Automating Verification of Functional Programs with Quantified Invariants

Abstract

We present the foundations of a verifier for higher-order functional programs with generics and recursive algebraic data types. Our verifier supports finding sound proofs and counterexamples even in the presence of certain quantified invariants and recursive functions. Our approach uses the same language to describe programs and invariants and uses semantic criteria for establishing termination. Our implementation makes effective use of SMT solvers by encoding first-class functions and quantifiers into a quantifier-free fragment of first-order logic with theories. We are able to specify properties of datastructure operations involving higher-order functions with minimal annotation overhead and verify them with a high degree of automation. Our system is also effective at reporting counterexamples, even in the presence of first-order quantification.

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

This paper presents the foundations of a system for verifying and falsifying (possibly quantified) properties of purely functional higher-order programs. The language we consider, although syntactically a subset of Scala, has simple type-theoretic foundations (Hindley-Milner type system with algebraic data types), but enriches it with specifications. Specifications include function pre- and postconditions as well as invariants, and they can refer to recursive functions and quantifiers over first-order values. Our tool supports automated finding of proofs and counterexamples for specifications and also establishes termination of functions to ensures soundness of reasoning. We illustrate our system on a set of examples.

Figure 1 shows the specification of a purely functional binary search tree storing unbounded integers (in source code denoted BigInt but here rendered $\mathbb{Z}$ for brevity). The code shows the type definition as well as its functions content, contains, and insert. The data type invariant appears as a **require** clause in the constructor Node. While there are many ways to define invariants for simple binary search trees (including some that are executable), this one is closest to textbook description and it is very concise. The specification of methods in **ensuring** clauses also indicate that they behave as expected: contains corresponds to set membership and insert to insertion into a set. Our system verifies that all the specifications and invariants hold, that all pattern matching is complete, and that all functions terminate. The verification takes under 1.5 seconds, which is notably faster than executing the scalac compiler itself.

Figure 2 shows the use of functions stored inside case classes along with quantified invariants enforcing that stored functions satisfy the supplied conditions. Using such data type definition, it is possible to manipulate functions together with specifications and

```
sealed abstract class Tree {
  def content: Set[ℤ] = this match {
    case Leaf() ⇒ Set.empty[ℤ]
    case Node(l, v, r) ⇒ l.content ++ Set(v) ++ r.content
  }

  def insert(value: ℤ): Node = (this match {
    case Leaf() ⇒ Node(Leaf(), value, Leaf())
    case Node(l, v, r) ⇒
      if (v < value) Node(l, v, r insert value)
      else if (v > value) Node(l insert value, v, r)
      else Node(l, v, r)
  }) ensuring(_.content == content ++ Set(value))

  def contains(value: ℤ): Boolean = (this match {
    case Leaf() ⇒ false
    case Node(l, v, r) ⇒
      if (v == value) true
      else if (v < value) r contains value
      else l contains value
  }) ensuring(_ == (content contains value))
}

case class Leaf() extends Tree
case class Node(left: Tree, value: ℤ, right: Tree) extends Tree {
require(∀((x:ℤ) ⇒ (left.content contains x) ⟹ x < value) &&
        ∀((x:ℤ) ⇒ (right.content contains x) ⟹ value < x)) }
```

Figure 1. Binary Search Tree with invariant specified within the Node data type definition.

```
case class ⇝[A,B](f: A ⇒ B, pre: A ⇒ Boolean, ens: B ⇒ Boolean) {
  require(∀((x: A) ⇒ pre(x) ⟹ ens(f(x))))

  def apply(x: A): B = {
    require(pre(x))
    f(x)
  } ensuring(ens) }

def map[A, B](list: List[A], f: A ⇝ B): List[B] = {
  require(∀((x:A) ⇒ list.contains(x) ⟹ f.pre(x)))
  list match {
    case Cons(x, xs) ⇒ Cons[B](f(x), map(xs, f))
    case Nil() ⇒ Nil[B]()
  } } ensuring (res ⇒ ∀((x: B) ⇒ res.contains(x) ⟹ f.ens(x)))
```

Figure 2. Encoding of first-class functions with contracts and its use to define a map on lists.

ensure that the conditions are propagated through data structures, as shown on the example of the map function on lists. The verification and termination of this program also succeed quickly.

Figure 3 shows a definition of (a relaxed form of) multisets using first-class functions. Using this data type, we specify merge sort, ensuring that the multiset of elements is preserved (Section 9 also discusses a version with generic elements mapped to keys.) Verification also succeeds quickly, and the only function for which the system does not automatically prove termination is merge itself,

```
case class Bag[T](f: T ⇒ ℤ) {
  def union(that: Bag[T]) = Bag((x: T) ⇒ f(x) + that.f(x))
  def equals(that: Bag[T]) = ∀((x: T) ⇒ f(x) == that.f(x)) }

def content[T](list: List[T]): Bag[T] = list match {
  case Nil() ⇒ Bag(_ ⇒ 0)
  case Cons(x, xs) ⇒
    Bag(y ⇒ if (x == y) 1 else 0) union content(xs) }

def isSorted(list: List[ℤ]): Boolean = list match {
  case Cons(x1, tail @ Cons(x2, _)) ⇒ x1 ≤ x2 && isSorted(tail)
  case _ ⇒ true }

def merge(l1: List[ℤ], l2: List[ℤ]): List[ℤ] = {
  require(isSorted(l1) && isSorted(l2))
  (l1, l2) match {
    case (Cons(x, xs), Cons(y, ys)) ⇒
      if (x ≤ y) Cons(x, merge(xs, l2))
      else Cons(y, merge(l1, ys))
    case _ ⇒ l1 ++ l2 }
} ensuring { res ⇒ isSorted(res) &&
  (content(res) equals (content(l1) union content(l2))) }

def split(list: List[ℤ]): (List[ℤ], List[ℤ]) = (list match {
  case Cons(x1, Cons(x2, xs)) ⇒
    val (s1, s2) = split(xs)
    (Cons(x1, s1), Cons(x2, s2))
  case _ ⇒ (list, Nil[ℤ]())
}) ensuring { res ⇒
  (content(res._1) union content(res._2)) equals content(list) }

def mergeSort(list: List[ℤ]): List[ℤ] = (list match {
  case Cons(_, Cons(_, _)) ⇒
    val (s1, s2) = split(list)
    merge(mergeSort(s1), mergeSort(s2))
  case _ ⇒ list
}) ensuring { res ⇒ isSorted(res) &&
  (content(res) equals content(list)) }
```

Figure 3. Higher-order functional encoding of multisets and the specification of mergeSort with multiset content semantics.

```
def associative[A](f: (A,A) ⇒ A): Boolean =
  ∀((x: A, y: A, z: A) ⇒ f(f(x, y), z) == f(x, f(y, z)))

def commutative[A](f: (A,A) ⇒ A): Boolean =
  ∀((x: A, y: A) ⇒ f(x, y) == f(y, x))

def rotates[A](f: (A,A) ⇒ A): Boolean =
  ∀((x: A, y: A, z: A) ⇒ f(f(x, y), z) == f(f(y, z), x))

def assocComm[A](f: (A,A) ⇒ A): Boolean = {
  require(associative(f))
  commutative(f) }.holds

def commAssoc[A](f: (A,A) ⇒ A): Boolean = {
  require(commutative(f))
  associative(f) }.holds

def commRotateAssoc[A](f: (A,A) ⇒ A): Boolean = {
  require(commutative(f) && rotates(f))
  associative(f) }.holds
```

Figure 4. Exploring dependencies between algebraic properties with counterexamples and proofs.

because the decrease in list size that split produces is not obvious to the system.

What makes our tool useful is that it can also report counterexamples. This feature works best for quantifier-free safety properties, but also works in some interesting cases for quantified properties. If in Figure 1 we swap the ordering in the appropriate line of contains to become **else if** (v > value) r.contains(value), our system (given an appropriate command-line option) finds the counterexample; it also confirms that the counterexample is real by performing execution that soundly interprets *a priory* unbounded quantifiers to confirm that the returned tree satisfies the invariants.

Our system can also find counterexamples for termination: for a function defining an interpreter for lambda calculus (Section 9) our termination checker ends up considering a possibly looping path and then uses the SMT solver to find that the well-known term $(\lambda x.xx)(\lambda x.xx)$ makes the path constraint satisfiable.

Figure 4 shows how function values and quantifiers enable abstraction of algebraic properties. Our system enables users to then reason about relationships between these properties. The shorthand holds is an abbreviation for **ensuring**(_ == **true**), and claims that a function with Boolean return type always returns **true**, so it can be used to encode conjectures and theorems. Our system finds counterexamples (over a domain of 7 uninterpreted elements) for the conjectures assocCom and comAssoc showing that neither of the properties associativity and commutativity implies the other. Furthermore, it proves comRotateAssoc, showing that commutativity and "rotation" imply associativity. These counterexamples and proofs are also found in time below 1.5 seconds.

Contributions. We present a verifier for higher-order functional programs with the ability to automatically find proofs and counterexamples. Our verifier incorporates the following new techniques:

- Verification of first-class functions along arbitrary paths in the program even when these are contained within input datastructures. We discuss a meaningful representation for reported counterexamples and present a sound and executable notion of first-class function equality. Our procedure is complete for counterexamples.
- Termination checking in the presence of higher-order functions, recursion and etc. We present a termination prover that builds upon the verification techniques we presented in a sound and efficient way, that conservatively allows the use of inductive reasoning in many useful cases.
- Support for arbitrarily nested ADT invariants that significantly reduce the annotation burden of proofs and provide better specification mechanisms for first-class functions. The technique we present is complete for counterexample finding as well.
- First-order quantifier instantiation with support for sound model finding. We present an extension to MBQI that enables model finding even in cases where the clause set is not *stratified* and discuss certain decidability implications.
- We discuss how the different presented techniques integrate with each other and consider soundness of the unified procedure.

We have evaluated our verifier on a set of benchmarks covering a wide range of features.

2. Preliminaries

Defining the programs. We start by defining the purely functional subset of Scala (called PureScala) on which our transformation is defined. A PureScala program P consists of a set of $definition$ nonterminal expansions as well as an $expr$ expansion. An expression can be a primitive operation, conditional, pattern matching, application of named or of first-class function. Algebraic data types are written as simple forms of Scala case classes. (Figure 6 in the Appendix shows the syntax.) The set of definitions that are relevant to the given expression is defined by the expression itself and is therefore not explicitly considered hereafter. Note that we use the terms program and expression interchangeably for this reason.

The typing rules for PureScala are as expected for call by value simply typed lambda calculus with recursive sum and product typesi (called algebraic data types or ADTs hereafter). We generally

assume that considered programs and expressions are well-typed. We therefore associate to each expression E its type E.tpe. We implicitly assume that variables and first-class function definitions are each associated with some fixed type, providing the initial Γ context for E : E.tpe.

Given an algebraic data type (ADT) T, we associate to T a set of constructors $\{ \text{cons}_{\mathtt{T},1}, \ldots, \text{cons}_{\mathtt{T},n} \}$ such that each $\text{cons}_{\mathtt{T},i}$ is a m_i-ary function of type $\mathtt{T}_{i,1} \times \cdots \times \mathtt{T}_{i,m_i} \to \mathtt{T}$. Instances of type T therefore have the shape $\text{cons}_{\mathtt{T},i}(\text{E}_1, \ldots, \text{E}_{m_i})$. We omit the T index and write cons_i when the relevant ADT type is clear from context. Expressions within an ADT instance (*i.e.* ADT *fields*) are accessed through pattern match case deconstruction.

We do not distinguish between recursive and anonymous function definitions (unless explicitly stated otherwise) as all functions are considered first-class. We use $\mathtt{f}, \mathtt{g}, \ldots$ to refer to first-class function expressions. We further associate to function f the following variable sequences $\mathtt{f}.args$ corresponding to its arguments and $\mathtt{f}.closures$ corresponding to some deterministic ordering of the free variables in f. Finally, we have the expression $\mathtt{f}.body$ that corresponds to the body of f.

We define vars(P) to be the *free variables* within program/expression P. If an expression contains no free variables, we say it is *ground*. We define the notion of *value* as

$$value \quad ::= \quad \mathbf{true} \mid \mathbf{false} \mid \text{cons}_i(value, \ldots, value) \mid \text{ground function } \mathtt{f}$$

Given a program P, we call a mapping m that relates each variable $\mathtt{v} \in \text{vars}(\text{P})$ to some value V : v.tpe a set of *inputs to* P. Note that $m[\![\text{P}]\!]$ must be ground, and evaluation semantics follow those of the simply typed lambda calculus with conditionals. Further note that any well-typed program P with inputs m will either evaluate to some value or diverge, evaluation cannot get *stuck*.

Basic formula terminology. A *signature* Σ consists of a set of *sorts*, one of *function* symbols, and a set of *predicate* symbols. We view predicates as functions with boolean-sorted result. Each function symbol is associated with a sequence of argument sorts (whose length is the function's *arity*) and a result sort. Functions with zero arity are called *constants*. We let $vars(\phi)$ consist of the set of uninterpreted constants in ϕ. A *term* can be either a constant or a function application $f(t_1, \ldots, t_n)$ where f is a function symbol and t_1 to t_n are terms. Boolean-sorted terms are sometimes called *atoms*, and we generically use *clause* to refer to any production using atoms, disjunction ($\vee$), conjunction ($\wedge$), and their derivatives as these can trivially be translated into CNF form. We sometimes use set inclusion notation over finite sets to denote equality disjuncts. We are generally interested in satisfiability of clause sets where uninterpreted function symbols are taken as existentially quantified.

We are also interested in Σ-theories, namely sets of Σ-sentences that are closed under logical deduction. We assume each theory T is associated to a single sort σ_T, and can introduce a set of interpreted function symbols whose interpretations are restricted by the models for T. We use the $\simeq$ symbol for the interpreted equality symbol. We are specifically interested in the theory of algebraic data types where a specific ADT theory T is defined similarly to an ADT T in PureScala. Namely, we associate to T the following interpreted function symbols for $1 \leq i \leq n$: $isCons_{T,i}$, $cons_{T,i}$, and $field_{T,i,j}$ for $1 \leq j \leq m_i$, as well as the usual axiomatization. We again omit the T index when it is clear from context. We assume theories are *separate*. Namely, given a theory T with associated sort σ_T, given an interpreted function symbol f associated to T and the following interpretation $t = M[\![f(t_1, \ldots, t_n)]\!]$, either i) the sort of t is σ_T, or ii) $t \subseteq M[\![t_i]\!]$ for some $i : 1 \leq i \leq n$. Note that this assumption does not hold for all interesting theories supported by SMT solvers (*e.g.* consider sets with cardinality), but it does hold for the theories on which our transformation depends. Moreover, our definition of *theory separation* can be further augmented to allow t with *non-container* sort such as integer or real. Such considerations are out of the scope of this paper.

Finally, we say a term t is *ground* iff it contains no uninterpreted symbol. Given a ground term t and the equivalence class of ground terms $G_t = \{ t_2 \mid \models t \simeq t_2 \}$, we call the term $t_{min} \in G_t$ with smallest size a *value* (given multiple terms with same size, one is selected by convention). Term interpretation in a model is assumed to always produce a value.

3. Verification of Programs with Function Values

In this section we introduce mapping from PureScala program expressions (containing first-class function applications) to quantifier-free terms and formulas suitable for an SMT solver that has no knowledge of recursive or first-class functions. We build on this translation to present a procedure to prove that a boolean-result PureScala expressions cannot return false. Our procedure iteratively refines information about named and anonymous functions, generating a sequence of quantifier-free formulas that constitute increasingly precise program approximations.

3.1 Transformation from Expressions to Formulas

The main difference between source expressions and target formula terms is that functions in the target formulas are represented using values from an uninterpreted domain. The translation function explicitly represents increasing amount of information about the applications of such functions.

From types to sorts. We first introduce a translation function $\mathcal{S}$ from PureScala types T to sorts. When T is boolean or an algebraic data type, then $\mathcal{S}(\mathtt{T})$ is an appropriately chosen unique formula sort. For function types, however, we introduce a special sort $\sigma_{\mathtt{f}}$ with infinite cardinality. All function types map to $\sigma_{\mathtt{f}}$. The only relation symbol defined on $\sigma_{\mathtt{f}}$ is $\simeq$ and represents a notion of equality.

Our typing relation supports only invariant type parameters. We can therefore consider that instantiations of an ADT type T with different type arguments correspond to disjoint sorts. This consideration further extends to generic function definitions. We can thus ignore generic type parameters in the following by assuming (potentially infinite) sets of fully typed definitions. As these sets are explored incrementally through as-needed type substitutions in the access of $\mathtt{f}.body$, the relevant portions of the definition sets remain finite. Finally, we allow uninstantiated type parameters in our formulas and let $\mathcal{S}$ map to some (unique) sort with infinite cardinality and equality (cardinality limitations are outside the scope of this paper).

Program names to formula constants. We define an injection $\mathcal{V}$ from PureScala variables to formula constants. (Such constants are, effectively, existentially quantified because we will be checking satisfiability of generated formulas.) For a variable v: T the result of mapping $\mathcal{V}(\mathtt{v})$ has sort $\mathcal{S}(\mathtt{T})$.

