ((b_2 \wedge r = Nil) \Rightarrow S_2) \end{aligned}$$

Given that S_1 contains an unconstrained function call, the under-approximation avoids it by enforcing $\neg b_1$. Since $F_1 \wedge \neg b_1$ is Unsat and F_1 is Sat, we unfold size(t_1) and obtain F_2. Here again, $F_2 \wedge \neg b_3$ 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.



$b_0 \wedge l = Cons(h,t) \wedge r = Cons(h,l) \wedge size(r) \leq 1$
F_0
Unfolding 1
S_1
$b_1 \wedge r = Cons(h_1,t_1)$
$b_2 \wedge r = Nil$
$size(r) \geq 0 \wedge size(r) = 1 + size(t_1)$
$size(r) = 0$
S_2
Unfolding 2
S_3
$b_3 \wedge t_1 = Cons(h_2,t_2)$
$b_4 \wedge t_1 = Nil$
$size(t_1) \geq 0 \wedge size(t_1) = 1 + size(t_2)$
$size(t_1) = 0$
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 function 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 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 Λ the set of all closures and associate to $\lambda \in \Lambda$ its arguments $\lambda_{arg,1}, ..., \lambda_{arg,n}$ and body λ_{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 $\mathcal{L} : \Lambda \longrightarrow U$ from closures to their identifying values and $\Lambda_t = \{\lambda_1, ..., \lambda_m\} \subseteq \Lambda$ 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:

$$f(x^n) = \begin{cases} \lambda_{1body}\left[\lambda_{1arg}^{\,n} \longrightarrow x^n\right] & \text{if } f = \mathcal{L}(\lambda_1) \\ \vdots & \vdots \\ \lambda_{m body}\left[\lambda_{m arg}^{\,n} \longrightarrow x^n\right] & \text{if } f = \mathcal{L}(\lambda_m) \\ \text{uninterpreted} & \text{otherwise} \end{cases}$$

$f(x^n)$ 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 Λ_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 apply1(f: Int ⇒ Int): Int = f(1)
```

and the invocation $apply1(x \Rightarrow x + 2)$, 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 applyPair(p: (Int ⇒ Int, Int)): Int = p._1(p._2)
```

As $p.\_1$ is no function-typed argument of $applyPair$, 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

```
def applyPair(f: Int ⇒ Int, p: Int): 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 environment 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 $f_{arg,1}, ..., f_{arg,n}$ denote the arguments of f and f_{body} its body. Likewise, we call H_λ the set of closures in H and for $\lambda \in H_\lambda$, we define $\lambda_{arg,i}$, and λ_{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_{var} the set of variables and $H_{val} = \{\mathbf{true}, \mathbf{false}\} \cup H_\lambda$ the set of values in H. We also define H_{ground} as the set of ground terms in H, namely $\eta \in H$ such that $FV(\eta) = \emptyset$ where $FV(\eta \in H)$ is the set of free variables in the program η. Finally, we define H_{type} the set of types in H, and for a function $f \in H_f$, let $f_{T,1}, ..., f_{T,n}$ denote the types associated to the arguments of f and f_T the return type. We also define $\lambda_{T,i}$ and λ_T in a similar manner for $\lambda \in H_\lambda$. We then define the usual typing relation $H : H_{type}$ on H and can therefore define $H_{v:T}$ the set of variables in H that type to T along with $H_{f:T}$ 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 $H_{ground} : T$ with call-by-value for functions which give us the evaluation relation $H_{ground} \longrightarrow H_{val}$ as defined in Figure 6. Note that for any $\eta \in H$, such that $H : T \in H_{type}$, given a mapping m_H such that each $\forall v : T_v \in FV(\eta).v \in m_H \wedge m_H[v] : T_v$, $\eta[m_H] \in H_{ground} : T$ is obtained by substitution and $\eta[m_H] \longrightarrow g \in H_{val}$ 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 $\mathcal{H}$ 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 $\mathcal{H}_{var}$ the set of variables in $\mathcal{H}$ and $\mathcal{H}_{v:T}$ the set associated to theory T (B for boolean and U for uninterpreted). We also define $\mathcal{H}_{f:T}$ the set of uninterpreted functions with signature T where T is a tuple of types in $\{B, U\}$. We can give a more formal definition of $\mathcal{L}$ introduced in 3.2 as $\mathcal{L} : H_\lambda \longleftrightarrow \mathcal{H}_{v:U}$ a bijection between closures and uninterpreted variables in $\mathcal{H}$. We also define a bijection $\mathcal{V} : H_{var} \longleftrightarrow \mathcal{H}_{var}$ between variables of H and $\mathcal{H}$. Given both these two functions, one can trivially build a correspondence between free variable mapping $m_H : H_{var} \longrightarrow H_{val}$ and model $m_{\mathcal{H}} : \mathcal{H}_{val} \longrightarrow \{True, False\} \cup \mathcal{H}_{v:U}$ (note that $\mathcal{H}_{v:U}$ 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 $\mathcal{F}_T : H_{f:T} \longrightarrow \mathcal{H}_{f:T}$ and a mapping $\gamma : H_{type} \longrightarrow \mathcal{H}_{f:U,T}$. We use $\mathcal{F}_T$ to encode named function calls and γ to perform dynamic dispatch on closures.

4.2 Defining the Transformation

Given the above domain definitions, we define a transformation $\mathcal{C}$ from a program $\eta \in H$ to a formula $c \in \mathcal{H}$ 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

$$\neg\mathbf{false} \longrightarrow \mathbf{true} \qquad \neg\mathbf{true} \longrightarrow \mathbf{false} \qquad \mathbf{if}(\mathbf{true})\, e_t \,\mathbf{else}\, e_e \longrightarrow e_t \qquad \mathbf{if}(\mathbf{false})\, e_t \,\mathbf{else}\, e_e \longrightarrow e_e$$

$$\frac{\mathbf{if}(e_c)\, e_t\,\mathbf{else}\, e_e \qquad e_c \longrightarrow e'_c}{\mathbf{if}(e'_c)\, e_t\,\mathbf{else}\, e_e} \qquad \frac{e_1 \longrightarrow e'_1}{\neg e_1 \longrightarrow \neg e'_1} \qquad \frac{e_j \in P_{val}, 1 \le j \le i-1 \qquad e_i \longrightarrow e'_i}{f(e_1, ..., e_i, ..., e_n) \longrightarrow f(e_1, ..., e'_i, ..., e_n)}$$

$$\frac{e_j \in P_{val}, 1 \le j \le n}{f(e_1, ..., e_n) \longrightarrow f_{body}[f_{arg,1} \longrightarrow e_1, ..., f_{arg,n} \longrightarrow e_n]} \qquad \frac{e \longrightarrow e'}{e(e_1, ..., e_n) \longrightarrow e'(e_1, ..., e_n)}$$

$$\frac{\lambda \in H_\lambda \qquad e_j \in H_{val}, 1 \le j \le i-1 \qquad e_i \longrightarrow e'_i}{\lambda(e_1, ..., e_i, ..., e_n) \longrightarrow \lambda(e_1, ..., e'_i, ..., e_n)} \qquad \frac{\lambda \in H_\lambda \qquad e_j \in H_{val}, 1 \le j \le n}{\lambda(e_1, ..., e_n) \longrightarrow \lambda_{body}\,[\lambda_{arg,1} \longrightarrow e_1, ..., \lambda_{arg,n} \longrightarrow e_n]}$$

$$\langle \mathbf{def}\, f(f_{arg,1} : \mathbf{Boolean}, ..., f_{arg,n} : \mathbf{Boolean}) = f_{body}\rangle^*\, e \longrightarrow e$$

Figure 6. Evaluation rules for $H_{ground} \longrightarrow H_{val}$

$$\begin{array}{rcl}
H & ::= & \langle \textit{Definition}\rangle^*\, \textit{Expr} \\
\textit{Definition} & ::= & \mathbf{def}\, f(H_{var} : \textit{Type}\ \langle, H_{var} : \textit{Type}\rangle^*) : \textit{Type} = \textit{Expr} \\
\textit{Type} & ::= & \mathbf{Boolean} \mid \big(\textit{Type}\ \langle, \textit{Type}\rangle^*\big) \Rightarrow \textit{Type} \\
\textit{Expr} & ::= & H_{var} \mid H_{val} \mid \neg \textit{Expr} \\
& & \mid \mathbf{if}(\textit{Expr})\, \textit{Expr}\, \mathbf{else}\, \textit{Expr} \\
& & \mid \big(H_{var} : \textit{Type}\ \langle, H_{var} : \textit{Type}\rangle^*\big) \Rightarrow \textit{Expr} \\
& & \mid \textit{Expr}(\textit{Expr}\ \langle, \textit{Expr}\rangle^*\,) \\
& & \mid f(\textit{Expr}\ \langle, \textit{Expr}\rangle^*\,)
\end{array}$$

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 $\mathcal{C} : H \times \mathcal{H}_{v:B} \longrightarrow R$ 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 $t \in T = \mathcal{H}_{v:B} \times \mathcal{H} \times P_f \times \mathcal{H}^*$ for invocations (see case 6 in $\mathcal{C}$) and both $p \in \Pi = \mathcal{H}_{v:B} \times \mathcal{H} \times \mathcal{H} \times \mathcal{H}^*$ (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 $2^T \times 2^\Pi \times 2^\Sigma$.

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 $\mathcal{C}$). Note that the former can have any type in the considered theories and the later is boolean. We can now define $\mathcal{C} : H \times \mathcal{H}_{v:B} \longrightarrow \mathcal{H} \times \mathcal{H} \times 2^T \times 2^\Pi \times 2^\Sigma$ such that

0. $\mathcal{C}(\langle f \in H_f\rangle^*\,\mathrm{E}, b) = \mathcal{C}(\mathrm{E}, b)$
1. $\mathcal{C}(v \in H_{var}, b) = (\mathcal{V}(v), \emptyset, \emptyset, \emptyset, \emptyset)$
2. $\mathcal{C}(\mathbf{true}/\mathbf{false}, b) = (True/False, \emptyset, \emptyset, \emptyset, \emptyset)$
3. $\mathcal{C}\,(\lambda \in H_\lambda, b) = (\mathcal{L}(\lambda), e, \emptyset, \emptyset, \{\lambda\})$ where
   (a) $e = \bigwedge_{\lambda_i \in \{\text{previous } \lambda\text{'s}\}} \mathcal{L}(\lambda) \neq \mathcal{L}(\lambda_i)$
   $\quad \wedge\ \bigwedge_{v \in FV(\eta)} \mathcal{L}(\lambda) \neq \mathcal{V}(v)$ for η original program
4. $\mathcal{C}(\neg\mathrm{E}, b) = (\neg c, e, \tau, \pi, \sigma)$ where $(c, e, \tau, \pi, \sigma) = \mathcal{C}(\mathrm{E}, b)$
5. $\mathcal{C}(\mathbf{if}\ (\mathrm{COND})\ \mathrm{THEN}\ \mathbf{else}\ \mathrm{ELSE}, b) = (r, e, \tau, \pi, \sigma)$ where given $b_t, b_e \in \mathcal{H}_{v:B}$ fresh variables and $(c, e, \tau, \pi, \sigma)_{[c,t,e]} = \mathcal{C}\,([\mathrm{COND}, \mathrm{THEN}, \mathrm{ELSE}]\,, [b, b_t, b_e])$, let
   (a) $r = \mathcal{V}(r_H \in H_{v:T})$ where r_H is a fresh variable and $\mathrm{THEN} : T$ and $\mathrm{ELSE} : T$
   (b) $e = e_c \wedge e_t \wedge e_e$