Function applications. Our procedure relies on tracking two sets that are computed by the transformation. The first set, A, tracks function applications and consists of 4-tuples $(b, c, tpe, [a_1, \ldots, a_n])$ where b is a Σ-constant specifying the *path condition* under which this function application can take place, c is a Σ-term with sort $\sigma_{\mathtt{f}}$ specifying the caller, tpe is the PureScala function type corresponding to c, and $[a_1, \ldots, a_n]$ are Σ-terms argument to c (sometimes written a^n for brevity). We define the projection functions $(\cdot)_b$, $(\cdot)_c$ and $(\cdot)_{tpe}$ for $app \in A$. The second set, F, tracks $(b, f, \mathtt{f}, [c_1, \ldots, c_n])$ 4-tuples where b is a Σ-constant as in A, f is a (fresh) Σ-constant with sort $\sigma_{\mathtt{f}}$, f is a first-class PureScala function discovered during translation, and c^n are Σ-terms corresponding to

the closures of $\mathtt{f}$. We also define the projection functions $(\cdot)_b$, $(\cdot)_f$ and $(\cdot)_\mathtt{f}$ for $fun \in F$.

Transformation rules. The transformation $(b, \text{E}) \rightsquigarrow (e, \phi, A, F)$ takes as input the Σ-constant b describing the current path condition and the expression E in PureScala and returns a 4-tuple where e is a Σ-term, ϕ is a set of clauses, and A, F are defined above. We further define the following projective transformation $(b, \text{E}) \rightsquigarrow_\mathtt{t} e$. The transformation is defined inductively, so given $\{\text{SUB}_1, \ldots, \text{SUB}_n\}$ the set of all direct children of E (expression children can be inferred from the definition of PureScala's syntax in Figure 6) with $(b, \text{SUB}_i) \rightsquigarrow (\_, \phi_i, A_i, F_i)$, we generally have $\phi = \bigcup_i \phi_i$, $A = \bigcup_i A_i$ and $F = \bigcup_i F_i$. Therefore, it is useful for the sake of clarity to introduce the difference projection $(b, \text{E}) \twoheadrightarrow_\phi \phi - \bigcup_i \phi_i$. Note that the result of $\twoheadrightarrow_\phi$ completely defines the ϕ resulting from the transformation $\rightsquigarrow$ of (b, E) (and vice-versa). We similarly introduce $\twoheadrightarrow_A$ and $\twoheadrightarrow_F$. The rules of inferrence described below assume by convention that $(b, \text{E}) \twoheadrightarrow_\phi \emptyset$, $(b, \text{E}) \twoheadrightarrow_A \emptyset$, and $(b, \text{E}) \twoheadrightarrow_F \emptyset$ when left unspecified.

We start by defining the fairly straightforward transformation of PureScala expressions for which closely corresponding encodings exist in the formula domain.

$$(b, \mathtt{v}) \rightsquigarrow_\mathtt{t} \mathcal{V}(\mathtt{v}) \qquad (b, \mathbf{true}) \rightsquigarrow_\mathtt{t} true \qquad (b, \mathbf{false}) \rightsquigarrow_\mathtt{t} false$$

$$\frac{(b, \text{E}_1) \rightsquigarrow_\mathtt{t} e_1 \qquad \ldots \qquad (b, \text{E}_m) \rightsquigarrow_\mathtt{t} e_m}{(b, \text{cons}_i(\text{E}_1, \ldots, \text{E}_m)) \rightsquigarrow_\mathtt{t} Cons_i(e_1, \ldots, e_m)}$$

In order to handle general function definitions without resorting to quantifiers, we translate these to constants and perform incremental dynamic dispatch over call sites discovered during the transformation. This leads to the following rules for function handling

$$\textsc{Fun}\ \frac{f \text{ fresh with sort } \sigma_\mathtt{f} \qquad \mathtt{f} \text{ function}}{(b, \mathtt{f}) \rightsquigarrow_\mathtt{t} f \qquad (b, \mathtt{f}) \twoheadrightarrow_F \{(b, f, \mathtt{f}, \mathcal{V}(\mathtt{f}.closures))\}}$$

$$\textsc{App}\ \frac{(b, \text{C}) \rightsquigarrow_\mathtt{t} c \qquad (b, \text{E}_1) \rightsquigarrow_\mathtt{t} e_1 \qquad \ldots \qquad (b, \text{E}_n) \rightsquigarrow_\mathtt{t} e_n}{\begin{array}{l}(b, \text{C}(\text{E}_1, \ldots, \text{E}_n)) \rightsquigarrow_\mathtt{t} \text{dispatch}_{\text{C}.tpe}(c, e_1, \ldots, e_n) \\ (b, \text{C}(\text{E}_1, \ldots, \text{E}_n)) \twoheadrightarrow_A \{(b, c, \text{C}.tpe, [e_1, \ldots, e_n])\}\end{array}}$$

where the FUN rule accumulates potential dispatchees, and the APP rule both introduces well-typed dispatch calls through type-indexed function symbols and perform the necessary bookkeeping for future inlining.

We now define the transformation of **if**-expressions and discuss how they introduce sound path conditions. The transformation of a (b, E) pair where

$$\text{E} ::= \textbf{if}\ (\text{COND})\ \text{THEN}\ \textbf{else}\ \text{ELSE}$$

relies on two boolean-sorted fresh constants b_t and b_e, one for each new path condition (namely COND and $\neg$COND). We also need a fresh constant r with sort $\mathcal{S}(\mathtt{T})$ where $\text{E} : \mathtt{T}$. Given $(b, \text{COND}) \rightsquigarrow_\mathtt{t} c$, $(b_t, \text{THEN}) \rightsquigarrow_\mathtt{t} e$, and $(b_e, \text{ELSE}) \rightsquigarrow_\mathtt{t} e$, we can define the clausal encoding of the **if**-expression as

$$\begin{array}{rl}\phi_{branch} = \{\ (b \wedge c) & \iff b_t, \quad b_t \implies (r \simeq t), \\ (b \wedge \neg c) & \iff b_e, \quad b_e \implies (r \simeq e)\ \}.\end{array}$$

Note that computation of t and e relies on the constants b_t and b_e, such that the clauses (and other bookkeeping information) resulting from the transformation of THEN and ELSE depend respectively on b_t and b_e. Furthermore, the clauses in ϕ_{branch} ensure that the data-flow can only take a single branch of the conditional, thus ensuring b_t and b_e are sound path conditions. The definition of ϕ_{branch} naturally leads to the following inferrence rule

$$\textsc{If}\ \frac{\begin{array}{c}b_t, b_e, r \text{ fresh} \qquad (b, \text{COND}) \rightsquigarrow (c, \phi_c, A_c, F_c) \\ (b, \text{THEN}) \rightsquigarrow (t, \phi_t, A_t, F_t) \qquad (b, \text{ELSE}) \rightsquigarrow (e, \phi_e, A_e, F_e)\end{array}}{\begin{array}{l}(b, \text{E}) \rightsquigarrow (\ r, \\ \qquad \phi_c \cup \phi_t \cup \phi_e \cup \phi_{branch}, \\ \qquad A_c \cup A_t \cup A_e, \\ \qquad F_c \cup F_t \cup F_e\ )\end{array}}$$

Finally, encoding of pattern match expressions is similar to that of **if**-expressions and can be found in the appendix.

Properties of transformation. We will now state several relevant properties of the transformation $\rightsquigarrow$. In this next part, we will consider the expression E with inputs m such that $m[\![\text{E}]\!] \xrightarrow{*} \text{V}$. We further consider $(b, \text{E}) \rightsquigarrow (e, \phi_\text{E}, A_\text{E}, F_\text{E})$ and $(b, \text{V}) \rightsquigarrow (v, \phi_\text{V}, \_, \_)$ as given. We start by noting that the transformation $\rightsquigarrow$ corresponds to an under-approximation of the expression E with respect to evaluation semantics.

Lemma 1. *There exists model* $M \models b \wedge \phi_\text{E} \wedge \phi_\text{V} \wedge e \simeq v$.

Consider $\text{E}_i \subseteq \text{E}$ such that $(b_i, \text{E}_i) \rightsquigarrow_\mathtt{t} t_i$ occured during the transformation of E. Note that all $\text{E}_i \subseteq \text{E}$ that are not contained within an anonymous function definition are visited by $\rightsquigarrow$. The position of E_i uniquely defines t_i, and vice-versa. Furthermore, they each uniquely define b_i, leading to the well-founded definitions $C_t(\text{E}_i) = t_i$, $C_\text{E}(t_i) = \text{E}_i$, $C_b(\text{E}_i) = b_i$, and $C_b(t_i) = b_i$. Furthermore, if the position E_i is encountered during evaluation of $m[\![\text{E}]\!]$, there exists a unique value E_V corresponding to that position. We can therefore define $C_\text{V}(\text{E}_i) = \text{E}_\text{V}$ if E_i is encountered during evaluation, and $C_\text{V}(\text{E}_i) = \bot$ otherwise.

Proposition 2. *For* $M \models b \wedge \phi_\text{E} \wedge \phi_\text{V} \wedge e \simeq v$ *and* $\text{E}_i \subseteq \text{E}$, $M \models C_b(\text{E}_i)$ *iff* $C_\text{V}(\text{E}_i) \neq \bot$.

As our encoding of first-class functions introduces fresh constants, the set of models satisfying Lemma 1 will be underconstrained in general. We therefore introduce the following lemma that deals more specifically with these constants

Lemma 3. *There exists an* M *satisfying Lemma 1 such that for* $\text{E}_\text{C} \subseteq \text{E}$ *where* $\text{E}_\text{C} ::= \text{C}(\text{E}_1, \ldots, \text{E}_n)$ *and for* $fun \in F_\text{E}$, *if* $M \models C_b(\text{E}_\text{C}) \wedge (fun)_b$, *then* $M \models C_t(\text{C}) \simeq (fun)_f$ *iff* $C_\text{V}(\text{C}) = C_\text{V}(C_\text{E}((fun)_f))$.

3.2 Unfolding Function Applications

We discuss how our procedure relates the uninterpreted function applications introduced by the APP rule to the associated function definitions.

We start by considering a constant-program pair (b_P, P) with $(b_\text{P}, \text{P}) \rightsquigarrow (e_\text{P}, \phi_\text{P}, A_\text{P}, F_\text{P})$. Given $app \in A_\text{P}$ and $fun \in F_\text{P}$ where $app = (b_c, c, tpe, a^n)$, $fun = (b_f, f, \mathtt{f}, c^n)$, and $tpe = \mathtt{f}.tpe$, consider the transformation $(b_\mathtt{f}, \mathtt{f}.body) \rightsquigarrow (e_\mathtt{f}, \phi_\mathtt{f}, A_\mathtt{f}, F_\mathtt{f})$ where $b_\mathtt{f}$ is a fresh constant. Further consider the substitution

$$\theta_\mathtt{f} = \{\ \mathcal{V}(\mathtt{f}.args) \to a^n\ \}\{\ \mathcal{V}(\mathtt{f}.closures) \to c^n\ \}.$$

Note that the first portion of $\theta_\mathtt{f}$ is entailed by the evaluation rules on function applications and the second part corresponds to the semantics of $m[\![\text{P}]\!]$ for inputs m. Now, consider the clause

$$\text{dispatch}_{\mathtt{f}.tpe}(c, a_1, \ldots, a_n) \simeq \theta_\mathtt{f}[\![e_\mathtt{f}]\!]. \qquad \text{(disp)}$$

Under the conditions b_c, b_f and $c \simeq f$, the (disp) clause along with the substituted sets $\theta_\mathtt{f}[\![\phi_\mathtt{f}]\!]$, $\theta_\mathtt{f}[\![A_\mathtt{f}]\!]$, and $\theta_\mathtt{f}[\![F_\mathtt{f}]\!]$ corresponds to taking the transformation *after* having inlined the function call associated with app in P. We therefore define

$$\begin{array}{rl}\mathcal{I}(app, fun) = (\ \theta_\mathtt{f}[\![\phi_\mathtt{f}]\!] \cup \{\ b_\mathtt{f} & \iff (b_c \wedge b_f \wedge c \simeq f), \\ & b_\mathtt{f} \implies (\text{disp})\ \},\ \theta_\mathtt{f}[\![A_\mathtt{f}]\!],\ \theta_\mathtt{f}[\![F_\mathtt{f}]\!]\ )\end{array}$$

which enables us to progressively refine the transformation of P by introducing new clauses corresponding to inlinings of function applications in P. Note that we preserve the $b \implies c$ structure of our clause set. Furthermore, if fun doesn't correspond to C, then all new elements will be irrelevant as $b_{\mathtt{f}}$ will not hold.

Lemma 4. *Given inputs m with $m[\![\text{P}]\!] \xrightarrow{*} \text{V}$ and $(b_{\text{P}}, \text{V}) \rightsquigarrow_{\mathtt{t}} v_{\text{P}}$, given $\text{E}_{\text{C}} \subseteq \text{P}$ where $C_t(\text{E}_{\text{C}}) = \text{dispatch}_{tpe}(c, a_1, \ldots, a_n)$ and $(C_b(\text{E}_{\text{C}}), C_{\text{V}}(\text{E}_{\text{C}})) \rightsquigarrow_{\mathtt{t}} v_{\text{C}}$, there exists model M such that either*

1. $M \models b_{\text{P}} \wedge \phi_{\text{P}} \wedge e_{\text{P}} \simeq v_{\text{P}} \wedge \theta_{\mathtt{f}}[\![\phi_{\mathtt{f}}]\!] \wedge \theta_{\mathtt{f}}[\![e_{\mathtt{f}}]\!] \simeq v_{\text{C}}$, *or*
2. $M \models b_{\text{P}} \wedge \phi_{\text{P}} \wedge e_{\text{P}} \simeq v_{\text{P}} \wedge \neg b_{\mathtt{f}}$.

Based on these considerations and assuming P : Boolean, we define the sequence $U(\text{P}) = u_0, u_1, \ldots$ of triplets $u_i = (\phi_i, A_i, F_i)$ where $u_0 = (\phi_{\text{P}} \cup \{b_{\text{P}}, \neg e_{\text{P}}\}, A_{\text{P}}, F_{\text{P}})$ and u_i is defined given $app_i \in A_{i-1}$ and $fun_i \in F_{i-1}$ as $u_i = u_{i-1} \cup \mathcal{I}(app_i, fun_i)$. The sequence of u_i can therefore be seen as progressively inlining function applications inside the program. This definition leads to the following theorem

Theorem 5. *Given* P : Boolean *with ϕ_i from $U(\text{P})$, if $\phi_i \in$ Unsat, then no inputs m exist such that $m[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$.*

Given some expression E : Boolean, if there exist no inputs m such that $m[\![\text{E}]\!] \xrightarrow{*} \mathbf{false}$, we say E is *valid*. We are generally interested in this work in either showing validity (as is the case for Theorem 5), or reporting inputs for which the given expression evaluates to **false**.

4. Inductive Assumptions Modulo Termination

In practice, most interesting properties dealing with recursive functions will never have $\phi_i \in$ Unsat as the uninterpreted dispatch symbols offer too high a degree of freedom. It therefore becomes useful to introduce some notion of inductiveness into the procedure. Consider an expression POST : Boolean such that there exists a function application $\mathtt{f}(\mathtt{v}_1, \ldots, \mathtt{v}_n) \subseteq \text{POST}$ where $\mathtt{f}.closures = \emptyset$, $\text{vars}(\text{POST}) = \{\, \mathtt{v}_1, \ldots, \mathtt{v}_n \,\}$, and $\mathtt{v}_1, \ldots, \mathtt{v}_n$ are taken as quantified. We extend the procedure to produce sound proofs in the presence of assumption POST when f is terminating.

Given app_i, fun_i selected in $U(\text{P})$ where $(fun_i)_{\mathtt{f}} = \mathtt{f}$, consider $b_{\mathtt{f}}$ and $\theta_{\mathtt{f}}$ introduced by $\mathcal{I}(app_i, fun_i)$. Recall that $b_{\mathtt{f}}$ holds iff $(app_i)_b \wedge (fun_i)_b \wedge (app_i)_c \simeq (fun_i)_f$ and $\theta_{\mathtt{f}} = \{\, \mathcal{V}(\mathtt{f}.args) \mapsto a^n \,\}$ for a^n from app_i (the second part of $\theta_{\mathtt{f}}$ is omitted as $\mathtt{f}.closures = \emptyset$). Now consider the transformation $(b_{\mathtt{f}}, \text{POST}) \rightsquigarrow (p, \phi_p, A_p, F_p)$ and let

$$\mathcal{P}_{\text{POST}}(app_i, fun_i) = (\ \theta_{\mathtt{f}}[\![\phi_p]\!] \cup \{\, b_{\mathtt{f}} \implies \theta_{\mathtt{f}}[\![p]\!] \,\},\ \theta_{\mathtt{f}}[\![A_p]\!], \theta_{\mathtt{f}}[\![F_p]\!]\ ).$$

For simplicity, we extend $\mathcal{P}_{\text{POST}}$ to a total function by letting $\mathcal{P}_{\text{POST}}(app_i, fun_i) = (\emptyset, \emptyset, \emptyset)$ when $(fun_i)_{\mathtt{f}} \neq \mathtt{f}$. We then also extend the procedure $U(\text{P})$ to $U_{\text{POST}}(\text{P})$ such that u_i is now defined as $u_i = u_{i-1} \cup \mathcal{I}(app_i, fun_i) \cup \mathcal{P}_{\text{POST}}(app_i, fun_i)$.