   $\quad \wedge\ b \implies (c_c \implies b_t \wedge \neg c_c \implies b_e)$
   $\quad \wedge\ b \implies (b_t \vee b_e) \wedge (\neg b_t \vee \neg b_e)$
   $\quad \wedge\ b_t \implies (r = c_t)$
   $\quad \wedge\ b_e \implies (r = c_e)$
   (c) $\tau = \tau_c \cup \tau_t \cup \tau_e$, $\pi = \pi_c \cup \pi_t \cup \pi_e$ and $\sigma = \sigma_c \cup \sigma_t \cup \sigma_e$
6. $\mathcal{C}(f(\mathrm{ARG}_1, ..., \mathrm{ARG}_n), b) = (v, e, \tau, \pi, \sigma)$ where given $(c_i, e_i, \tau_i, \pi_i, \sigma_i) = \mathcal{C}(\mathrm{ARG}_i, b)$ for $1 \le i \le n$, let
   (a) $v = \mathcal{F}_{f_T^n \Rightarrow f_T}(f)(c_1, ..., c_n)$
   (b) $\tau = \{(b, v, f, [c_1, ..., c_n])\} \cup \bigcup_{i=1}^n \tau_i$
   (c) $e = \bigwedge_{i=1}^n e_i$, $\pi = \bigcup_{i=1}^n \pi_i$ and $\sigma = \bigcup_{i=1}^n \sigma_i$
7. $\mathcal{C}\,(\mathrm{C}\,(\mathrm{ARG}_1, ..., \mathrm{ARG}_n)\,, b) = (v, e, \tau, \pi, \sigma)$ where given $(c_i, e_i, \tau_i, \pi_i, \sigma_i) = \mathcal{C}(\mathrm{ARG}_i, b)$ for $1 \le i \le n$ and $(c_0, e_0, \tau_0, \pi_0, \sigma_0) = \mathcal{C}(\mathrm{C}, b)$, let
   (a) $v = \gamma(T)(c_0, c_1, ..., c_n)$ where $\mathrm{C} : T$

(b) $\pi = \{(b, v, c_0, [c_1, ..., c_n])\} \cup \bigcup_{i=0}^{n} \pi_i$

(c) $e = \bigwedge_{i=0}^{n} e_i$, $\tau = \bigcup_{i=0}^{n} \tau_i$ and $\sigma = \bigcup_{i=0}^{n} \sigma_i$

We further define the functions $\mathcal{C}_H : H \longrightarrow \mathcal{H} \times 2^T \times 2^\Pi \times 2^\Sigma$ and $\mathcal{C}_\mathcal{H} : H \longrightarrow \mathcal{H}$: given $\eta \in H$, let b_{start} be a fresh variable and compute $(c, e, \tau, \pi, \sigma) = \mathcal{C}(\eta, b_{start})$. Let $r = c \wedge e \wedge b_{start}$ in $\mathcal{C}_H(\eta) = (r, \tau, \pi, \sigma)$ and $\mathcal{C}_\mathcal{H}(\eta) = r$.

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 $e_1 = f(\text{ARG}_1, ..., \text{ARG}_n) \subseteq \eta$, let us define $e_f = f_{body}\left[f^n_{arg} \longrightarrow \text{ARG}^n\right]$ and $\eta_f = \eta\left[e_1 \longrightarrow e_f\right]$. Also, for $e_2 = \text{C}(\text{ARG}_1, ..., \text{ARG}_n) \subseteq \eta$ with $\text{C}[m_H] \longrightarrow \lambda$, we define $e_\lambda = \lambda_{body}\left[\lambda^n_{arg} \longrightarrow \text{ARG}^n\right]$ and $\eta_\lambda = \eta\left[e_2 \longrightarrow e_\lambda\right]$. These unfoldings preserve evaluation and give us for $g \in H_{val}$ that

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

We now want to define unfolding for formulas in $\mathcal{H}$. Given $(c, \tau, \pi, \sigma) = \mathcal{C}_H(\eta)$, we define function invocation unfolding for $t = (b, v, f, c^n) \in \tau$. Let $(c_f, r_f, \tau_t, \pi_t, \sigma_t) = \mathcal{C}(f_{body}, b)\left[\mathcal{V}(f^n_{arg}) \longrightarrow c^n\right]$ and $I_f(t) = r_f \wedge (b \implies v = c_f)$ in $c_t = c \wedge I_f(t)$, the unfolding of t in c. We know from the definition of $\mathcal{C}$ that $b \implies P(v)$ in c for some proposition P, so c_t is equivalent to $c\left[v \longrightarrow c_f\right] \wedge r_f$. Therefore, for any model $m_\mathcal{H}$, we have

$$m_\mathcal{H} \models c_t \implies m_\mathcal{H} \models c.$$

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 $\mathcal{L}(\lambda)$. Let $b_p = b \wedge (c_0 = \mathcal{L}(\lambda))$, $(c_\lambda, r_\lambda, \tau_p, \pi_p, \sigma_p) = \mathcal{C}(\lambda_{body}, b_p)\left[\mathcal{V}(\lambda^n_{arg}) \longrightarrow c^n\right]$ and $I_\lambda(p, \lambda) = r_\lambda \wedge (b_p \implies v = c_\lambda)$ in $c_p = c \wedge I_\lambda(p, \lambda)$, the unfolding of p in c conditional on $c_0 = \mathcal{L}(\lambda)$. Note that when we require equality between c_0 and $\mathcal{L}(\lambda)$, this is modulo a given model $m_\mathcal{H}$, so the full statement would be $m_\mathcal{H} \models c_0 = \mathcal{L}(\lambda)$. Our definition of $\mathcal{C}$ guarantees a top-level conjunct in c_p that states $\mathcal{L}(\lambda) \neq \mathcal{L}(\lambda_i)$ for any $\lambda_i \neq \lambda$ and $\mathcal{L}(\lambda) \neq v$ for v in $FV(\eta)$, so any model $m_\mathcal{H} \models c_p$ will provide a valid equality check between c_0 and $\mathcal{L}(\lambda)$. Again, for any model $m_\mathcal{H}$, we have

$$m_\mathcal{H} \models c_p \implies m_\mathcal{H} \models c.$$

It is interesting to note that this definition of unfolding function applications extends to any caller variable including ν such that $\nu \in FV(\eta)$ for η 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 $\mathcal{I}_f(c, t) = (c_t, \tau_t, \pi_t, \sigma_t)$ and $\mathcal{I}_\lambda(c, p, \lambda) = (c_p, \tau_p, \pi_p, \sigma_p)$.

4.4 Interpretation Independence

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

In our definition of $\mathcal{C}$, function invocations and applications are handled by replacing them by a fresh variable in the resulting formula. We call these calls *uninterpreted* and it is clear that for a formula $c = \mathcal{C}_\mathcal{H}(\eta \in H)$ with model $m_\mathcal{H} \models c$ and associated m_H, if $m_\mathcal{H}$ depends on such calls then $(c, m_\mathcal{H})$ may not accurately reflect (η, 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 $(c, \tau, \pi, \sigma) = \mathcal{C}_\mathcal{H}(\eta)$ and model $m_\mathcal{H} \models c$, we define $v_\tau = \{v \mid (b, v, f, c^n) \in \tau\}$ 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

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

$$v_{p,\lambda} = \left\{ v \mid \begin{array}{c} I_\lambda((b, v, c_0, c^n) \in \pi, \lambda \in \sigma) \in TLC(c) \\ \wedge\ m_\mathcal{H} \models c_0 = \mathcal{L}(\lambda) \end{array} \right\}$$

We call $m_\mathcal{H}$ interpretation-independent if $\forall m \neq m_\mathcal{H}$ such that $m[v_i] = m_\mathcal{H}[v_i]$ for all $v_i \in UF(c) - (v_\tau - v_t) - (v_\pi - v_{p,\lambda})$ 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 η as $\mathcal{C}$ 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 $T \in H_{type}$ and $m_\mathcal{H} \models \mathcal{C}_\mathcal{H}(\eta)$, if $m_\mathcal{H}$ is interpretation-independent, then corresponding m_H is such that $\eta[m_H] \longrightarrow$ **true**.*

Proof. We will start by defining a helper function $\mathcal{C}_\wedge$ for $\eta_i \subseteq \eta$ and associated b_i where $\mathcal{C}_\wedge(\eta_i, b_i) = c \wedge e$ given

$(c, e, \tau, \pi, \sigma) = \mathcal{C}(\eta_i, b_i)$. Note that $\mathcal{C}_\wedge(\eta_i, b_i)$ depends on all conjuncts generated in $\mathcal{C}$ for the pair (η_i, b_i).

We prove by induction that for $\eta_i \subseteq \eta$ with associated b_i, if $m_\mathcal{H} \models b_i$ then

$$m_\mathcal{H} \models \mathcal{C}_\wedge(\eta_i, b_i) \implies \eta_i[m_H] \longrightarrow \textbf{true} \quad (1)$$
$$m_\mathcal{H} \models \neg\mathcal{C}_\wedge(\eta_i, b_i) \implies \eta_i[m_H] \longrightarrow \textbf{false} \quad (2)$$
$$m_\mathcal{H} \models \mathcal{C}_\wedge(\eta_i, b_i) = \mathcal{L}(\lambda) \implies \eta_i[m_H] \longrightarrow \lambda \in H_\lambda (3)$$

The full inductive proof can be found in the Appendix.

To complete the proof, it suffices to note that $m_\mathcal{H} \models b_{start}$ and $m_\mathcal{H} \models \mathcal{C}_\wedge(\eta, b_{start})$ by construction and we therefore have $\eta[m_H] \longrightarrow \textbf{true}$. □

4.5 Blocking Calls

Now that we have a transformation from programs $\eta \in H$ to formulas $(c, \tau, \pi, \sigma) = \mathcal{C}_H(\eta)$ and the definition of a class of formulas and models which accurately reflect programs and inputs, we need a bridge from one to the other.

The transformation $\mathcal{C}$ guarantees that all branches in the decision tree are associated a fresh variable b_t or b_e and for each function call in η, we have either $(b, v, f, c^n) \in \tau$ or $(b, v, c_0, c^n) \in \pi$ where $b \in \{b_t, b_e \text{ generated by } \mathcal{C}\} \cup \{b_{start}\}$. We therefore have that each function call appears on the right-hand side of an implication of the shape $b \implies P(v)$ in c where b is fresh and encodes branch selection during evaluation. Based on these observations, any model $m_\mathcal{H} \models c$ such that $m_\mathcal{H} \models \neg b$ must be interpretation-independent with respect to v.