We want to show that the result of Theorem 5 is preserved in the extended $U_{\text{POST}}(\text{P})$. Clearly, if there exist inputs m to POST such that $m[\![\text{POST}]\!] \xrightarrow{*} \mathbf{false}$, then Lemma 1 tells us that there exists $M \models b_{\mathtt{f}} \wedge \phi_p \wedge \neg p$. Consequently, there exists an app, fun pair where $\mathcal{P}_{\text{POST}}(app, fun) = (\phi, \_, \_)$ such that $\phi \in$ Unsat. As the existence of inputs m to POST *does not* imply the inexistence of inputs m_{P} to P such that $m_{\text{P}}[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$, such a case leads to unsoundness. In order to guard against this occurence, we must therefore show that there exist no such inputs m.

If we consider $U_{\text{POST}}(\text{POST})$ directly, our clause set ϕ_i will, at some point, contain both $\neg p$ from the initial transformation of POST and p from $\mathcal{P}_{\text{POST}}(app_i, fun_i)$, thus leading to $\phi \in$ Unsat even when inputs exist. We therefore consider the expression E_{POST} defined as

$$\text{E}_{\text{BODY}} ::= \{\, \mathtt{f}.args \mapsto \mathtt{v}^n \,\}[\![\mathtt{f}.body]\!]$$
$$\text{E}_{\text{POST}} ::= \{\, \mathtt{f}(\mathtt{v}_1, \ldots, \mathtt{v}_n) \mapsto \text{E}_{\text{BODY}} \,\}[\![\text{POST}]\!]$$

Lemma 6. *Given ϕ_i from $U_{\text{POST}}(\text{E}_{\text{POST}})$, if $\phi_i \in$ Unsat and* POST *terminates on all inputs, then there exist no inputs m such that $m[\![\text{POST}]\!] \xrightarrow{*} \mathbf{false}$.*

Showing validity of POST is interesting in its own right, however it is can be useful to consider arbitrary property P.

Theorem 7. *Given ϕ_i from $U_{\text{POST}}(\text{P})$, if $\phi_i \in$ Unsat,* POST *is valid, and* POST *terminate on all inputs, then there exist no inputs m such that $m[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$.*

Note that we can trivially extend the $U_{\text{POST}}(\cdot)$ procedure to a set of expressions $\text{POST}_1, \ldots, \text{POST}_m$ as all $\mathcal{P}_{\text{POST}_i}$ are independent. However, the termination condition on each POST_i becomes somewhat more complex in this setting. Indeed, divergence may appear due to inter-dependencies between the different assumptions even though each POST_i terminates for all inputs on its own. We associate to each POST_i its relevant function application $\mathtt{f}_i(\mathtt{v}_{i,1}, \ldots, \mathtt{v}_{i,n_i})$. Given inputs m, we inductively define the set S_m of all expressions relevant to termination

1. $m[\![\text{POST}_i]\!] \in S_m$,
2. if $\mathtt{f}_i(\text{E}_1, \ldots, \text{E}_n)$ was seen during evaluation of $\text{S} \in S_m$, then $\{\, \mathtt{v}_k \mapsto \text{V}_k \mid 1 \leq k \leq n \,\}[\![\text{POST}_i]\!] \in S_m$.

We say $\text{POST}_1, \ldots, \text{POST}_m$ are *post-terminating* for inputs m iff for all $\text{S} \in S_m$, evaluation of S is terminating. Note that for a single POST, post-termination for all inputs is equivalent to termination for all inputs. Let us now define $U_{\text{POST}}(\{\, \text{POST}_1, \ldots, \text{POST}_m \,\}, \text{P})$ in the obvious way using the relevant $\mathcal{P}_{\text{POST}_i}$ in the inductive definition of u_i. We can thus conclude with the following corollary

Corollary 8. *Given ϕ_i from $U_{\text{POST}}(\{\text{POST}_1, \ldots, \text{POST}_m\}, \text{P})$, if $\phi_i \in$ Unsat, $\text{POST}_1, \ldots, \text{POST}_m$ are valid, and $\text{POST}_1, \ldots, \text{POST}_m$ are post-terminating for all inputs, then there exist no inputs m such that $m[\![\text{P}]\!] \xrightarrow{*} \mathbf{true}$.*

5. Proving Termination

The soundness for inductive assumptions discussed in the previous section relies on termination of the expression for all inputs. Our verification framework therefore features an automated termination checker, which checks termination of each function definition. In principle, we could choose any method to ensure termination of computations, as long as we only use inductive reasoning for computations we have shown terminating. Note that it is also permissible to use properties previously proven by induction to establish termination of subsequent properties, which allows proving termination using non-trivial semantic arguments. Our process of verifying a program thus involves an interleaved sequence of verification and termination checks, as is common in interactive proof assistants.

Our termination checker requires that the recursion in type definitions is restricted to recursive reference of a type (with the same formal type parameters) in the field position. In particular, it disallows recursive definitions in which the type-level recursion involves function type constructor. The data types that the termination checker accepts can therefore be encoded in polymorphic lambda calculus [31], [28, Chapter 23]. Furthermore, function values stored in the arguments of the function whose termination is being checked are assumed to be terminating.

Given our definition of evaluation semantics, an infinite sequence of evaluation steps must contain an infinite subsequence of function application reduction steps. In the following, we distinguish between

anonymous functions of the shape $(\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \mathrm{E}$ (which are considered first-class) and named functions (which are not). We assume named functions are lifted to anonymous functions when appearing in first-class positions.

It is useful when constructing termination proofs to be able to consider some notion of *path* under which a given sub-expression can be evaluated. Namely, given expressions $\mathrm{E}, \mathrm{S} \subseteq \mathrm{E}$ and PATH, if for all inputs m we have $m[\![\mathrm{PATH}]\!] \longrightarrow^* \mathbf{true}$ iff $m[\![\mathrm{E}]\!]$ encounters the position of S during evaluation, then we call PATH the *path* to S in E. Given our formalization of PureScala, it is difficult to consider the notion of *path* in the presence of pattern match expressions. Let us therefore consider the extension to PureScala with the following expression non-terminals $\mathsf{isCons}_{\mathsf{T},i}(\cdot)$ and $\mathsf{field}_{\mathsf{T},i,j}(\cdot)$, as well as the corresponding trivial extensions to $\rightsquigarrow$. It is clear that given a program P, we can now transform it into an equivalent program (with respect to evaluation and $\rightsquigarrow$ transformation) that contains no pattern match expressions. We assume hereafter that all considered expressions have been thus transformed. We can now talk about the path under which an expression $\mathrm{E} \subseteq \mathrm{P}$ will be evaluated (noted path(E)), computed as the conjunction of all **if**-expression conditionals (respectively their negation depending on which branch contains E) that must hold for evaluation to reach E. For $\mathrm{E} \subseteq ((\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \mathrm{R}) \subseteq \mathrm{P}$, we relax our definition of *path* to avoid tracking the anonymous function to its application points and let path(E) contain the free variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$. Note that when these are taken as universally quantified, this constitutes a sound approximation of first-class function application points.

Our aim is to show that all evaluation traces must be finite by establishing that the set of named function application reduction steps in each trace must be finite. Given the expression E, we therefore consider the set

$$C(\mathrm{E}) = \{\, \mathtt{f}(\mathrm{A}_1, \ldots, \mathrm{A}_n) \mid \mathtt{f}(\mathrm{A}_1, \ldots, \mathrm{A}_n) \subseteq \mathrm{E} \,\}$$

and for each named function f in P, we let $C_\mathtt{f} = C(\mathtt{f}.body)$. As in the path case, given anonymous function $(\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \mathrm{E}$ defined within $\mathtt{f}.body$, all named function applications within E belong to $C_\mathtt{f}$ as well. We then define the following relation

$$\frac{\begin{array}{c}\mathrm{A}_1, \ldots, \mathrm{A}_n \text{ values} \qquad m_{free} \text{ inputs} \qquad \mathtt{g}(\mathrm{E}_1, \ldots, \mathrm{E}_m) \in C_\mathtt{f} \\ \theta = m_{free} \cup \{\, \mathtt{v}_k \mapsto \mathrm{A}_k \mid \mathtt{v}_k \in \mathtt{f}.args \,\} \\ \theta[\![\mathrm{path}(\mathtt{g}(\mathrm{E}_1, \ldots, \mathrm{E}_m))]\!]^* \longrightarrow \mathbf{true} \qquad \theta[\![\mathrm{E}_i]\!]^* \longrightarrow \mathrm{V}_i, 1 \leq i \leq m\end{array}}{(\mathtt{f}(\mathrm{A}_1, \ldots, \mathrm{A}_n), \mathtt{g}(\mathrm{V}_1, \ldots, \mathrm{V}_m)) \in R}$$

Note that we use the extra inputs m_{free} to cover the free variables in $\mathtt{g}(\mathrm{E}_1, \ldots, \mathrm{E}_m)$ that appear due to anonymous function arguments.

We assume without loss of generality that $\mathrm{P} ::= \mathtt{f}_0(\mathtt{v}_1, \ldots, \mathtt{v}_n)$ as any program P can be brought into such a form by introducing a fresh named function $\mathtt{f}_0$ such that $\mathtt{f}_0.body = \mathrm{P}$ and $\mathtt{f}_0.args = \mathrm{vars}(\mathrm{P})$. Note that as $\mathtt{f}_0$ is fresh, it will only appear once in any evaluation trace given inputs. Now for inputs m, consider an evaluation trace such that the named function application $\mathtt{g}(\mathrm{B}_1, \ldots, \mathrm{B}_m)$ is due to be reduced at some point in the trace, $\mathtt{g} \neq \mathtt{f}_0$ and $\mathrm{B}_1, \ldots, \mathrm{B}_m$ are values. As evaluation has reached this point, there must exist some named function application reduction $\mathtt{f}(\mathrm{A}_1, \ldots, \mathrm{A}_n)$ in the trace such that $(\mathtt{f}(\mathrm{A}_1, \ldots, \mathrm{A}_n), \mathtt{g}(\mathrm{B}_1, \ldots, \mathrm{B}_m)) \in R$. Indeed, even when $\mathtt{g}(\mathrm{B}_1, \ldots, \mathrm{B}_m)$ is contained within a closure which has arbitrarily travelled through the program, this closure must have been created at some point in the body of some $\mathtt{f}(\mathrm{A}_1, \ldots, \mathrm{A}_n)$. Furthermore, as we universally quantify over potential closure arguments, any concrete arguments given to the closure will have been considered during construction of R, hence the inclusion. We assume that input first-class functions are terminating and support the assumption by checking that all functions defined in program terminate. This leads to the following statement

Proposition 9. *Given inputs m, if the trace of $m[\![\mathtt{f}_0.body]\!]$ contains infinitely many named function application reduction steps, then there must exist an infinite sequence* $\mathrm{APP}_0, \mathrm{APP}_1, \ldots$ *such that* $\mathrm{APP}_0 ::= \mathtt{f}_0(m[\![\mathtt{f}_0.args]\!])$ *and* $(\mathrm{APP}_i, \mathrm{APP}_{i+1}) \in R$.

A well-understood and general technique used to show finiteness of named function application chains consists of selecting a *measure* from the domain of a well-founded relation and showing that it decreases in all possible chains. Let us consider the following measure that corresponds to the structural size of the provided argument. For each ADT type T in the program, let

$$\begin{array}{l}\mathbf{def}\ \mathsf{size}_\mathsf{T}(\mathsf{adt}\colon \mathsf{T}) \colon \mathbb{N} = \mathsf{adt}\ \mathbf{match}\ \{ \\ \quad \mathbf{case}\ \mathsf{cons}_1(\mathsf{v}_{1,1},\ldots,\mathsf{v}_{1,m_1}) \Rightarrow 1 + \textstyle\sum_j \mathsf{size}_{\mathsf{T}_{1,j}}(\mathsf{v}_{1,j}) \\ \quad \ldots \\ \quad \mathbf{case}\ \mathsf{cons}_n(\mathsf{v}_{n,1},\ldots,\mathsf{v}_{n,m_n}) \Rightarrow 1 + \textstyle\sum_j \mathsf{size}_{\mathsf{T}_{n,j}}(\mathsf{v}_{n,j})\ \}\end{array}$$

where $\mathsf{size}_{\mathsf{T}_2}(\mathtt{x}) = 0$ for non-ADT type T_2. Note that by adding type parameters to the size_T definitions, we can support generic ADT definitions with a finite number of function definitions. It is trivial to (manually) show that each size_T function is terminating. Given expression E, we write $\mathrm{size}_{\mathrm{E}.tpe}(\mathrm{E})$ as $\mathrm{size}(\mathrm{E})$ for brevity. Now given $\mathtt{g}(\mathrm{E}_1, \ldots, \mathrm{E}_n) \in C_\mathtt{f}$, consider the expression

$$\begin{array}{rl}\mathrm{DEC}(\mathtt{g}(\mathrm{E}_1, \ldots, \mathrm{E}_n)) ::= & \mathbf{if}\ (\mathrm{path}(\mathtt{g}(\mathrm{E}_1, \ldots, \mathrm{E}_n)))\ \{ \\ & \textstyle\sum_i \mathrm{size}(\mathrm{E}_i) < \sum_{\mathtt{v}_k \in \mathtt{f}.args} \mathrm{size}(\mathtt{v}_k) \\ & \}\ \mathbf{else}\ \{\ \mathbf{true}\ \}\end{array}$$

Let us assume that for all $\mathtt{f} \in \mathrm{P}$, $\mathrm{E} \in C_\mathtt{f}$ and inputs m we have $m[\![\mathrm{DEC}(\mathrm{E})]\!] \longrightarrow^* \mathbf{true}$. Now consider the sequence $\mathrm{APP}_0, \mathrm{APP}_1, \ldots$ where $\mathrm{APP}_0 ::= \mathtt{f}_0(m[\![\mathtt{f}_0.args]\!])$, $\mathrm{APP}_i ::= \mathtt{f}_i(\mathrm{A}_{i,1}, \ldots, \mathrm{A}_{i,n_i})$ and $(\mathrm{APP}_i, \mathrm{APP}_{i+1}) \in R$. Given $\sum_j \mathrm{size}(\mathrm{A}_{i,j}) \longrightarrow^* \mathrm{V}_i$, we can consider the sequence $\mathrm{V}_1 > \mathrm{V}_2 > \cdots \geq 0$. As no infinite strictly decreasing sequence can exist in the domain of a well-founded relation, Proposition 9 ensures that there is no trace of $m[\![\mathtt{f}_0.body]\!]$ in which an infinite number of named function application reduction steps occur. Furthermore, we also know that the maximal recursion depth for any named function is upper bounded by h computed as

$$\sum\nolimits_{\mathtt{v}_k \in \mathtt{f}_0.args} \mathrm{size}(m[\![\mathtt{v}_k]\!])^* \longrightarrow h.$$

Let us now consider the transformation of P such that all named functions are transformed into anonymous functions and recursive calls are inlined up to depth $h + 1$ (an arbitrary value can be chosen for the result of the deepest call). Clearly, evaluation of the transformed program must achieve the same result as evaluation of P for inputs m. As a result, we obtain program without recursion. Thanks to our assumptions that recursion does not involve recursive type constructors, we can encode data types as well into types of polymorphic lambda calculus. Therefore, the execution of the program reduces to a call-by-value evaluation in polymorphic lambda calculus [31], [28, Chapter 23], in which all evaluations terminate. Consequently, our check ensures termination of the execution. We obtain the following proposition.

Proposition 10. *If there exists no infinite sequence* $\mathrm{APP}_0, \mathrm{APP}_1, \ldots$ *such that* $\mathrm{APP}_0 ::= \mathtt{f}_0(m[\![\mathtt{f}_0.args]\!])$ *and* $(\mathrm{APP}_i, \mathrm{APP}_{i+1}) \in R$*, and all types in* P *are well-founded, then* P *terminates on inputs* m.

Verification-based termination. We discussed in previous sections various procedures for showing *validity* of PureScala programs, namely the absence of models such that the program evaluates to a certain value. We can thus employ the $U(\cdot)$ procedure described previously to construct termination proofs

Theorem 11. *If for each* $\mathrm{E} \in \bigcup_{\mathtt{f} \in \mathrm{P}} C_\mathtt{f}$ *there exists some* ϕ_i *from* $U(\mathrm{DEC}(\mathrm{E}))$ *such that* $\phi_i \in \mathsf{Unsat}$*, then* P *terminates for all inputs.*

Remember that soundness of the $U_{\mathrm{POST}}(\cdot)$ procedure for inductive assumptions requires termination for all inputs of the POST expression. We therefore cannot use this procedure when construction

termination proofs as long as we haven't established termination of POST. Further recall that when considering multiple inductive assumptions, the termination requirement becomes much stronger. Therefore, when considering termination of program P with a set of inductive assumptions $\text{POST}_1, \ldots, \text{POST}_m$, we extend the definition of $C_{\mathtt{f}}$ to also contain the set of calls stemming from each relevant POST_i. Formally, given $\mathtt{f}_i(\mathtt{v}_{i,1}, \ldots, \mathtt{v}_{i,n_i})$ associated to POST_i, we extend $C_{\mathtt{f}_i}$ such that

$$\frac{\theta_{args} = \{\, \mathtt{v}_{i,k} \mapsto \mathtt{u}_k \mid \mathtt{u}_k \in \mathtt{f}_i.args \,\} \quad \theta_{inline} = \{\, \mathtt{f}_i(\mathtt{v}_{i,1}, \ldots, \mathtt{v}_{i,n_i}) \mapsto \mathtt{f}_i.body \,\}}{C((\theta_{args} \cup \theta_{inline})[\![\text{POST}_i]\!]) \subseteq C_{\mathtt{f}_i}}$$

Note that this extension to $C_{\mathtt{f}_i}$ potentially introduces new pairs into R and relates it as follows with *post-termination*

Proposition 12. *If* $\text{POST}_1, \ldots, \text{POST}_m$ *is not post-terminating for inputs* m*, then there exist* $\mathtt{f}_i$ *associated to* POST_i*, and an infinite sequence* $\text{APP}_0, \text{APP}_1, \ldots$ *such that* $\text{APP}_0 ::= \mathtt{f}_i(m[\![\mathtt{f}_i.args]\!])$ *and* $(\text{APP}_i, \text{APP}_{i+1}) \in R$*.*

If we can show that for all $\mathtt{f} \in \text{P}$, $\text{E} \in C_{\mathtt{f}}$ (using the new definition), and inputs m we have $m[\![\text{DEC}(\text{E})]\!] \xrightarrow{*} \mathbf{true}$, then P must post-terminate on all inputs (no infinite decreasing and lower-bounded chain can exist). However, $U_{\text{POST}}(\cdot)$ cannot be used to verify the validity of $\text{DEC}(\text{E})$!