Function invocations. Given v_τ and v_t from Definition 1, we can define $\mathcal{B}_\tau(\tau, v_\tau, v_t) = \bigwedge_{(b,v,f,c^n)\in\tau\wedge v\in(v_\tau - v_t)} \neg b$ which gives us that any $m_\mathcal{H} \models c \wedge \mathcal{B}_\tau(\tau, v_\tau, v_t)$ is interpretation-independent with respect to all v generated during function invocation transformation by definition of interpretation-independence. Unfortunately, the definition of v_t is not well suited to building an iterative process for (c, τ) as it is rather abstract. However, given c_i, τ_i and $t_i \in \tau_i$, we can build c_{i+1} and τ_{i+1} such that $(c_{i+1}, \tau_t, \pi_t, \sigma_t) = \mathcal{I}_f(t)(c, t_i)$ and $\tau_{i+1} = (\tau_i - \{t_i\}) \cup \tau_t$. Based on these, we can define $\mathcal{B}_f(\tau_i) = \bigwedge_{(b,v,f,c^n)\in\tau_i} \neg b$ and prove the following lemma:

Lemma 3. *If (c_i, τ_i) are built from $(c_0, \tau_0, \pi_0, \sigma_0) = \mathcal{C}_H(\eta \in H)$, then $\mathcal{B}_f(\tau_i) \implies \mathcal{B}_\tau(\tau_{all}, v_\tau, v_t)$ where (v_τ, v_t) depend on c_i and $\tau_{all} = \bigcup_{j=0}^{i} \tau_i$ is the union of all τ generated during unfolding.*

Function applications. Dealing with v_π and $v_{p,\lambda}$ is slightly more complex as we have the added constraint of $m_\mathcal{H} \models \mathcal{L}(\lambda) = c_0$, so set transformations are not sufficient to build a valid process. We introduce here the cartesian product type $\Psi = \mathcal{H}_{v:B} \times \mathcal{H}_{var} \times H_\lambda \times \mathcal{H}_{v:U} \times \mathcal{H}^*$ with associated product operator $Y : 2^\Pi \times 2^\Sigma \longrightarrow 2^\Psi$ and projectors $\mathcal{P}_{[b,v,\lambda,c_0,c^n]}((b, v, \lambda, c_0, c^n) \in \Psi) = [b, v, \lambda, c_0, c^n]$. We can now define an iterative process for (c, π, σ, ψ) such that given c_i, π_i, σ_i, ψ_i and $q_i \in \psi_i$, let $(c_{i+1}, \tau_q, \pi_q, \sigma_q) = \mathcal{I}_\lambda(q_i)$ in $\pi_{i+1} = \pi_i \cup \pi_q$, $\sigma_{i+1} = \sigma_i \cup \sigma_q$ and $\psi_{i+1} = (\psi_i - \{q\}) \cup Y(\pi_i, \sigma_q) \cup Y(\pi_q, \sigma_i) \cup Y(\pi_q, \sigma_q)$. Note that π_i and σ_i are strictly increasing with respect to set inclusion, and $\bigcup_{j=0}^{i} \psi_j = Y(\pi_i, \sigma_i)$. In other words, ψ_i is the cartesian product of π_i and σ_i minus the q_i selected at each iteration. Now observe that for each $q_i = (b, v, \lambda, c_0, c^n)$, if $m_i \models c_i$ 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 π_q on Ψ such that $q_1 \;{}_{\pi_q} q_2$ iff q_1 and q_2 share a common source in Π. 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 ψ with respect to π_q. For $q_\pi \in Q_\pi(Q)$, all elements share a common (b, v, c_0, c^n), so we can view q_π as $(b, v, \Lambda, c_0, c^n)$ where $\Lambda = \{\mathcal{P}_\lambda(q) \mid q \in q_\pi\}$. If we look at $q_{uf}(i) = \{q_j \mid 0 \le j < i\}$, for $q_\pi = (b, v, \Lambda, c_0, c^n) \in Q_\pi(q_{uf}(i))$, if there exists a $\lambda \in \Lambda$ such that $m_i \models \mathcal{L}(\lambda) = c_0$, then m_i is interpretation-independent with respect to v. Also, we have that if $m_i \models \neg b$ then m_i is interpretation-independent with respect to v as v is found on the right-hand side of an implication from b in c_i. These observations lead to the following constraint on b given Λ and c_0

$$\mathcal{B}_q(q_\pi) = \neg\left(\bigvee_{\lambda\in\Lambda} c_0 = \mathcal{L}(\lambda)\right) \implies \neg b$$

Furthermore, we can extend this constraint to all unfoldings as

$$\mathcal{B}_Q(i) = \bigwedge_{q_\pi \in Q_\pi(q_{uf}(i))} \mathcal{B}_q(q_\pi)$$

Finally, let $\mathcal{B}_{left}(\psi_i) = \{b \mid (b, v, c_0, c^n) \in \psi_i\} - \{b \mid (b, v, \Lambda, c_0, c^n) \in Q_\pi(q_{uf}(i))\}$ and in $\mathcal{B}_\lambda(\psi_i) = \mathcal{B}_Q(i) \wedge \bigwedge_{b\in\mathcal{B}_{left}(\psi_i)} \neg b$. Assuming a $\mathcal{B}_{\pi,\sigma}$ defined by analogy to $\mathcal{B}_\tau$, we have the following lemma:

Lemma 4. *If c_i, π_i, σ_i and ψ_i are built from $(c_0, \tau_0, \pi_0, \sigma_0) = \mathcal{C}_H(\eta \in H)$ and $\psi_0 = Y(\pi_0, \sigma_0)$, then $\mathcal{B}_\lambda(\psi_i) \implies \mathcal{B}_{\pi,\sigma}(Y(\pi, \sigma), v_\pi, v_{p,\lambda})$ where $(v_\pi, v_{p,\lambda})$ depend on c_i.*

Defining the process. We discussed an iterative process satisfying certain properties above, let us now define it completely. Let $\mathcal{U}(\eta) = u_0, u_1, u_2, ...$ 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 is even] select $t \in \tau_i$ and define c_{i+1} and τ_{i+1} as discussed in the function invocation case. The remaining items are obtained as $\pi_{i+1} = \pi_i \cup \pi_t$, $\sigma_{i+1} = \sigma \cup \sigma_t$ and $\psi_{i+1} = \psi_i \cup Y(\pi_i, \sigma_t) \cup Y(\pi_t, \sigma_i) \cup Y(\pi_t, \sigma_t)$.
- if [i is odd] select $q \in \psi_i$ and define c_{i+1}, π_{i+1}, σ_{i+1} and ψ_{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 $u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \in \mathcal{U}(\eta)$, if $m_i \models c_i \wedge \mathcal{B}_f(\tau_i) \wedge \mathcal{B}_\lambda(\psi_i)$, 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 $\mathcal{B}\ \mathcal{B}(u_i) = \{b \mid (b, v, f, c^n) \in \pi_i\} \cup \mathcal{B}_{left}(\psi_i)$, which leads to the final theorem.

Theorem 6. *For $\eta \in H$ with η : $\mathbf{Boolean}$ such that for all $f(e_1, ..., e_n) \subseteq \eta$, f is terminating and $\exists m.\eta[m] \longrightarrow \mathbf{true}$, there is a $u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \in \mathcal{U}(\eta)$ for which $\exists m_\mathcal{H}.m_\mathcal{H} \models c_i \wedge \mathcal{B}_f(\tau_i) \wedge \mathcal{B}_\lambda(\psi_i)$, and by converting $m_\mathcal{H}$ to m_H, we have $\eta[m_H] \longrightarrow \mathbf{true}$.*

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 $\mathcal{U}(\eta)$ 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 η : $\mathbf{Boolean}$ and $u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \in \mathcal{U}(\eta)$, if c_i is Unsat, then clearly $c_i \wedge \mathcal{B}_f(\tau_i) \wedge \mathcal{B}_\lambda(\psi_i)$ is Unsat, and furthermore, for any $j > i$, we have that $c_j \wedge \mathcal{B}_f(\tau_j) \wedge \mathcal{B}_\lambda(\psi_j)$ 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.

Table 1. Summary of evaluation results, featuring lines of code, (V)alid, (I)nvalid and (U)nknown verification conditions and running time of our tool.

Operation	LoC	V	I	U	Time (s)
`List.forall`	105	15	1	0	0.44
`List.exists`	20	7	0	0	0.17
`List.map`	60	6	4	0	0.31
`List.sort`	51	3	1	0	0.11
`List.flatMap`	48	8	0	0	0.24
`List.foldRight`	101	20	0	0	0.94
`CommutativeFold`	141	18	4	0	0.42
`ListOps`	111	17	0	0	0.33
`OptionMonad`	47	9	0	0	0.13
`DeMorganSets`	23	2	0	0	0.07
`AssocSets`	23	2	0	0	0.07
`SetOps`	16	0	1	0	0.04
`Closures`	50	4	0	0	0.20
`Continuations`	27	1	1	0	0.07
`Switch`	16	0	2	0	0.07
`Transformations`	49	3	1	0	0.63
`ParBalanceFold`	206	33	0	2	0.45
`FiniteQuantifiers`	39	1	0	0	157.00
Total	1082	149	15	2	161.69
Total (non-degenerate)	847	115	15	0	4.69

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 particularily 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.

All of the benchmarks in Table 1 make some use of higher-order functions. Our system generates a number of verifi-

[1] https://github.com/epfl-lara/leon

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 perfomed 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 10^3 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 enables 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

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

[2] C. Benzmüller and N. Sultana. LEO-II version 1.5. In *PxTP 2013*, volume 14 of *EPiC Series*, pages 2–10, 2013.

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

[4] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP*, 2010.

[5] P. Castéran and Y. Bertot. *Interactive theorem proving and program development. Coq'Art*. Springer Verlag, 2004.

[6] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-aided reasoning: an approach*. Kluwer Academic Publishers, 2000.

[7] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In Z. Shao and B. C. Pierce, editors, *POPL*, 2009.

[8] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL*, 2010.

[9] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A Higher-Order Relational Constraint Solver. Technical report, MIT-CSAIL-TR-2014-018, 2014. URL `http://hdl.handle.net/1721.1/89157`.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.

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

[12] C. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL*, 2011.

[13] A. Reynolds and V. Kuncak. Induction for SMT solvers. In *VMCAI*, 2015.

[14] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[15] O. Shivers. Control-flow analysis in scheme. In R. L. Wexelblat, editor, *PLDI*, pages 164–174. ACM, 1988. .

[16] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

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

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

[19] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.

[20] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *ICFP*, 2014.

Appendix: Proofs

List of complete proofs ommited when discussing counter-example completeness.