Let us consider the call-graph G_{P} such that $(\mathtt{f}, \mathtt{g}) \in G_{\text{P}}$ iff $\mathtt{f} \in \text{P}$ and $\mathtt{g}(\text{E}_1, \ldots, \text{E}_n) \in C_{\mathtt{f}}$. We let G_{P}^* be the transitive closure of G_{P}. Given $\mathtt{f} \in \text{P}$, we let $Calls(f) = \{\, \mathtt{g} \mid (\mathtt{f}, \mathtt{f}) \in G_{\text{P}}^* \,\}$ and $Posts(\mathtt{f}) = \{\, \text{POST}_i \mid (\mathtt{f}, \mathtt{f}_i) \in G_{\text{P}}^* \,\}$. If we can show that for all $\mathtt{g} \in Calls(\mathtt{f})$, $\text{E} \in C_{\mathtt{g}}$, and inputs m we have $m[\![\text{DEC}(\text{E})]\!] \xrightarrow{*} \mathbf{true}$, then $Posts(\mathtt{f})$ is post-terminating for all inputs. Indeed, if $(\mathtt{f}, \mathtt{h}) \notin G_{\text{P}}^*$, then $\mathtt{h}$ is entirely irrelevant to post-termination of $Posts(\mathtt{f})$. These considerations lead to the following statement

Theorem 13. *For* $\mathtt{f} \in \text{P}$*, if* $Posts(\mathtt{f})$ *is post-terminating for all inputs,* $Posts(\mathtt{f})$ *are valid, and for each* $\text{E} \in \bigcup_{g \in \text{P}} C_{\mathtt{g}}$ *there exists some* ϕ_i *from* $U_{\text{POST}}(Posts(\mathtt{f}), \text{DEC}(\text{E}))$ *such that* $\phi_i \in \mathit{Unsat}$*, then* P *terminates for all inputs.*

We can thus construct a termination checking procedure that leverages (independently verified) inductive assumptions in a sound way.

Non-termination. If termination proving fails, we consider the relation $R_{\mathtt{f}} \subseteq R$ such that $(\mathtt{f}(\text{A}_1, \ldots, \text{A}_n), \mathtt{g}(\text{B}_1, \ldots, \text{B}_m)) \in R_{\mathtt{f}}$ iff there exists $\mathtt{g}(\text{E}_1, \ldots, \text{E}_m) \subseteq \mathtt{f}.body$ such that i) there exists no $((\mathtt{x}_1, \ldots, \mathtt{x}_n) \Rightarrow \text{R}) \subseteq \mathtt{f}.body$ for which $\mathtt{g}(\text{E}_1, \ldots, \text{E}_m) \subseteq \text{R}$, and ii) given $\theta = \{\, \mathtt{v}_k \mapsto \text{A}_k \mid \mathtt{v}_k \in \mathtt{f}.args \,\}$ we have $\theta[\![\text{path}(\mathtt{g}(\text{E}_1, \ldots, \text{E}_m))]\!] \xrightarrow{*} \mathbf{true}$ and $\theta[\![\text{E}_k]\!] \xrightarrow{*} \text{B}_k$, $1 \leq k \leq m$. Intuitively, for all elements of $R_{\mathtt{f}}$, if an evaluation trace contains the first element of the pair, it must contain the second as well. If we can find a sequence $\text{APP}_1, \ldots, \text{APP}_m$ such that $(\text{APP}_i, \text{APP}_{i+1}) \in R_{\mathtt{f}}$ and $\text{APP}_1 = \text{APP}_m$, then for each APP_i, we have a set of arguments that cause non-terminating of the corresponding function. Given some call-graph cycle, the above conditions can be encoded as a satisfiability query and handled by our verification procedure (see below).

6. Counterexamples and Function Equivalence

We have discussed how our procedure can be used to soundly verify properties of PureScala programs, however, its real strength lies in reporting counterexamples to these properties when they exist. We show how our unfolding-based approach can be extended to provide incremental encodings of program over-approximations which correspond to concrete executions when satisfied.

Inputs and equality. Given $(\phi_i, A_i, F_i) \in U(\text{P})$ with $M \models \phi_i$, let us consider the conversion from M to a set of inputs m. Given a value term $t \in M$, we therefore need a means of translating it back to a PureScala expression. As the sort conversion function $\mathcal{S}$ is not bijective, we further provide the expected type of t. Additionally, as the interpretation of the function corresponding to a $\sigma_{\mathtt{f}}$-sorted term is given by the interpretation of the associated dispatch symbol, our translation depends on the model M. We inductively define the reverse transformation $\mathcal{C}_M$ as follows

$$\mathcal{C}_M(true, \mathsf{Boolean}) = \mathbf{true} \qquad \mathcal{C}_M(false, \mathsf{Boolean}) = \mathbf{false}$$

$$\frac{\text{E}_1 = \mathcal{C}_M(e_1, \mathsf{T}_{i,1}) \quad \ldots \quad \text{E}_m = \mathcal{C}_M(e_m, \mathsf{T}_{i,m})}{\mathcal{C}_M(Cons_i(e_1, \ldots, e_m), \mathsf{T}) = \text{cons}_i(\text{E}_1, \ldots, \text{E}_n)}$$

It remains to consider the translation when t has sort $\sigma_{\mathtt{f}}$ (and the expected type is therefore $\mathsf{T} = (\mathsf{T}_1, \ldots, \mathsf{T}_n) \Rightarrow \mathsf{T}_{res}$. We rely here on the set A_i of function applications witnessed up to this point, and consider the set $A_i(t) \subseteq A_i$ such that for $app \in A_i(t)$, we have $M \models (app)_b$, $M[\![(app)_c]\!] = t$, and $(app)_{tpe} = \mathsf{T}$. We also assume that we can construct some arbitrary value V_{res} of type T_{res} (note that otherwise, no terminating function exists for T). Given $A_i(t) = \{\, app_1, \ldots, app_m \,\}$ where each $app_j = (b_j, c_j, tpe, a_j^n)$, we define

$$\frac{\begin{array}{c}\text{E}_{j,k} = \mathcal{C}_M(M[\![a_{j,k}]\!], \mathsf{T}_k), 1 \leq j \leq m, 1 \leq k \leq n \\ \text{R}_j = \mathcal{C}_M(M[\![\text{dispatch}_{tpe}(c_j, a_{j,1}, \ldots, a_{j,n})]\!], \mathsf{T}_{res}), 1 \leq j \leq m\end{array}}{\begin{array}{l}\mathcal{C}_M(t, \mathsf{T}) = (\mathtt{v}_1 : \mathsf{T}_1, \ldots, \mathtt{v}_n : \mathsf{T}_n) \Rightarrow \\ \quad \mathbf{if}\ (\mathtt{v}_1 == \text{E}_{1,1}\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \mathtt{v}_n == \text{E}_{1,n})\ \text{R}_1 \\ \qquad\qquad \vdots \\ \quad \mathbf{else\ if}\ (\mathtt{v}_1 == \text{E}_{m,1}\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \mathtt{v}_n == \text{E}_{m,n})\ \text{R}_m \\ \quad \mathbf{else}\ \text{V}_{res}\end{array}}$$

It is often interesting given a variable $\mathtt{v}$ to consider $\mathcal{C}_M(\mathcal{V}(\mathtt{v}), \mathtt{v}.tpe)$ which we write $\mathcal{C}_M(\mathtt{v})$ for short. We also write $\mathcal{C}_M(\text{vars}(\text{P}))$ to describe the inputs $\{\, \mathtt{v}_k \mapsto \mathcal{C}_M(\mathtt{v}_k) \mid \mathtt{v}_k \in \text{vars}(\text{P}) \,\}$.

Note that the above definition of function-typed term extraction imposes a strong constraint on the kind of first-class functions considered as inputs to the program. However, by disallowing the equality predicate between function-typed expressions in our programs, the above becomes a sound representation of (terminating) functions. Not that our procedure doesn't consider non-terminating functions in program inputs and will not be able to construct V_{res} when no terminating input exists. However, we consider such cases as pathological and these can be identified (and reported) based on the program type definitions.

One can define the `&&` operator through an **if**-expression. Furthermore, meaningful equality predicates exist both in the theory of boolean algebra ($\iff$) and that of ADTs. However, although we do ensure the $\simeq$ predicate is defined on terms of sort $\sigma_{\mathtt{f}}$, our encoding of first-class functions into fresh constants doesn't ensure a sound congruence relation. Furthermore, equality between functions is a problematic notion in itself. Indeed, full functional equality (in the HOL sense) is undecidable and would therefore make no sense in our operational semantics. Conversely, reference equality would break purity and would require explicit modelling of the heap.

As mentionned previously, syntactic equality of functions can lead to sound models given the existence of inputs. Such a definition of equality can furthermore be efficiently implemented through techniques such as hash-consing. Finally, one can define a corresponding encoding into the formula domain. We start by inductively defining the notion of *simple* PureScala expressions as

$$\begin{array}{rl} simple ::= & \mathtt{v} \mid \mathbf{true} \mid \mathbf{false} \mid \text{cons}_i(simple, \ldots, simple) \\ & \mid \text{function}\ \mathtt{f}\end{array}$$

Note that given a *simple* expression E and a mapping θ from free variables in E to PureScala values, $\theta[\![\text{E}]\!]$ is a value. Now consider the set F obtained from some number of applications

of the FUN rule defined in $\rightsquigarrow$. Given $fun_1, fun_2 \in F$ where $fun_j = (b_j, f_j, \mathtt{f}_j, c_j^n), j \in \{1,2\}$, consider the closure substitution $\theta_j = \{\, \mathcal{V}(\mathtt{f}_j.closures) \mapsto c^n \,\}$. We then inductively define the $eq(\mathrm{E}_1, \mathrm{E}_2)$ predicate as follows

1. if for $j \in \{1,2\}$, E_j is *simple* and $\mathrm{vars}(\mathrm{E}_j) \subseteq \mathtt{f}_j.closures$,
$$eq(\mathrm{E}_1, \mathrm{E}_2) \iff \theta_1[\![C_t(\mathrm{E}_1)]\!] \simeq \theta_2[\![C_t(\mathrm{E}_2)]\!];$$
2. else if for $j \in \{1,2\}$, $\mathrm{E}_j = \mathtt{v}_j$, then $eq(\mathrm{E}_1, \mathrm{E}_2) \iff \mathtt{v}_1 = \mathtt{v}_2$;
3. else if E_1 and E_2 are produced by the same non-terminal in the PureScala grammar, then given their respective children $\mathrm{E}_{1,1}, \ldots, \mathrm{E}_{1,n}$ and $\mathrm{E}_{2,1}, \ldots, \mathrm{E}_{2,n}$,
$$eq(\mathrm{E}_1, \mathrm{E}_2) \iff eq(\mathrm{E}_{1,1}, \mathrm{E}_{2,1}) \wedge \cdots \wedge eq(\mathrm{E}_{1,n}, \mathrm{E}_{2,n});$$
4. otherwise, $\neg eq(\mathrm{E}_1, \mathrm{E}_2)$ must hold.

We define the $\overset{?}{\sim}$ predicate such that $(f_1, \mathtt{f}_1, c_1^n) \overset{?}{\sim} (f_2, \mathtt{f}_2, c_2^n)$ corresponds to the clause given by $f_1 \simeq f_2$ iff $eq(\mathtt{f}_1, \mathtt{f}_2)$. We further extend the definition to $fun_1 \overset{?}{\sim} fun_2$ (b_1 and b_2 are irrelevant). We can now state the following proposition that follows from the freshness of f_1 and f_2

Proposition 14. *For $fun_1, fun_2 \in F$, there must exist some model M such that $M \models fun_1 \overset{?}{\sim} fun_2$.*

Note that in case No. 1 of the definition of eq, we refer to the associated term $C_t(\mathrm{E}_j)$ where E_j *has not yet* been transformed. However, as E_j does not depend on $\mathtt{f}_j.args$, θ_j is equivalent with $\theta_{\mathtt{f}}$ defined in $\mathcal{I}(app, fun)$ for such expressions, and these can be computed ahead of time. Let us now consider the transformation $(b, \mathrm{E}_j) \rightsquigarrow (e_j, \phi_j, A_j, F_j)$ for some given b. Our definition of *simple* expressions ensures that e_j doesn't depend on b, and both ϕ_j and A_j are empty. Further note that the $\overset{?}{\sim}$ predicate between members of $\theta_j[\![F_j]\!]$ doesn't depend on b and its choice is therefore irrelevant in our use cases.

Given $x, y \in F$, let $F_{eq}(x, y)$ be the union of the $\theta_j[\![F_j]\!]$ sets corresponding to each E_j encountered during computation of $x \overset{?}{\sim} y$ along with x and y themselves. Further let $F^*_{eq}(x, y)$ be the fixpoint where

1. $F_{eq}(x, y) \subseteq F^*_{eq}(x, y)$, and
2. given $x', y' \in F^*_{eq}(x, y)$, $F_{eq}(x', y') \subseteq F^*_{eq}(x, y)$.

One can see that F^*_{eq} must be finite as there can only exist finitely many lambdas within a program. Given a set F, we can thus define the clause set

$$EQ_{\mathtt{f}}(F) = \{\, x' \overset{?}{\sim} y' \mid x, y \in F \wedge x', y' \in F^*_{eq}(x, y) \,\}$$

and extend Proposition 14 to the set F, again ensured by freshness

Proposition 15. *There must exist some model $M \models EQ_{\mathtt{f}}(F)$.*

Lemma 16. *For $fun_1, fun_2 \in F$ where $fun_j = (\_, f_j, \mathtt{f}_j, \_)$ and $\mathtt{f}_j$ is ground for $j \in \{1,2\}$, we have $\mathtt{f}_1 = \mathtt{f}_2$ iff some model $M \models f_1 \simeq f_2 \wedge EQ_{\mathtt{f}}(F)$ exists.*

Enumerating relevant functions. We have until now only discussed how a sound congruence relation can be enforced on the tuples fun_1, fun_2 produced by $\rightsquigarrow$, yet no such tuples exist in F for first-class functions that are *free* in the program. As our programs cannot refer to function equality, it is conservative to consider that the sets of input functions and functions defined within the program do not intersect. Indeed, purity cannot be broken by this constraint. Furthermore, since our definition of function equality depends on the syntax of the function, function-typed input extraction $\mathcal{C}_M$ can be easily extended to ensure that if $M[\![t_1]\!] \neq M[\![t_2]\!]$ for two $\sigma_{\mathtt{f}}$ sorted terms t_1, t_2, then $\mathcal{C}_M(t_1, \mathtt{T}) \neq \mathcal{C}_M(t_2, \mathtt{T})$ by adding redundant **if**-expressions to the resulting first-class function bodies. It is therefore safe to assert that $f_1 \not\simeq f_2$ if at least one of f_1, f_2 is free in the program. This observation enables us to avoid explicit modelling of the value associated to each free function in the program.

It remains to discuss how to obtain an exhaustive enumeration of the free functions in P. We introduce here the notion of *sort enumeration*. Given a sort σ, its enumeration consists of the sequence $t_0, t_1, \ldots$ of terms with sort σ such that the following conditions are satifsied:

Completeness: for term t of sort σ, there exists $j \in \mathbb{N}$ and some model M such that $M \models t \simeq t_j$.

$\sigma_{\mathtt{f}}$-exhaustiveness: for any value term t of sort σ with subterm $s \subset t$ of sort $\sigma_{\mathtt{f}}$, there exists $j \in \mathbb{N}$, $s_j \subset t_j$ and some model M such that $M \models t \simeq t_j$ and $M \models s \simeq s_j$.

Unicity: for t_i, t_j in the enumeration such that $i \neq j$ and for model M, we must have $M \models t_i \not\simeq t_j$.

It is clear that an enumeration of all value terms of sort σ will satisfy the previous conditions. The proposed definition is more general to enable useful optimizations (consider for example the constant i of integer sort which is a valid enumeration of integers).

Let us now consider the formula ϕ with $v \in vars(\phi)$ where v has sort σ and $t_0, t_1, \ldots$ is an enumeration of σ. Consider the set of terms $E_i(v) = \{s \mid s \subseteq t_j, j \leq i \wedge s \text{ has sort } \sigma_{\mathtt{f}}\}$ and the clause $\eta_i(v) = v \in \{\, t_j \mid j \leq i \,\}$. The completeness condition of sort enumerators ensures satisfiability is preserved and gives us the following proposition:

Proposition 17. *If $M \models \phi$ exists, then there exists $j \in \mathbb{N}$ and model M_j such that $M_j \models \phi \cup \eta_j(v)$.*

More importantly, sort enumerators give us the means to consider the set of all relevant functions, leading to the following lemma:

Lemma 18. *Given a model $M \models \eta_i(v)$, there is no term $s \subseteq M[\![v]\!]$ with sort $\sigma_{\mathtt{f}}$ such that $M \models s \notin E_i(v)$.*

When considering multiple sort enumerations jointly, we require that the sets of constants introduced by each enumeration be disjoint. This property can be trivially ensured through freshening.

Consider the set A obtained through some number of applications of the APP rule defined in $\rightsquigarrow$, and V the set of constants obtained by transforming the free variables in the program such that constants in A, F and $EQ_{\mathtt{f}}(F)$ either belong to V or were introduced by the $\rightsquigarrow$ transformation. As the sort $\sigma_{\mathtt{f}}$ is not truly related to a theory of first-class functions, we must consider function terms contained *within* free functions. Indeed, free function application results that contain terms with sort $\sigma_{\mathtt{f}}$ are not considered during our enumeration and therefore won't belong to any $E_i(v)$. Now consider $app \in A$ where $app = (b, c, tpe, a^n)$: if the clause $c \in E_i(v)$ holds, then we must enumerate the sort of $\mathrm{dispatch}_{tpe}(c, a_1, \ldots, a_n)$ (noted $\mathrm{disp}(app)$ hereafter) as it may contain further relevant functions. We can therefore define the set of all potentially relevant function terms as

$$E_{all,i}(V, A) = \bigcup\nolimits_{v \in V} E_i(v) \cup \bigcup\nolimits_{app \in A} E_i(\mathrm{disp}(app)).$$

Remember that we want to enforce distinctness with respect to the function tuples in F, thus leading to the definition

$$EQ_{f,i}(V, A, F) = \{\, f \not\simeq (fun)_f \mid f \in E_{all,i}(V, A), fun \in F \,\}.$$

Proposition 19. *There must exist model $M \models EQ_{f,i}(V, A, F)$.*

We have discussed how $\eta_i(v)$ will bind the terms in $E_i(v)$ to v, however we cannot assert $\eta_i(\mathrm{disp}(app))$ directly as $\mathrm{disp}(app)$ is only free when $(app)_c \in E_{all,i}(V, A)$ holds. We therefore want a *conditional* binding for $E_i(\mathrm{disp}(app))$, given by the clause set

$\eta_{A,i}(V, A)$ defined as follows

$$\text{conditionally}_i(app, S) = (app_c) \in S \implies \eta_i(\text{disp}(app))$$
$$\eta_{A,i}(V, A) = \{\, \text{conditionally}_i(app, E_{all,i}(V, A)) \mid app \in A \,\}.$$

Finally, given $(\phi_i, A_i, F_i) \in U(\text{P})$ and $V = \{\, \mathcal{V}(\text{v}) \mid \text{v} \in \text{vars}(\text{P}) \,\}$, we define the following clause set

$$\psi_i = EQ_{\texttt{f}}(F_i) \cup EQ_{f,i}(V, A_i, F_i) \cup \eta_i(V) \cup \eta_{A,i}(V, A_i).$$

Over-approximating the program. Given the uninterpreted nature of the various dispatch_{tpe} function symbols, we want to consider an over-approximation of the program where only paths for which all relevant inlinings have taken place are allowed. We define $D_i = \{\, (app_j, fun_j) \mid j \leq i \,\}$, the set of all application-function pairs that have been considered (*i.e.* dispatched) up to this point. For $app \in A_i$, we define the set of functions that have been considered as dispatch targets for app

$$T_i(app) = \{\, (fun_j)_f \mid (app_j, fun_j) \in D_i, app_j = app \,\}.$$

Now consider the case where the caller $(app)_c$ is in fact *free* in the program (*i.e.* comes from a free variable). No dispatch for app is necessary as the interpretation for $(app)_c$ defined by the reverse transformation $\mathcal{C}_M$ corresponds to the semantics given by dispatch. We can therefore define the set of handled callers for app

$$T_{all,i}(app) = T_i(app) \cup E_{all,i}.$$

Remember that each clause in ϕ_i is of the shape $b \implies c$ for some constant b. If $(app)_c \notin T_{all,i}(app)$ holds, then all parts of the program that depend on the result of the function application associated to app must be disallowed as the relevant function inlining hasn't taken place yet. This condition extends to each $app \in A_i$ and leads to the clause set

$$\rho_i = \{\, (app)_c \notin T_{all,i}(app) \implies \neg(app)_b \mid app \in A_i \,\}.$$

We assume a fair selection strategy of (app_i, fun_i) at each inlining step, so for any $app \in A_i$, $fun \in F_i$ there exists $j \in \mathbb{N}$ such that $app = app_j$ and $fun = fun_j$. We can thus state the following:

Theorem 20. *Given a model $M \models \phi_i \cup \psi_i \cup \rho_i$, we have $\mathcal{C}_M(\text{vars}(\text{P}))[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$.*

Theorem 21. *If inputs m exist such that $m[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$, then there exists $i \in \mathbb{N}$ and model M such that $M \models \phi_i \cup \psi_i \cup \rho_i$.*

Corollary 22. *For inductive assumption* POST *where* POST *is valid and terminates on all inputs, given $(\phi_i, A_i, F_i) \in U_{\text{POST}}(\text{P})$ with associated ψ_i, ρ_i and model $M \models \phi_i \cup \psi_i \cup \rho_i$, we have $\mathcal{C}_M(vars(\text{P}))[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$.*

Conversely, if inputs m exist such that $m[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$, there exists $j \in \mathbb{N}$ and model M such that given $(\phi_j, A_j, F_j) \in U_{\text{POST}}(\text{P})$ with associated ψ_j, ρ_j, we have $M \models \phi_j \cup \psi_j \cup \rho_j$.

7. Algebraic Data Type Invariants

Given an ADT type T, we define an algebraic data type invariant as a PureScala expression INV with a single free variable x: T that is universally quantified. Semantically, the ADT invariant implies that for any *valid* instance E_{T} of type T, we must have $\{\, \text{x} \mapsto \text{E}_{\mathsf{T}} \,\}[\![\text{INV}]\!] \xrightarrow{*} \textbf{true}$. We describe how the concept of sort enumeration can be further leveraged to support finding models that agree with ADT invariants.

It is interesting to note that ADT invariants do not increase the expressivity of contracts for first-order functions (though they do decrease the annotation burden). However, they enable some level of specification for first-class functions by leveraging the type system. There exists a clear relation between our ADT invariants and refinement types. However, as our refinement is *named*, we need only consider T-typed instance construction points to ensure a valid

$$\frac{}{\textbf{true} \triangleright \text{INV}} \qquad \frac{}{\textbf{false} \triangleright \text{INV}}$$

$$\frac{\text{E}_1 \triangleright \text{INV} \quad \ldots \quad \text{E}_m \triangleright \text{INV} \quad \text{cons}_i(\text{E}_1, \ldots, \text{E}_m) : \mathsf{T} \quad \{\, \text{x} \mapsto \text{cons}_i(\text{E}_1, \ldots, \text{E}_m) \,\}[\![\text{INV}]\!]^* \longrightarrow \textbf{true}}{\text{cons}_i(\text{E}_1, \ldots, \text{E}_m) \triangleright \text{INV}}$$

$$\frac{\text{E}_1 \triangleright \text{INV} \quad \ldots \quad \text{E}_m \triangleright \text{INV} \quad \text{cons}_i(\text{E}_1, \ldots, \text{E}_m) : \mathsf{T}_2, \mathsf{T}_2 \neq \mathsf{T}}{\text{cons}_i(\text{E}_1, \ldots, \text{E}_m) \triangleright \text{INV}}$$

$$\frac{\texttt{f} : (\mathsf{T}_1, \ldots, \mathsf{T}_n) \Rightarrow \mathsf{T}_{res} \quad \forall \text{E}_1 : \mathsf{T}_1, \ldots, \text{E}_n : \mathsf{T}_n, \texttt{f}(\text{E}_1, \ldots, \text{E}_n)^* \longrightarrow \text{V} \triangleright \text{INV}}{\texttt{f} \triangleright \text{INV}}$$

Figure 5. Inferrence rules for the $\text{E} \triangleright \text{INV}$ relation.

program typing. We therefore generate and prove corresponding verification conditions and show termination for all inputs of the given INV expression, thus intuitively ensuring both progress and preservation in the corresponing refinement type system.

Unlike relevant function enumeration, our notion of ADT invariants is not based on some need to relate terms resulting from the transformation $\rightsquigarrow$ but on the expressions themselves. It is therefore not so clear what the set of relevant instances of T actually consists of. We clarify the notion through a *holds* relation on PureScala values where $\text{E} \triangleright \text{INV}$ is defined in Figure 5. Given a set of inputs m to a program P returned by $\mathcal{C}_M$, we say INV *holds on* m iff for all $\text{v} \in \text{vars}(\text{P})$, we have $m[\![\text{v}]\!] \triangleright \text{INV}$.

Let us consider the formula ϕ with $v \in vars(\phi)$ where v has sort σ. For any term t of sort σ, we are naturally interested in all subterms $s \subseteq t$ of sort σ_{T} where $\sigma_{\mathsf{T}} = \mathcal{S}(\mathsf{T})$. We therefore extend the definition of sort enumeration $t_0, t_1, \ldots$ for sort σ by adapting the $\sigma_{\texttt{f}}$-exhaustiveness condition to the σ_{T} sort:

σ_{T}-exhaustiveness: for any value term t of sort σ with subterm $s \subset t$ of sort σ_{T}, there exists $j \in \mathbb{N}$, $s_j \subset t_j$ and some model M such that $M \models t \simeq t_j$ and $M \models s \simeq s_j$.

We define the set $S_i(v) = \{\, s \mid s \subseteq t_j, j \leq i \wedge s \text{ has sort } \sigma_{\mathsf{T}} \,\}$ by analogy with $E_i(v)$ of all subterms in $t_0, \ldots, t_i$ that have sort σ_{T}. Note that $\eta_i(v)$ ensures terms in $S_i(v)$ are bound as well as those in $E_i(v)$. Proposition 17 trivially extends to enumerations with σ_{T}-exhaustiveness and we can further state the following lemma:

Lemma 23. *Given a model $M \models \eta_i(v)$, there is no term $s \subseteq M[\![v]\!]$ with sort σ_{T} such that $M \models s \notin S_i(v)$.*

As with function applications, the definition of ADT invariants leads us to consider $(b_{\mathsf{T}}, \text{INV}) \rightsquigarrow (inv, \phi_{\mathsf{T}}, A_{\mathsf{T}}, F_{\mathsf{T}})$ where b_{T} is a fresh constant. Given the subterm $s \in S_i(v)$ that comes from enumeration term t_j, we define the substitution $\theta_s = \{\, \mathcal{V}(\text{x}) \mapsto s \,\}$. Intuitively, as long as $v \simeq t_j$, the substitutions $\theta_s[\![\{inv\} \cup \phi_{\mathsf{T}}]\!]$, $\theta_s[\![A_{\mathsf{T}}]\!]$ and $\theta_s[\![F_{\mathsf{T}}]\!]$ correspond to taking the transformation of P && $\{\, \text{x} \mapsto \mathcal{C}_M(s) \,\}[\![\text{INV}]\!]$. This leads to the definition

$$\mathcal{P}_{\text{INV}}(s) = (\, \theta_s[\![\phi_{\mathsf{T}}]\!] \cup \{\, b_{\mathsf{T}} \iff (v \simeq t_j), \; b_{\mathsf{T}} \implies \theta_s[\![inv]\!] \,\}, \theta_s[\![A_{\mathsf{T}}]\!], \theta_s[\![F_{\mathsf{T}}]\!] \,).$$

Note that, again, we preserve the $b \implies c$ structure of the resulting clause set. We require that b_{T} be fresh for each $s \in S_i(v)$ as it is fully defined by the resulting clause set. As in the relevant function terms case, the set of all relevant σ_{T}-sorted terms is given by

$$S_{all,i}(V, A) = \bigcup\nolimits_{v \in V} S_i(v) \cup \bigcup\nolimits_{app \in A} S_i(\text{disp}(app)).$$

Recall that our definition of $\mathcal{C}_M(t, (\mathsf{T}_1, \ldots, \mathsf{T}_n) \Rightarrow \mathsf{T}_{res})$ supposes the ability to generate a value v_{res} of type T_{res}. As our goal is to

produce inputs m such that INV *holds on* m, we must ensure that $\mathtt{v}_{res} \triangleright \text{INV}$ for all first-class functions in $\mathcal{C}_M(\text{vars}(\mathtt{P}))$. The proof of Theorem 20 shows that we can construct such a value by extending u_0 with $(\{b_\mathsf{T}, inv\} \cup \phi_\mathsf{T}, A_\mathsf{T}, F_\mathsf{T})$. However, if $\neg$INV is *valid*, namely no value $\mathtt{E}_\mathsf{T}$ exists such that $\{\,\mathtt{x} \mapsto \mathtt{E}_\mathsf{T}\,\}[\![\text{INV}]\!] \xrightarrow{*} \mathbf{true}$, then ϕ_i may be Unsat even though no value of type T is required in m. Given $\mathtt{v} \in \text{vars}(\mathtt{P})$, one can easily determine whether $\mathtt{v}.tpe$ will require an instance $\mathtt{v}_\mathsf{T}$ of type T inside of $\mathcal{C}_M(\text{vars}(\mathtt{P}))$. We say vars(P) *depend on* T iff there exists a $\mathtt{v} \in \text{vars}(\mathtt{P})$ such that $\mathtt{v}.tpe$ requires an instance $\mathtt{v}_\mathsf{T}$.

Based on these considerations, we extend the $U(\mathtt{P})$ procedure to $U_{\text{INV}}(\mathtt{P}) = u_0, u_1, \ldots$ in the following way. Given $u_0' \in U(\mathtt{P})$ and $(b_\mathtt{P}, \text{INV}) \rightsquigarrow (inv, \phi_p, A_p, F_p)$, if (and only if) vars(P) *depend on* T, we let $u_0 = u_0' \cup (\{inv\} \cup \phi_p, A_p, F_p)$ and add x to vars(P). Furthermore, we extend $\mathcal{C}_M(t, (\mathtt{T}_1, \ldots, \mathtt{T}_n) \Rightarrow \mathtt{T}_{res})$ such that $\mathcal{C}_M(\mathtt{x})$ is used when constructing $\mathtt{v}_{res}$ if $\mathtt{T}_{res}$ depends on T. Otherwise, we have $u_0 = u_0'$ and vars(P) are left unchanged. Finally, u_i is defined given u_{i-1} as

$$u_i = u_{i-1} \cup \mathcal{I}(app_i, fun_i) \cup \bigcup\nolimits_{s \in S_{all,i}(\text{vars}(\mathtt{P}), A_i)} \mathcal{P}_{\text{INV}}(s).$$

Clearly, given $(\phi_i, A_i, F_i) \in U_{\text{INV}}(\mathtt{P})$, the extension to $\phi_i \cup \psi_i \cup \rho_i$ only adds constraints to potential models, giving us

Proposition 24. *For* $(\phi_i, A_i, F_i) \in U_{\text{INV}}(\mathtt{P})$ *with associated* ψ_i, ρ_i*, given* $M \models \phi_i \cup \rho_i \cup \psi_i$*, then* $\mathcal{C}_M(vars(\mathtt{P}))[\![\mathtt{P}]\!] \xrightarrow{*} \mathbf{false}$*.*

The more interesting aspect of the updated procedure resides in the *kind* of satisfying models that are produced. Indeed, $\mathcal{P}_{\text{INV}}(s)$ for $s \in S_{\mathtt{P},i}$ ensures that INV is considered for all relevant ADT terms, thus leading to the following statements:

Theorem 25. *For* $(\phi_i, A_i, F_i) \in U_{\text{INV}}(\mathtt{P})$ *with associated* ψ_i, ρ_i*, if* $M \models \phi_i \cup \psi_i \cup \rho_i$*, then* INV holds on $\mathcal{C}_M(\text{vars}(\mathtt{P}))$.

Theorem 26. *If inputs* m *exist such that* INV holds on m *and* $m[\![\mathtt{P}]\!] \xrightarrow{*} \mathbf{false}$*, then there exists* $(\phi_i, A_i, F_i) \in U_{\text{INV}}(\mathtt{P})$ *with associated* ψ_i, ρ_i *such that* $M \models \phi_i \cup \psi_i \cup \rho_i$ *exists.*

It is clear that the above results can be extended to an arbitrary set $\text{INV}_1, \ldots, \text{INV}_m$ of ADT invariants as all $\mathcal{P}_{\text{INV}_i}$ are independent. Furthermore, one can safely use U_{POST} in conjunction with U_{INV} as both procedures are independent. Note that $\text{INV}_1, \ldots, \text{INV}_m$ cannot introduce relevant non-termination as unsoundness will only arise if $\text{POST}_1, \ldots, \text{POST}_n$ are not post-terminating for all inputs.

8. Explicit First-Order Quantification

To specify assumptions and invariants about first-class functions, we typically which to describe the behavior of these functions on infinite families of inputs. For this purpose, we introduce explicit universal first-order quantifiers in our specification language. Instead of relying on SMT solvers to instantiate quantifiers, our system performs quantifier instantiation internally. Thanks to this approach, our system can report counterexamples for some of the queries on which our SMT solvers return "Unknown" in their default configuration. While we acknowledge that the use of quantifiers is expensive, it is a price to pay for modular reasoning about first-class functions. Note that our users need not rely on explicit quantifiers as often as in some other verification approaches: constructs such as recursion and ADT invariants can be thought as particularly structured forms of quantification, which our system handles more efficiently, as discussed in previous sections.