Theorem 2. *For $\eta \in H$ with $\eta : T$ for some $T \in H_{type}$ and $m_{\mathcal{H}} \models \mathcal{C}_{\mathcal{H}}(\eta)$, if $m_{\mathcal{H}}$ is interpretation-independent, then corresponding m_H is such that $\eta[m_H] \longrightarrow \mathbf{true}$.*

Proof. We will start by defining a helper function $\mathcal{C}_\wedge$ for $\eta_i \subseteq \eta$ and associated b_i where $\mathcal{C}_\wedge(\eta_i, b_i) = c \wedge e$ given $(c, e, \tau, \pi, \sigma) = \mathcal{C}(\eta_i, b_i)$. Note that $\mathcal{C}_\wedge(\eta_i, b_i)$ depends on all conjuncts generated in $\mathcal{C}$ for the pair (η_i, b_i).

Let us prove by induction that for $\eta_i \subseteq \eta$ with associated b_i, if $m_{\mathcal{H}} \models b_i$ then

$$m_{\mathcal{H}} \models \mathcal{C}_\wedge(\eta_i, b_i) \implies \eta_i[m_H] \longrightarrow \mathbf{true} \quad (4)$$

$$m_{\mathcal{H}} \models \neg\mathcal{C}_\wedge(\eta_i, b_i) \implies \eta_i[m_H] \longrightarrow \mathbf{false} \quad (5)$$

$$m_{\mathcal{H}} \models \mathcal{C}_\wedge(\eta_i, b_i) = \mathcal{L}(\lambda) \implies \eta_i[m_H] \longrightarrow \lambda \in H_\lambda \quad (6)$$

If $\eta_i \sim \langle Definition\rangle^* \; Expr$, then the induction step is trivial. The same holds for $\eta_i \sim g \in H_{val}$ and $\eta_i \sim v \in H_{var}$. If $\eta_i \sim \neg\eta_j \in H$, then the definition of $\mathcal{C}$ tells us that b_j associated to η_j is the same as b_i. Therefore, $m_{\mathcal{H}} \models \mathcal{C}_\wedge(\eta_j, b_j)$ implies both $\eta_j[m_H] \longrightarrow \mathbf{true}$ (by induction) and $m_{\mathcal{H}} \models \neg\mathcal{C}_\wedge(\eta_i, b_i)$, which gives us $\eta_i[m_H] \longrightarrow \mathbf{false}$. 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 $\eta_i \sim f(\eta_{A1}, ..., \eta_{An})$. 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 $\eta_i \sim v \in H_{var}$ and the hypothesis holds. In the second case, all sub-term b_j's are the same as b_i so the induction hypothesis holds for $\eta_{A1}, ..., \eta_{An}$. Let us augment models $m_{\mathcal{H}}$ and m_H to $m'_{\mathcal{H}}$ and m'_H by respectively adding bindings for $f_{arg,1}$ and $\mathcal{V}(f^n_{arg})$. We described unfolding equivalence in H, so assuming by symmetry that $m'_{\mathcal{H}} \models \mathcal{C}_\wedge(f_{body}, b_i)$, we have $f_{body}[m'_H] \longrightarrow \mathbf{true}$ and these observations imply both $m_{\mathcal{H}} \models I_f(\eta_i, b_i)$ and $\eta_i[m_H] \longrightarrow \mathbf{true}$. The $\eta_i \sim \eta_\lambda(\eta_{A1}, ..., \eta_{An})$ case is similar but when dealing with the unfolded case for (p, λ), we must also consider $\eta_\lambda[m_H] \longrightarrow \lambda_k$ where $\lambda_k \neq \lambda$. If this is the case, $m_{\mathcal{H}} \models \mathcal{L}(\lambda_k) \neq \mathcal{L}(\lambda)$ and therefore $m_{\mathcal{H}} \models \neg b_p$ from $I_\lambda(p, \lambda)$, so we fall back to the $\eta_i \sim v \in H_{var}$ case and preserve validity.

It remains to consider $\eta_i \sim \mathbf{if}(\eta_c)\, \eta_t \,\mathbf{else}\, \eta_e$. We can assume by symmetry that $m_{\mathcal{H}} \models \mathcal{C}_\wedge(\eta_c, b_c)$ and $m_{\mathcal{H}} \models \mathcal{C}_\wedge(\eta_t, b_t)$ and therefore $m_{\mathcal{H}} \models \mathcal{C}_\wedge(\eta_i, b_i)$. The definition of $\mathcal{C}$ again tells us that b_c associated to η_c is the same as b_i and the induction hypothesis implies that $\eta_c[m_H] \longrightarrow \mathbf{true}$. We also know given the definitions of e in the **if** case of $\mathcal{C}$ that $m_{\mathcal{H}} \models b_i \implies \mathcal{C}_\wedge(\eta_c, b_c) \implies b_t$ and therefore $m_{\mathcal{H}} \models b_t$. Again, the induction hypothesis tells us that $\eta_t[m_H] \longrightarrow \mathbf{true}$, and evaluation rules on H give us $\eta_i[m_P] \longrightarrow \mathbf{true}$.