Consider the PureScala expression $\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}$. Clearly, no reasonable evaluation semantics can be attached to $\forall$, so we define its semantics as follows: given inputs m, we say $\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}$ *holds on* m iff for every mapping

$$m_{\text{QUANT}} = \{\, \mathtt{x}_k \mapsto \mathtt{E}_k \mid \mathtt{E}_k : \mathtt{x}_k.tpe,\ \mathtt{E}_k \text{ ground},\ 1 \leq k \leq m \,\},$$

we have $(m \cup m_{\text{QUANT}})[\![\text{PROP}]\!] \xrightarrow{*} \mathbf{true}$. Now assume that $\mathtt{x}_1$ to $\mathtt{x}_m$ only appear in argument position to function calls in PROP and $\mathtt{x}_k.tpe$ is first-order for $1 \leq k \leq m$ (*i.e.* no instance of $\mathtt{x}_k.tpe$ contains a function). Let $(b_\mathtt{P}, \text{PROP}) \rightsquigarrow (p, \phi_p, A_p, F_p)$. Given function type $\mathtt{T} = (\mathtt{T}_1, \ldots, \mathtt{T}_n) \Rightarrow \mathtt{T}_{res}$, consider the set $G_{\mathsf{T},j}(A_p)$, $1 \leq j \leq n$ of tuples (b_G, t) where b_G corresponds to the condition under which t is a valid quantifier-free instantiation for argument position j. Further consider the set $X_k(A_p)$, $1 \leq k \leq m$ of tuples (b_X, s) where b_X is the condition under which s is a valid quantifier-free instantiation for the quantified variable $\mathtt{x}_k$. Given $(b_k, s_k) \in X_k(A_p), 1 \leq k \leq m$, we let

$$\text{inst}((b_1, s_1), \ldots, (b_m, s_m)) = (\, b_1 \wedge \cdots \wedge b_m,\ \{\, \mathcal{V}(\mathtt{x}_k) \mapsto s_k \mid 1 \leq k \leq m \,\}\,).$$

For each $app \in A_p$ where $app = (b, \_, \mathtt{T}, a^n)$ and argument position $1 \leq j \leq n$, consider the following set constraints

$$\textsc{Init-G}\ \frac{a_j \text{ quantifier-free}}{(b, a_j) \in G_{\mathsf{T},j}(A_p)} \qquad \textsc{Init-X}\ \frac{a_j = \mathcal{V}(\mathtt{x}_k)}{G_{\mathsf{T},j}(A_p) = X_k(A_p)}$$

$$\textsc{Expand}\ \frac{\begin{array}{c} a_j \textit{ not} \text{ quantifier-free} \\ (b_1, s_1) \in X_1(A_p) \quad \ldots \quad (b_m, s_m) \in X_m(A_p) \\ (b_s, \theta_s) = \text{inst}((b_1, s_1), \ldots, (b_m, s_m)) \end{array}}{(b \wedge b_s, \theta_s[\![a_j]\!]) \in G_{\mathsf{T},j}(A_p)}$$

One should note here that the sets $G_{\mathsf{T},j}(A_p)$ and $X_k(A_p)$ are not necessarily finite. We let $\mathcal{X}(A_p) = X_1(A_p) \times \cdots \times X_m(A_p)$ and given $x \in \mathcal{X}(A_p)$ where $x = ((b_1, s_1), \ldots, (b_m, s_m))$, we write inst$(x)$ for inst$((b_1, s_1), \ldots, (b_m, s_m))$.

Let us now define the sequence $Q(A_p) = q_0, q_1, \ldots$ where $q_i = (\mathcal{X}_i, A_i)$ such that both $\mathcal{X}_i$ and A_i are finite. We define $\mathcal{X}_{\text{INIT}}(A_p)$ as the set obtained using only the inferrence rules INIT-G and INIT-X on A_p. Note that $\mathcal{X}_{\text{INIT}}(A_p)$ is guaranteed finite. We therefore let $q_0 = (\mathcal{X}_{\text{INIT}}(A_p), A_p)$. Let us now consider $x_i \in \mathcal{X}_{i-1}$ where $(b_s, \theta_s) = \text{init}(x_i)$. We define A_i as

$$A_i = A_{i-1} \cup \{\, (b \wedge b_s, \theta_s[\![c]\!], tpe, \theta_s[\![a^n]\!]) \mid (b, c, tpe, a^n) \in A_{i-1}, a^n \text{ not quantifier-free} \,\}$$

and let $\mathcal{X}_i = \mathcal{X}_{\text{INIT}}(A_i)$. Note that this definition of $\mathcal{X}_i$ corresponds to applying a single time the EXPAND rule to the set $\mathcal{X}_{i-1}$ and computing the (finite) fixpoint with respect to INIT-G and INIT-X.

Lemma 27. *For* $Q(A_p) = (\mathcal{X}_0, \_), (\mathcal{X}_1, \_), \ldots, \bigcup_i \mathcal{X}_i = \mathcal{X}(A_p)$.

Note that given $(\mathcal{X}_i, A_i) \in Q(A_p)$, $x \in \mathcal{X}_i$, $(b_k, s_k) \in x$ and some subterm $t = \text{dispatch}_{tpe}(\ldots) \subseteq s_k$, there must always exists a corresponding $app \in A_i$ such that $t = \text{disp}(app)$ and $\models b_k \iff (app)_b$ as $\mathcal{X}_{\text{INIT}}$ does not introduce new terms. This ensures that the different underlying $U(\cdot)$ procedures can simply consider the set A_i for their various use cases as it faithfully represents the set of dispatches taking place in the formulas.

Given $x \in \mathcal{X}_i$ and fresh constant b_q, consider the transformation $(b_q, \text{PROP}) \rightsquigarrow (q, \phi_q, A_q, F_q)$ and $(b_s, \theta_s) = \text{inst}(x)$. The substituted sets $\theta_s[\![\{q\} \cup \phi_q]\!]$, $\theta_s[\![A_q]\!]$ and $\theta_s[\![F_q]\!]$ correspond to the instantiation of PROP with the quantifier-free terms s_1 to s_m. We want this instantiation to be conditioned on b_1 to b_m to ensure the quantifier-free terms are well-formed. Based on this, we define

$$\mathcal{P}_{\text{QUANT}}(x) = (\, \theta_s[\![\phi_q]\!] \cup \{\, b_q \iff bs,\ b_q \implies \theta_s[\![q]\!], \theta_s[\![A_q]\!], \theta_s[\![F_q]\!]\,).$$

We now define the sequence $U_{\text{QUANT}}(\text{PROP}) = u_1, u_2, \ldots$ of triplets $u_i = (\phi_i, A_i, F_i)$ where $u_0 = (\{b_q, q\} \cup \phi_q, A_q, F_q)$. Note that this simply corresponds to introducing the quantifier-free instantiations $(true, \mathcal{V}(\mathtt{x}_k))$ into each X_k for $1 \leq k \leq m$, a sound instantiation for non-empty sort $\mathcal{S}(\mathtt{x}_k.tpe)$. For u_i, we nondeterministically chose one of the following:

1. $u_i = u_{i-1} \cup \mathcal{I}(app_i \in A_{i-1}, fun_i \in F_{i-1})$,
2. $u_i = u_{i-1} \cup \mathcal{P}_{\text{QUANT}}(x_i \in \mathcal{X}_{\text{INIT}}(A_i))$.

We assume a fair alternation between the two presented options and fair selections of app_i, fun_i, x_i. Note that for $(\mathcal{X}_j, A_j) \in Q(A_q)$, there exists $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$ such that $A_j \subseteq A_i$ and $\mathcal{X}_j \subseteq \mathcal{X}_{\text{INIT}}(A_i)$.

Theorem 28. *Given $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$, if $\phi_i \in$ Unsat, then there exist no inputs m on which $\forall \mathsf{x}_1, \ldots, \mathsf{x}_m.\text{PROP}$ holds.*

Let us now discuss the parallel between our definition of $\mathcal{X}(A_p)$ and the work presented in [10]. Consider the clause set ϕ_p where $\mathcal{V}(\mathsf{x}_k), 1 \leq k \leq m$ are quantified. As our transformation introduces a single uninterpreted function symbol for each function type T, our set constraints correspond exactly to those discussed in [10], with the addition of conditionals. Note that we do not consider clause-indexed instantiations for the quantified variables as the notion of *clause* is not so clear in PureScala, yet this simplification only translates into extra (sound) instantiations in our procedure. The conditional associated to each $\mathcal{P}_{\text{QUANT}}$ instantiation corresponds to a *path condition* under which each instantiation is relevant. As long as PROP falls into a fragment that is covered in [10] (namely only pseudo-macro function definitions), our preserves Theorem 1 from [10] as negated *path conditions* will translate into clause satisfiability for any quantifier-free instantiation of a given clause, therefore rendering the associated quantifier-free instantiations irrelevant. Therefore, there are interesting fragments for which U_{QUANT} constitutes a semi-decision procedure.

Remember from our definition of U_{QUANT} that there may exist $app_q \in A_0$ such that app is not quantifier-free. Now consider $u_i = u_{i-1} \cup \mathcal{I}(app_q, fun_i)$ for some $i \in \mathbb{N}$; given our definition of $\mathcal{X}_i = \mathcal{X}_{\text{INIT}}(A_i)$, if $\mathcal{I}(app_q, fun_i)$ introduces an $app \in A_i$ with argument a_j such that $a_j = \mathcal{V}(\mathsf{x}_k)$, $\mathcal{X}_i$ will satisfy the new set constraint given by INIT-X. In other words, our instantiation procedure will automatically account for the implicit universal quantification given by the definition of $(app)_{\mathtt{f}}$, namely

$$\forall \mathsf{x}_1, \ldots, \mathsf{x}_n.\mathtt{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n) = \{\, \mathtt{f}.args \mapsto \mathsf{x}^n \,\}[\![\mathtt{f}.body]\!].$$

Let us now extend the procedure to model finding. Consider $(\mathcal{X}_i, A_i) \in Q(A_p)$ and model M. We say M *finitely interprets* $\mathcal{X}_i$ iff for each $x \in \mathcal{X}_i$ with $(b_s, \theta_s) = \text{inst}(x)$, we either have

1. $M \models \neg b_s$, or
2. for each $app_q \in A_i$ where $app_q = (b_q, c_q, tpe_q, a_q^n)$ such that app_q is *not* quantifier-free, there exists $app_g \in A_i$ where $app_g = (b_g, c_g, tpe_g, a_g^n)$ such that $M \models b_q \iff b_g$, $M \models \theta_s[\![c_q]\!] \simeq c_g$, $tpe_q = tpe_g$ and $M \models \theta_s[\![a_q^n]\!] \simeq a_g^n$.

Lemma 29. *For $(\mathcal{X}_i, A_i) \in Q(A_p)$ and model M, if M* finitely interprets $\mathcal{X}_i$*, then M* finitely interprets $\mathcal{X}_{i+1}$.

As $\mathcal{X}_i$ and A_i are finite, the above conditions can be encoded into a clause set γ_i. Note that given a model $M \models \gamma_i$, then for all $j > i$, we have $M \models \gamma_j$. For a given $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$, we already had associated ψ_i, ρ_i, and we can now consider an associated γ_i obtained from A_i and $\mathcal{X}_{\text{INIT}}(A_i)$. These considerations lead to the following theorem

Theorem 30. *Given $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$ with associated ψ_i, ρ_i, γ_i, and $M \models \phi_i \cup \psi_i \cup \rho_i \cup \gamma_i$, if $\forall \mathsf{x}_1, \ldots, \mathsf{x}_m.\text{PROP}$ is in a semi-decidable fragment, then $\forall \mathsf{x}_1, \ldots, \mathsf{x}_m.\text{PROP}$* holds on $\mathcal{C}_M(\text{vars}(\forall \mathsf{x}_1, \ldots, \mathsf{x}_m.\text{PROP}))$.

One should note at this point that all first-class functions contained within $\mathcal{C}_M(\cdot)$ have finite range. As our exploration of relevant arguments is complete for semi-decidable fragments, if inputs where all first-class functions have finite range exists, then these ranges will eventually be covered by the considered arguments, leading to the following (conditional) completeness guarantee for model finding

Theorem 31. *If there exist inputs m such that for each first-class function $\mathtt{f} \in m$, the set $\{\, \mathrm{v} \mid \mathtt{f}(\mathrm{E}_1, \ldots, \mathrm{E}_n) \xrightarrow{*} \mathrm{v} \,\}$ is finite and $\forall \mathsf{x}_1, \ldots, \mathsf{x}_m.\text{PROP}$ is in a semi-decidable fragment, then there exists $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$ with associated ψ_i, ρ_i, γ_i, and model M such that $M \models \phi_i \cup \psi_i \cup \rho_i \cup \gamma_i$.*

It is again trivial to extend $U_{\text{QUANT}}(\cdot)$ to multiple $\text{PROP}_1, \ldots, \text{PROP}_m$. Note that when considering U_{QUANT} in conjunction with U_{INV}, we must ensure that our assumption on type inhabitance is satisfied, thus conditionning soundness of validity proofs to satisfiability of the relevant INV_i expression.

It is also interesting to consider the procedure that takes a quantifier-free program P as argument and shows that to inputs m exist such that $\text{PROP}_1, \ldots, \text{PROP}_m$ *hold on* m and $m[\![\mathrm{P}]\!] \xrightarrow{*} \mathbf{false}$. Given the transformation $(b_{\mathrm{P}}, \mathrm{P}) \rightsquigarrow (p, \phi_{\mathrm{P}}, A_{\mathrm{P}}, F_{\mathrm{P}})$, it suffices to let $(\phi_{\mathrm{P}} \cup \{b_{\mathrm{P}}, \neg p\}, A_{\mathrm{P}}, F_{\mathrm{P}}) \subseteq u_0$ in $U_{\text{QUANT}}(\{\, \text{PROP}_1, \ldots, \text{PROP}_m \,\}, \mathrm{P})$ to obtain such a procedure. However, termination considerations now appear, much as in the U_{POST} case. These are handled similarly to U_{POST} by extending the call-graph with new relevant edges and showing that the considered measure remains strictly decreasing.

9. Implementation and Evaluation

We have implemented the techniques described in this paper as a plugin for the scalac compiler that accepts a subset of Scala and performs verification of such programs. We illustrate the behavior of our verifier on a sample of benchmarks to indicate that our techniques are of practical interest; a more detailed description of implementation and evaluation is beyond the scope of this anonymous submission.

Table 1 shows the results we choose to highlight. Most verification benchmarks feature some use of higher-order functions and a fair selection display useful interactions between ADT invariants and first-order quantifiers. Our benchmarks present a non-trivial List library with the usual higher-order operators which feature verified contracts describing various properties such as associativity (map, flatMap), equivalence (exists, forall, folds, etc.), and the monadic laws for flatMap. It is interesting to note that we can produce interface-like features by passing around first-class functions. Consider for example the following signature

```
def mergeSort[T](list: List[T])(key: T ⇒ ℤ): List[T] = { ... }
ensuring { res ⇒
  isSorted(res)(key) && content(list)(key) == content(res)(key) }
```

where the key function provides a mapping into the well-ordered domain of integers. Multiset contents and sortedness are then defined with respect to the provided key function.

ADT invariants have enabled a significant decrease of the annotation burden in certain involved benchmarks such as binary search tree insertion verification (Tree). First-order quantifiers are featured in the Tree, MergeSort (where they are used to encode higher-order functional equality), ArraySearch and AssocCommut (various mathematical properties of associative and commutative functions) benchmarks. Note that although the performance of these benchmarks is slower then those without quantifiers, such verification tasks remain tractable and feature a very high degree of automation. In terms of performance, our implementation of the quantifier instantiation procedure is in many cases not as fast as implementations that would use internal data structures of SMT solvers, but we have identified cases where quantifier instantiation techniques in the solvers we use (Z3 and CVC4) do not succeed in reporting models for quantified conjectures, whereas our approach does.

Our termination prover shows termination of all listed verification benchmarks, though it requires the introduction of ghost

Table 1. Summary of evaluation results for verification, featuring lines of code, (V)alid, (I)nvalid and (U)nknown verification conditions, as well as running time of the procedure.

Operation	LoC	V	I	U	Time (s)
`List`	820	242	0	0	6.85
`Tree`	95	15	0	1	1.10
`StateMonad`	212	19	0	0	0.50
`OptionMonad`	49	9	0	0	0.39
`ExceptionMonad`	61	6	0	0	0.36
`FoldAssociative`	104	24	0	0	1.40
`BalancedBrackets`	225	39	0	0	1.12
`ArraySearch`	166	29	2	0	20.64
`MergeSort`	67	13	0	0	11.64
`AssocCommut`	44	3	2	0	13.80
Total	1843	399	4	1	57.80

Table 2. Summary of evaluation results for termination checker, featuring lines of code, (P)roved functions, (N)on-terminating inputs reported and (U)nknown, as well as running time of our tool.

Operation	LoC	P	N	U	Time (s)
`NNF`	101	14	0	0	13.49
`Ackermann`	10	2	0	0	1.54
`QuickSort`	62	10	0	0	8.48
`RedBlackTree`	60	8	0	0	1.25
`LambdaCalculus`	53	4	1	0	11.78
`CyclicReasoning`	30	1	3	0	8.13
`HOTransformations`	41	3	0	0	5.43
`McCarthy91`	10	0	0	1	(150.43)
Total	357	42	4	1	50.10

variables to handle binary search and merge sort as the relevant measures are entirely non-trivial. We further list certain interesting termination benchmarks that showcase the flexibility of our tool. The evaluation results for these can be found in Table 2. Our prover can also report non-terminating inputs in certain classic cases of non-termination. For example, out prover is able to show that evaluation is non-terminating in the untyped lambda calculus (Appendix D.1) and produces the non-terminating input $(\lambda x.xx)(\lambda x.xx)$, namely the well-known omega term. Our termination checker defers non-termination checking until all positive techniques have failed to produce a proof, contributing to larger execution times in non-terminating cases. Overall, we obtain a good tradeoff between automation and performance by using a cascading model of termination checking that starts by invoking simple (and fast) procedures and then discharges failed checks to more powerful techniques. In practice, a significant portion (almost half) of our termination proofs are constructed without requiring any solver interaction, justifying this incremental approach.

10. Related Work

Verification of higher-order functional programs has gained recent traction through different significant approaches.

Model checkers for higher-order programs. Dependent type systems with refinement types such as *Liquid Types* [32, 39] and approaches based on model checking higher-order recursion schemas [15, 16, 27] are powerful verification techniques that elegantly handle higher-order functions. These procedures can benefit from counterexample-guided abstraction refinement (CEGAR) for high levels of automation and even boast relative completeness results for program typing [36]. It can however prove quite challenging to identify the cause of proof failure [38]. Note that our ADT invariants can be viewed as some form of refinement types [13, 37]. Recent work has shown that shape analysis of symbolic execution traces can prove effective at identifying contract violations in a (relatively) complete manner [23, 24]. These techniques, although similar in their aim to ours, use significantly different approaches and are therefore adapted to orthogonal problems such as dynamically typed languages and mutable state. Other techniques focus on unfolding first-order recursive functions [34, 35] or treat the simpler case of functions without datastructures or quantifiers [40].

Theorem provers and solvers. Automatic HOL theorem provers such as LEO II [4] and Satallax [8] feature semi-decidability results for certain HOL semantics. Given some suitable encoding, these systems can automatically verify properties in the fragment we discussed, however, their effectiveness on benchmarks with heavy usage of data types and integers is expected to be limited [33]. Saturation-based first-order theorem provers [1, 25] can efficiently handle *first-order* logic with equality [2, 17] but handling theories and higher-order functions remains a difficulty despite progress [29]. Interactive theorem provers such as Isabelle/HOL [26] and Coq [5] enable large scale real world verification efforts at the cost of automation. Partial automation [7], and counterexample finders for higher-order functions [6, 22] further improve the usability of these tools, though user interaction generally remains a requirement. Finally, SMT solvers such as Z3 feature powerful quantifier instantiation procedures [10] (see Section 8). Considering quantifiers at the source level (as in our case) can significantly improve the performance of program verification [20].

Automated termination provers. Automated termination proving for first-order programs has long existed in industrial-grade frameworks such as ACL2 [9, 14, 21]. Recent work has introduced various procedures that tackle the higher-order case as well [11, 12, 18] through a combination of reachability analysis and upper-bounding the call-graph path lengths (size-change termination [19] in our case). Our procedure is able to automatically synthesize (and assume) inductive invariants relevant to the considered measure, thus significantly increasing its effectiveness in practice. We have found our termination prover to suffice our needs when writting interesting and applied benchmarks.

Finite model finders. All models reported by our procedure have finite nature (they have a finite range in the case of first-class functions). Techniques for finite model finding can be quite effective at counterexample reporting by explicitly considering potential models [6], uses an underlying relational constraint solver to guarantee completeness for bounded domains [22], or through incremental model search where model size is increased at each step [3, 30]. Our system uses specification execution to speed up model finding.

11. Conclusions

Our system has been implemented by gradually enriching a verifier for first-order programs with features that contribute to modular specification and verification. The resulting design aims for simplicity of the specification language: it uses the programming language expressions for specifications, and builds on well-understood concepts of contracts, invariants, and quantifiers. The system is effective when used in fully automated mode, which makes it convenient for concisely documenting and verifying algorithms and data structures.

References

[1] L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I, chapter 11, pages 353–397. Kluwer, 1998.

[2] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In F. Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.

[3] P. Baumgartner, J. Bax, and U. Waldmann. Finite quantification in hierarchic theorem proving. In *IJCAR*, volume 8562 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2014.

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

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

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

[7] S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[8] C. E. Brown. Satallax: An automatic higher-order prover. In *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 111–117. Springer, 2012.

[9] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore. Acl2s: "the ACL2 sedan". *Electr. Notes Theor. Comput. Sci.*, 174(2):3–18, 2007.

[10] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009. doi: 10.1007/978-3-642-02658-4_25.

[11] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7, 2011.

[12] N. D. Jones and N. Bohr. Call-by-value termination in the untyped lambda-calculus. *Logical Methods in Computer Science*, 4(1), 2008.

[13] G. Kaki and S. Jagannathan. A relational framework for higher-order shape analysis. In *ICFP*, pages 311–324. ACM, 2014.

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

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

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

[17] K. Korovin and C. Sticksel. iprover-eq: An instantiation-based theorem prover with equality. In *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 196–202. Springer, 2010.

[18] T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 392–411. Springer, 2014.

[19] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92. ACM, 2001.

[20] K. R. M. Leino and C. Pit-Claudel. Trigger selection strategies to stabilize program verifiers.

[21] P. Manolios and A. Turon. All-termination(t). In *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2009.

[22] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *ICSE (1)*, pages 609–619. IEEE Computer Society, 2015.

[23] P. C. Nguyen and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 446–456, 2015.

[24] P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Higher-order symbolic execution for contract verification and refutation. *CoRR*, abs/1507.04817, 2015.

[25] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning (Volume 1)*, chapter 7. Elsevier and The MIT Press, 2001.

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

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

[28] B. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Mass., 2001.

[29] V. Prevosto and U. Waldmann. SPASS+T. In *ESCoR: Empirically Successful Computerized Reasoning*, volume 192, 2006.

[30] A. Reynolds, C. Tinelli, A. Goel, and S. Krstic. Finite model finding in SMT. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 640–655. Springer, 2013.

[31] J. C. Reynolds. *Logical foundations of functional programming*, chapter Introduction. Addison-Wesley, 1990.

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

[33] N. Sultana, J. C. Blanchette, and L. C. Paulson. LEO-II and satallax on the sledgehammer test bench. *J. Applied Logic*, 11(1):91–102, 2013.

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

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

[36] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, pages 75–86. ACM, 2013.

[37] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013.

[38] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: experience with refinement types in the real world. In *Haskell*, pages 39–51. ACM, 2014.

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

[40] N. Voirol, E. Kneuss, and V. Kuncak. Counter-example complete verification for higher-order functions. In P. Haller and H. Miller, editors, *Scala@PLDI*, pages 18–29. ACM, 2015. doi: 10.1145/2774975.2774978.

APPENDIX

A. Syntax of PureScala

B. Encoding Pattern Match Expressions

Consider pattern matching expressions that deconstruct algebraic data type instances into their component fields. We require pattern matches to cover all ADT constructors. Given a set of n cases where pattern match case $i \in \{1, \ldots, n\}$ has shape

$$\text{case}_i ::= \textbf{case}\ \text{cons}_i(\mathsf{v}_{i,1}, \ldots, \mathsf{v}_{i,m})\ \Rightarrow\ \text{E}_i$$

and given pattern match expression

$$\text{E} ::= \text{SCRUT}\ \textbf{match}\ \{\ \text{case}_1\ \ldots\ \text{case}_n\ \},$$

the transformation of (b, E) closely resembles that of **if**-expressions. For each case_i, we define b_i a fresh boolean-sorted constant. Given $\text{E} : \mathsf{T}$, we also define the fresh constant r with sort $\mathcal{S}(\mathsf{T})$. Given $(b, \text{SCRUT}) \rightsquigarrow^{\mathsf{t}} s$, we consider for each case_i the substitution $\theta_i = \{\, \mathcal{V}(\mathsf{v}_{i,j}) \rightarrow field_{i,j}(s) \mid 1 \leq j \leq m \,\}$ and compute

$(b_i, \text{E}_i) \rightsquigarrow^{\text{t}} e_i$ for $1 \leq i \leq n$. This lets us define the clausal encoding of pattern match expressions as

$$\begin{aligned}\phi_{split} = \{\ (b \wedge isCons_i(s)) \iff b_i \mid 1 \leq i \leq n\ \} \\ \cup \{\ b_i \implies (r \simeq \theta_i[\![e_i]\!]) \mid 1 \leq i \leq n\ \}.\end{aligned}$$

Note that unlike in the **if**-expression case for b_t and b_e, we don't need to explicitly ensure $b_i \implies \neg b_j$ when $i \neq j$ as this is ensured by the ADT theory ($isCons_i(s)$ is mutually exclusive with $isCons_j(s)$). This leads to our final inferrence rule

$$\textsc{Match}\ \frac{\begin{array}{c} r, b_1, \ldots, b_n \text{ fresh} \qquad (b, \text{SCRUT}) \rightsquigarrow (s, \phi_s, A_s, F_s) \\ (b_i, \text{E}_i) \rightsquigarrow (e_i, \phi_i, A_i, F_i), 1 \leq i \leq n \end{array}}{\begin{aligned}(b, \text{E}) \rightsquigarrow (\ & r, \\ & \phi_s \cup \theta_1[\![\phi_1]\!] \cup \cdots \cup \theta_n[\![\phi_n]\!] \cup \phi_{split}, \\ & A_s \cup \theta_1[\![A_1]\!] \cup \cdots \cup \theta_n[\![A_n]\!], \\ & F_s \cup \theta_1[\![F_1]\!] \cup \cdots \cup \theta_n[\![F_n]\!]\ )\end{aligned}}$$

We use $\theta_i[\![\cdot]\!]$ on all terms stemming from the transformation of E_i to ensure correspondence between the match deconstruction semantics and the ADT theory axiomatization. Our transformation therefore guarantees that $field_{i,j}(s)$ terms can only occur in positions where $isCons_i(s)$ must hold.

C. Proofs

Lemma 1. *There exists model $M \models b \wedge \phi_{\text{E}} \wedge \phi_{\text{V}} \wedge e \simeq v$.*

Proof. First note that v is not necessarily ground as it may contain constants introduced by the transformation of first-class functions in V. However, ϕ_{V} must be empty given the definition of value expression V. Furthermore, given two expressions E_1, E_2 and constant b with $(b, \text{E}_i) \rightsquigarrow (e_i, \phi_{\text{E}_i}, \_, \_)$, $m[\![\text{E}_i]\!] \xrightarrow{*} \text{V}_i$, and $(b, \text{V}_i) \rightsquigarrow^{\text{t}} v_i$ for $i \in \{1, 2\}$, we have $vars(v_1) \cap vars(v_2) = \emptyset$ as all introduced constants are fresh. Therefore, given models $M_i \models b \wedge \phi_{\text{E}_i} \wedge e_i \simeq v_i, i \in \{1, 2\}$, there must exist a model $M \models b \wedge \phi_{\text{E}_1} \wedge \phi_{\text{E}_2} \wedge e_1 \simeq v_1 \wedge e_2 \simeq v_2$. It is then trivial to conclude the proof by induction over the structure of E. ☐

Lemma 3. *There exists an M satisfying Lemma 1 such that for $\text{E}_{\text{C}} \subseteq \text{E}$ where $\text{E}_{\text{C}} ::= \text{C}(\text{E}_1, \ldots, \text{E}_n)$ and for $fun \in F_{\text{E}}$, if $M \models C_b(\text{E}_{\text{C}}) \wedge (fun)_b$, then $M \models C_t(\text{C}) \simeq (fun)_f$ iff $C_{\text{V}}(\text{C}) = C_{\text{V}}(C_{\text{E}}((fun)_f))$.*

Proof. Follows directly from Lemma 1 and Proposition 2. Indeed, if $C_t(\text{C}) \simeq (fun)_f$ must hold, then C and $C_{\text{E}}((fun)_f)$ are one and the same in $m[\![\text{E}]\!]$. Otherwise, an arbitrary (different) value can be chosen for the constant $(fun)_f$. ☐

Lemma 4. *Given inputs m with $m[\![\text{P}]\!] \xrightarrow{*} \text{V}$ and $(b_{\text{P}}, \text{V}) \rightsquigarrow^{\text{t}} v_{\text{P}}$, given $\text{E}_{\text{C}} \subseteq \text{P}$ where $C_t(\text{E}_{\text{C}}) = \text{dispatch}_{tpe}(c, a_1, \ldots, a_n)$ and $(C_b(\text{E}_{\text{C}}), C_{\text{V}}(\text{E}_{\text{C}})) \rightsquigarrow^{\text{t}} v_{\text{C}}$, there exists model M such that either*

1. *$M \models b_{\text{P}} \wedge \phi_{\text{P}} \wedge e_{\text{P}} \simeq v_{\text{P}} \wedge \theta_{\texttt{f}}[\![\phi_{\texttt{f}}]\!] \wedge \theta_{\texttt{f}}[\![e_{\texttt{f}}]\!] \simeq v_{\text{C}}$, or*
2. *$M \models b_{\text{P}} \wedge \phi_{\text{P}} \wedge e_{\text{P}} \simeq v_{\text{P}} \wedge \neg b_{\texttt{f}}$.*

Proof. Follows from Lemma 3 and the correspondence between the definition of the $\theta_{\texttt{f}}$ substitution and the operational semantics for both functions (closure substitution) and function applications (argument substitution). ☐

Theorem 5. *Given $\text{P} : \textsf{Boolean}$ with ϕ_i from $U(\text{P})$, if $\phi_i \in \textsf{Unsat}$, then no inputs m exist such that $m[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$.*

Proof. Let us assume inputs m exist such that $m[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$. Lemma 1 tells us that there exists $M_0 \models b_{\text{P}} \wedge \neg e_{\text{P}} \wedge \phi_{\text{P}}$. Lemma 4 then gives us a model $M_1 \models \phi_1$ and the proof follows through by induction on M_i. ☐

Lemma 6. *Given ϕ_i from $U_{\text{POST}}(\text{E}_{\text{POST}})$, if $\phi_i \in \textsf{Unsat}$ and POST terminates on all inputs, then there exist no inputs m such that $m[\![\text{POST}]\!] \xrightarrow{*} \textbf{false}$.*

Proof. First off, evaluation semantics of function application clearly ensure a correspondance between inputs to POST and E_{POST}. Let us assume there exist inputs m such that $m[\![\text{POST}]\!] \xrightarrow{*} \textbf{false}$. Consider the set S defined as the smallest fixpoint where

1. $\texttt{f}(m[\![\text{v}_1]\!], \ldots, m[\![\text{v}_n]\!]) \in S$,
2. if $\texttt{f}(\text{E}_1, \ldots, \text{E}_n) \in S$, then for each $\texttt{f}(\text{E}'_1, \ldots, \text{E}'_n)$ encountered during evaluation of $\{\ \text{v}^n \mapsto \text{E}^n\ \}[\![\text{POST}]\!]$, $\texttt{f}(\text{E}'_1, \ldots, \text{E}'_n) \in S$.

We assume m is selected such that for each $\texttt{f}(\text{E}_1, \ldots, \text{E}_n) \in S$, either $\text{E}_i = m[\![\text{v}_i]\!], 1 \leq i \leq n$, or $\{\ \text{v}^n \mapsto \text{E}^n\ \}[\![\text{POST}]\!] \xrightarrow{*} \textbf{true}$. Note that such inputs can be obtained by recursively selecting the arguments of the offending application in S as inputs m. Further note that termination of each $\{\ \text{v}^n \mapsto \text{E}^n\ \}[\![\text{POST}]\!]$ is ensured by the condition on POST.