To complete the proof, it suffices to note that $m_{\mathcal{H}} \models b_{start}$ and $m_{\mathcal{H}} \models \mathcal{C}_\wedge(\eta, b_{start})$ by construction and we therefore have $\eta[m_H] \longrightarrow \mathbf{true}$. □

Lemma 3. *If (c_i, τ_i) are built from $(c_0, \tau_0, \pi_0, \sigma_0) = \mathcal{C}_H(\eta \in H)$, then $\mathcal{B}_f(\tau_i) \implies \mathcal{B}_\tau(\tau_{all}, v_\tau, v_t)$ where (v_τ, v_t) depend on c_i and $\tau_{all} = \bigcup_{j=0}^{i} \tau_i$ is the union of all τ generated during unfolding.*

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

Lemma 4. *If c_i, π_i, σ_i and ψ_i are built from $(c_0, \tau_0, \pi_0, \sigma_0) = \mathcal{C}_H(\eta \in H)$ and $\psi_0 = Y(\pi_0, \sigma_0)$, then $\mathcal{B}_\lambda(\psi_i) \implies \mathcal{B}_{\pi,\sigma}(Y(\pi, \sigma), v_\pi, v_{p,\lambda})$ where $(v_\pi, v_{p,\lambda})$ depend on c_i.*

Proof. For any $v \in v_\pi$, we either have (1) an associated $(b, v, \Lambda, c_0, c^n) \in Q_\pi(q_{uf}(i))$ or (2) a $(b, v, c_0, c^n) \in \psi_i$. 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 $m_i \models c_0 = \mathcal{L}(\lambda)$ for one of these λ's or we have $m_i \models \neg b$, both options leading to interpretation-independence.
2. We have that $b \in \mathcal{B}_{left}(\psi_i)$ by definition and interpretation-independence is therefore also guaranteed.

□

Theorem 6. *For $\eta \in H$ with $\eta : \mathbf{Boolean}$ such that for all $f(e_1, ..., e_n) \subseteq \eta$, f is terminating and $\exists m.\eta[m] \longrightarrow \mathbf{true}$, there is a $u_i = (c_i, \tau_i, \pi_i, \sigma_i, \psi_i) \in \mathcal{U}(\eta)$ for which $\exists m_{\mathcal{H}}.m_{\mathcal{H}} \models c_i \wedge \mathcal{B}_f(\tau_i) \wedge \mathcal{B}_\lambda(\psi_i)$, and by converting $m_{\mathcal{H}}$ to m_H, we have $\eta[m_H] \longrightarrow \mathbf{true}$.*

Proof. We will begin by proving that for any b from τ_i or π_i, there exists a $j > i$ such that $b \notin \mathcal{B}(\tau_j, \psi_j)$ where $\mathcal{B}(\tau_j, \psi_j) = \mathcal{B}_f(\tau_j) \cup \mathcal{B}_\lambda(\psi_j)$. Let us argue by contradiction that there exists an infinite chain in $\mathcal{U}(\eta)$ of $u_l, u_{l+1}, u_{l+2}, ...$ with $0 \leq l$ such that $b \in \mathcal{B}(\tau_k, \psi_k)$ for all $k \geq l$.

We start by looking at which conditions are necessary for b to belong to $\mathcal{B}(\tau_{i+1}, \psi_{i+1})$ given $b \in \mathcal{B}(\tau_i, \psi_i)$. 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 $b \in \mathcal{B}(\tau_{i+1}, \psi_{i+1})$. Given the definitions of u_{i+1} and $\mathcal{C}$, we can easily see that

$$\begin{array}{rcl} E_b & ::= & f(e_1, ..., e_n) \\ & & \mid \lambda(e_1, ..., e_n) \\ & & \mid \mathbf{if}(E_b)\, e_1 \,\mathbf{else}\, e_2 \\ & & \mid \neg E_b \end{array}$$

We therefore have that an infinite chain of u_k where $b \in \mathcal{B}(\tau_k, \psi_k)$ 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 $\mathcal{B}_Q(i)$ clause. For $q = (b, v, \Lambda_i, c_0, c^n) \in Q_\pi(q_{unfol}(i))$, only Λ_i depends on i and it is increasing in i since any later q_j with $j > i$ such that $q \;{}_{\pi_q} q_j$ will imply $\Lambda_i \cup \{\mathcal{P}_\lambda(q_j)\} \subseteq \Lambda_{j+1}$. 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 $e = \mathbf{if}(\text{COND}[m])\,\text{THEN}[m] \,\mathbf{else}\, \text{ELSE}[m]$ and C of all nodes $e = \lambda(\text{E}_1, ..., \text{E}_n)$ where $\lambda \in H_\lambda$. Finally, let $I_{\mathcal{B}}$ be the union of all $\{b_t, b_e\}$ generated at corresponding points $\mathcal{C}(e, b)$ with $e \in I$ along with b_{start} and C_λ be the set of all caller λ's in C. Note that all functions encountered are terminating so I, $I_{\mathcal{B}}$, C and C_λ are finite.

We have just seen that for all $b \in I_{\mathcal{B}}$, there exists a $k_b \in \mathbb{N}$ such that $b \notin \mathcal{B}(\tau_k, \psi_k)$. Also, for all $\lambda \in C_\lambda$ there exists a $k_\lambda \in \mathbb{N}$ such that for all $(b, v, \Lambda_{k_\lambda}, c_0, c^n) \in Q_\pi(q_{uf}(k_\lambda))$, $\lambda \in \Lambda_{k_\lambda}$. Based on these, we can define

$$\hat{k} = max(\max_{b \in I_{\mathcal{B}}} k_b, \max_{\lambda \in C_\lambda} k_\lambda)$$

and let $m_{\mathcal{H}} \models c_{\hat{k}} \wedge \mathcal{B}_f(\tau_{\hat{k}}) \wedge \mathcal{B}_\lambda(\psi_{\hat{k}})$. Since m exists and all extra variables introduced by $\mathcal{C}$ are free, $m_{\mathcal{H}}$ is guaranteed to exist, and Theorems 2 and 5 ensure $\eta[m_H] \longrightarrow \mathbf{true}$ for m_H associated to $m_{\mathcal{H}}$. □