Lemma 1 ensures that for each $\texttt{f}(\text{E}_1, \ldots, \text{E}_n) \in S$ with associated $b_{\texttt{f}}, app_j, fun_j$ where $(fun_j)_{\texttt{f}} = \texttt{f}$, given the clause set obtained by $(\phi_{\text{POST},j}, \_, \_) = \mathcal{P}_{\text{POST}}(app_j, fun_j)$, there exists a model $M_j \models b_{\texttt{f}} \wedge \phi_{\text{POST},j}$. Finally, we can adapt option No. 1 of Lemma 4 to $M \models b_{\text{P}} \wedge \phi_{\text{P}} \wedge \neg e_{\text{P}} \wedge \theta_{\texttt{f}}[\![\phi_p]\!] \wedge \theta_{\texttt{f}}[\![p]\!]$, thus concluding our proof. ☐

Theorem 7. *Given ϕ_i from $U_{\text{POST}}(\text{P})$, if $\phi_i \in \textsf{Unsat}$, POST is valid, and POST terminate on all inputs, then there exist no inputs m such that $m[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$.*

Proof. We assume inputs m exist such that $m[\![\text{P}]\!] \xrightarrow{*} \textbf{false}$. Validity and termination of POST, along with Lemma 6, ensure that there exists $M_j \models \phi_{\text{POST},j}$ for each $(\phi_{\text{POST},j}, \_, \_) = \mathcal{P}_{\text{POST}}(app_j, fun_j)$. Theorem 5 then ensures there exists $M_{\text{P}} \models \phi_{\text{P},k}$ from $U(\text{P})$ and Lemma 4 finally guarantees that M_{P} and all M_j can be unified into a single model $M \models \phi_i$ from $U_{\text{POST}}(\text{P})$. ☐

Theorem 11. *If for each $\text{E} \in \bigcup_{\texttt{f} \in \text{P}} C_{\texttt{f}}$ there exists some ϕ_i from $U(\text{DEC}(\text{E}))$ such that $\phi_i \in \textsf{Unsat}$, then P terminates for all inputs.*

Proof. Given $\text{E} \in C_{\texttt{f}}$, Theorem 5 ensures that no inputs m exist such that $m[\![\text{DEC}(\text{E})]\!] \xrightarrow{*} \textbf{false}$, namely, either $m[\![\text{DEC}(\text{E})]\!] \xrightarrow{*} \textbf{true}$ or $m[\![\text{DEC}(\text{E})]\!]$ diverges. However, if $m[\![\text{DEC}(\text{E})]\!]$ does not terminate, then there must exist an infinite sequence of function application evaluations $\text{APP}_0, \text{APP}_1, \ldots$ such that $\text{APP}_0 ::= \texttt{f}(m[\![\texttt{f}.args]\!])$

$$\begin{array}{rcl}
definition & ::= & \textbf{abstract class}\ id\ tdecls \\
 & & \mid \textbf{case class}\ id\ tdecls\ (\ decls\ )\ \textbf{extends}\ id\ tdecls \\
 & & \mid \textbf{def}\ id\ tdecls\ (\ decls\ )\ :\ type = expr \\
tdecls & ::= & \epsilon \mid [\ id\ \langle\ , id\ \rangle^*\ ] \\
decls & ::= & \epsilon \mid id\ :\ type\ \langle\ , id\ :\ type\ \rangle^* \\
expr & ::= & \textbf{true} \mid \textbf{false} \mid id \\
 & & \mid \textbf{if}\ (\ expr\ )\ expr\ \textbf{else}\ expr \\
 & & \mid id\ tparams\ (\ \langle\ expr\ \langle\ , expr\ \rangle^*\ \rangle?\ ) \\
 & & \mid expr\ (\ \langle\ expr\ \langle\ , expr\ \rangle^*\ \rangle?\ ) \\
 & & \mid (\ decls\ )\ \Rightarrow\ expr \\
 & & \mid expr\ \textbf{match}\ \{ \\
 & & \quad \langle\ \textbf{case}\ id\ (\ \langle\ id\ \langle\ , id\ \rangle^*\ \rangle?\ )\ \Rightarrow\ expr\ \rangle^+ \\
 & & \quad \} \\
tparams & ::= & \epsilon \mid [\ type\ \langle\ , type\ \rangle^*] \\
type & ::= & id\ tparams \mid \textsf{Boolean} \\
id & ::= & \textsf{IDENT}
\end{array}$$

Figure 6. Syntax of PureScala

and $(\text{APP}_i, \text{APP}_{i+1}) \in R$. For each $\text{APP}_i ::= \mathtt{f}_{i+1}(\text{A}_{i,1}, \ldots, \text{A}_{i,n_i})$, there exists by definition some $\mathtt{f}_{i+1}(\text{E}_{i,1}, \ldots, \text{E}_{i,n_i}) \in C_{\mathtt{f}_i}$ and inputs m_i such that $m_i[\![\text{path}(\mathtt{f}_{i+1}(\text{E}_{i,1}, \ldots, \text{E}_{i,n_i}))]\!] \xrightarrow{*} \mathbf{true}$ and $m_i[\![\text{E}_{i,j}]\!] \xrightarrow{*} \text{A}_{i,j}$ for $1 \leq j \leq n_i$. Given $\sum_j \text{size}(\text{A}_{i,j}) \xrightarrow{*} \text{V}_i$, we therefore have an infinite sequence $\text{V}_1 > \text{V}_2 > \cdots \geq 0$ and obtain a contradiction. Applying Proposition 10 then concludes our proof. □

Theorem 13. *For $\mathtt{f} \in$ P, if $Posts(\mathtt{f})$ is post-terminating for all inputs, $Posts(\mathtt{f})$ are valid, and for each $\text{E} \in \bigcup_{g \in \text{P}} C_{\mathtt{g}}$ there exists some ϕ_i from $U_{\text{POST}}(Posts(\mathtt{f}), \text{DEC}(\text{E}))$ such that $\phi_i \in$ Unsat, then* P *terminates for all inputs.*

Proof. Follows from conditions of U_{POST} soundness begin satisfied and similar arguments to Theorem 11. □

Lemma 16. *For $fun_1, fun_2 \in F$ where $fun_j = (\_, f_j, \mathtt{f}_j, \_)$ and $\mathtt{f}_j$ is ground for $j \in \{1, 2\}$, we have $\mathtt{f}_1 = \mathtt{f}_2$ iff some model $M \models f_1 \simeq f_2 \wedge EQ_{\mathtt{f}}(F)$ exists.*

Proof. Proposition 15 ensures that there is an $M \models EQ_{\mathtt{f}}(F)$. By induction on the definition of eq, one sees that the only non-trivial instance is case No. 1 when $C_t(\text{E}_1)$ or $C_t(\text{E}_2)$ contains some term with sort $\sigma_{\mathtt{f}}$. However, as the $\mathtt{f}_j$ are ground, the definition of $EQ_{\mathtt{f}}$ ensures that there is a $x \overset{?}{\sim} y$ clause in $EQ_{\mathtt{f}}(F)$ for such terms. One then concludes the proof by induction on the set of first-class functions in $\mathtt{f}_j$ that satisfy the conditions of case No. 1. □

Lemma 18. *Given a model $M \models \eta_i(v)$, there is no term $s \subseteq M[\![v]\!]$ with sort $\sigma_{\mathtt{f}}$ such that $M \models s \notin E_i(v)$.*

Proof. Trivial if $\sigma = \sigma_{\mathtt{f}}$. The clause $\eta_i(v)$ ensures there exists $t_j, j \leq i$ such that $M \models v \simeq t_j$. The $\sigma_{\mathtt{f}}$-exhaustiveness constraint on the enumeration of σ ensures that there exists a model M_2 and term t_k such that $M_2 \models M[\![v]\!] \simeq t_k$ and for any $s \subset M[\![v]\!]$ with sort $\sigma_{\mathtt{f}}$, there exists $s_k \subset t_k$ such that $M_2 \models s \simeq s_k$. Unicity further guarantees that $j = k$ and therefore $M[\![v]\!] = M_2[\![t_k]\!]$. □

Theorem 20. *Given a model $M \models \phi_i \cup \psi_i \cup \rho_i$, we have $\mathcal{C}_M(\text{vars}(\text{P}))[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$.*

Proof. We start by defining $\mathcal{C}'_M$, an extension to $\mathcal{C}_M$ with the following extra rule

$$\frac{\begin{array}{c} fun \in F_i, fun = (\_, f, \mathtt{f}, c^n) \\ M \models t \simeq f \qquad \text{E}_i = \mathcal{C}'_M(c_i, \mathtt{f}.closures_i.tpe), 1 \leq i \leq n \end{array}}{\mathcal{C}'_M(t, \mathtt{f}.tpe) = \{\, \mathtt{f}.closures \mapsto \text{E}^n \,\}[\![\mathtt{f}]\!]}$$

It is clear given the definition of $\mathcal{C}'_M$ that there exists a left inverse to $\mathcal{C}'_M$, which we will call $\mathcal{C}'^{\leftarrow}_M$, such that $\mathcal{C}'^{\leftarrow}_M(\mathcal{C}'_M(t, \mathtt{T})) = t$.

Consider the constant-expression pair b, E that was transformed during $U(\text{P})$ and consider the corresponding $(b, \text{E}) \rightsquigarrow (e, \phi, A, F)$. Note that vars(E) − vars(P) is not necessarily empty due to term substitutions in pattern-matching expression transformation and $\mathcal{I}(app_i, fun_i)$ dispatch. Although (e, ϕ, A, F) may appear substituted in $U(\text{P})$, it is clear that one can determine the subset of app_j, fun_j pairs in D_i that are related to E. We therefore let Φ_{E} consist of the unsubstituted portion of $\phi_i \cup \psi_i \cup \rho_i$ that stems from E (note that $\phi \subseteq \Phi_{\text{E}}$). We show by induction that if there exists $M_{\text{E}} \models \{b\} \cup \Phi_{\text{E}}$, then $\mathcal{C}_{M_{\text{E}}}(\text{vars}(\text{E}))[\![\text{E}]\!] \xrightarrow{*} \mathcal{C}'_{M_{\text{E}}}(e, \text{E}.tpe)$.

Let us consider the case of pattern-matching expressions where

$$\text{E} ::= \text{S}\ \mathbf{match}\ \{\ \ldots\ \mathbf{case}\ \text{cons}_j(\mathtt{v}_{j,1}, \ldots, \mathtt{v}_{j,m_j}) \Rightarrow \text{R}_j\ \ldots\ \}.$$

By induction, we have $\mathcal{C}_{M_{\text{E}}}(\text{vars}(\text{S}))[\![\text{S}]\!] \xrightarrow{*} \text{V}$ and can assume without loss of generality that $\text{V} ::= \text{cons}_j(\text{E}_1, \ldots, \text{E}_{m_j})$. Consider the model $M_{\text{R}_j} = M_{\text{E}}\{\, \mathcal{V}(\mathtt{v}_{j,k}) \mapsto \mathcal{C}^{\leftarrow}_{M_{\text{E}}}(\text{E}_k) \mid 1 \leq k \leq m_j \,\}$ and note that $M_{\text{E}} \models \{b\} \cup \Phi_{\text{E}}$ implies that $M_{\text{R}_j} \models \{C_b(\text{R}_j)\} \cup \Phi_{\text{R}_j}$ and $\mathcal{C}'_{M_{\text{E}}}(e, \text{E}.tpe) = \mathcal{C}'_{M_{\text{R}_j}}(C_t(\text{R}_j), \text{R}_j.tpe)$.

Now consider the case of function application expressions where $\text{E} ::= \text{C}(\text{E}_1, \ldots, \text{E}_n)$, and consider $(C_b(\text{E}), \text{E}) \rightsquigarrow_{\mathtt{t}} \text{disp}(app)$ where $app = (\_, c, \_, a^n)$. We further consider the unsubstituted sets A_{E}, F_{E} and $T_{\text{E}}(app)$ that stem from E. We examine two mutually exclusive cases:

- $M_{\text{E}} \models c \in T_{\text{E}}(app)$: consider $fun \in F_{\text{E}}$ corresponding to c where $fun = (\_, \_, \mathtt{f}, c^m)$, as well as $\text{C}_j = \mathcal{C}'_{M_{\text{E}}}(c_j, \mathtt{v}_j.tpe)$ for $\mathtt{v}_j \in \mathtt{f}.closures$ and $\text{A}_k = \mathcal{C}'_{M_{\text{E}}}(a_k, \mathtt{v}_k.tpe)$ for $\mathtt{v}_k \in \mathtt{f}.args$. Let us now define the following models

$$M_{\mathtt{f}} = M_{\text{E}}\{\, \mathcal{V}(\mathtt{v}_j) \mapsto \mathcal{C}^{\leftarrow}_{M_{\text{E}}}(\text{C}_j) \mid \mathtt{v}_j \in \mathtt{f}.closures \,\},$$
$$M_{\mathtt{f}.body} = M_{\mathtt{f}}\{\, \mathcal{V}(\mathtt{v}_k) \mapsto \mathcal{C}^{\leftarrow}_{M_{\text{E}}}(\text{A}_k) \mid \mathtt{v}_k \in \mathtt{f}.args \,\}.$$

  Since $M_{\text{E}} \models \{c \in T_{\text{E}}(app), C_b(\text{E})\} \cup \Phi_{\text{E}}$, we have that $M_{\mathtt{f}.body} \models \{b_{\mathtt{f}}\} \cup \Phi_{\mathtt{f}.body}$. Given $(b_{\mathtt{f}}, \mathtt{f}.body) \rightsquigarrow_{\mathtt{t}} body$, we therefore have $\mathcal{C}'_{M_{\text{E}}}(e, \text{E}.tpe) = \mathcal{C}'_{M_{\mathtt{f}.body}}(body, \mathtt{f}.body.tpe)$. Finally, Lemma 16 ensures consistency of the uninterpreted function symbol applications $\text{disp}(app)$ given the ground first-class function $\{\, \mathtt{v}_j \mapsto \text{C}_j \mid \mathtt{v}_j \in \mathtt{f}.closures \,\}[\![\mathtt{f}]\!]$.
- $M_{\text{E}} \models c \in E_{all,i}(\text{vars}(\text{P}), A_{\text{E}})$: the ψ_i clause set ensures that $M[\![E_{all,i}(\text{vars}(\text{P}), A_{\text{E}})]\!] \cap M[\![T_{\text{E}}(app)]\!] = \emptyset$, and therefore $\mathcal{C}'_{M_{\text{E}}}(c, \text{C}.tpe) = \mathcal{C}_{M_{\text{E}}}(c, \text{C}.tpe)$. To conclude this case, it suffices to note that $\mathcal{C}_{M_{\text{E}}}(\text{vars}(\text{E}))[\![\text{E}]\!] \xrightarrow{*} \mathcal{C}'_{M_{\text{E}}}(e, \text{E}.tpe)$ by definition of $\mathcal{C}_{M_{\text{E}}}$.

The remaining cases for E follow trivially from the inductive hypothesis. To conclude our proof, one notes that $M \models \{b_{\text{P}}\} \cup \Phi_{\text{P}}$ where $\Phi_{\text{P}} = \phi_i \cup \psi_i \cup \rho_i$ and $M \models \neg C_t(\text{P})$, therefore we have $\mathcal{C}'_M(C_t(\text{P})) = \mathbf{false}$ and $\mathcal{C}_M(\text{vars}(\text{P}))[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$. □

Theorem 21. *If inputs m exist such that $m[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$, then there exists $i \in \mathbb{N}$ and model M such that $M \models \phi_i \cup \psi_i \cup \rho_i$.*

Proof. By Proposition 15, Proposition 17 and Proposition 19 along with freshness of all relevant constants, we have $M_\psi \models \psi_i$ existence. Theorem 5 further ensures that $M_\phi \models \phi_i$ exists and the proof of Theorem 5 in combination with Lemma 16 ensures that $M_{\phi,\psi} \models \phi_i \cup \psi_i$ exists as well.

Let us now consider ρ_i. We define the set A_m of all app tuples corresponding to function applications encountered during evaluation of $m[\![\text{P}]\!]$. Similarly, let F_m the set of fun tuples corresponding to encountered first-class functions that are defined within P. Finally, let $\mathtt{F}_m$ the set of first-class functions defined in m that can appear in caller position of $app \in A_m$. Note that this restriction filters out the first-class functions in equality positions of $\mathcal{C}_M$ outputs.

Our fairness requirement ensures that $(app)_c \in T_i(app)$ will eventually be satisfied for $app \in A_m$ when $(app)_c$ corresponds to some $fun \in F_m$. For $\mathtt{f} \in \mathtt{F}_m$, consider $\mathtt{f}_2$ of the shape returned by $\mathcal{C}_M$ such that $\mathtt{f}$ and $\mathtt{f}_2$ agree on all relevant inputs given by A_m and the shape of $\mathtt{f}_2$ ensures distinctness from functions in F_m (ensured by definition of $\mathcal{C}_M$). As no equality predicate on function types exists in P, given $m_2 = \{\, \mathtt{f} \mapsto \mathtt{f}_2 \,\}[\![m]\!]$, it is clear that $m_2[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$. There must therefore exist some inputs m_k such that $m_k[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$ and the associated $\mathtt{F}_{m_k}$ contains only functions of the shape returned by $\mathcal{C}_M$, thus ensuring satisfiability of $\phi_i \cup \psi_i \cup \rho_i$ for some $i \in \mathbb{N}$. □

Lemma 23. *Given a model $M \models \eta_i(v)$, there is no term $s \subseteq M[\![v]\!]$ with sort $\sigma_{\mathtt{T}}$ such that $M \models s \notin S_i(v)$.*

Proof. Consider the case where $\sigma = \sigma_{\mathtt{T}}$. Unlike for Lemma 18, this case is non-trivial as there may exist $s \subset M[\![v]\!]$ with sort $\sigma_{\mathtt{T}}$. Furthermore, given t_j such that $M \models v \simeq t_j$, theory separation doesn't ensure that there is a corresponding $s_j \subset t_j$ such that

$s = M[\![s_j]\!]$ as t_j and s have the same sort $\sigma_{\mathtt{T}}$. However, ADT thery axioms do ensure that such a correspondance exists, an observation that is not clear for any arbitrary theory. The rest of the proof closely follows the general case from Lemma 18. □

Theorem 25. *For* $(\phi_i, A_i, F_i) \in U_{\text{INV}}(\text{P})$ *with associated* ψ_i, ρ_i*, if* $M \models \phi_i \cup \psi_i \cup \rho_i$*, then* INV holds on $\mathcal{C}_M(\text{vars}(\text{P}))$.

Proof. Given $\mathtt{v} \in \text{vars}(\text{P})$, it follows from Lemma 23 and induction on $\triangleright$ that for each $\text{E} \subseteq \mathcal{C}_M(\mathtt{v})$ such that $\text{E} : \mathtt{T}$, there exists $s \in S_{all,i}(\text{vars}(\text{P}), A_i)$ such that $\mathcal{C}_M(s, \mathtt{T}) = \text{E}$. Furthermore, the $b_{\mathtt{T}}$ constant corresponding to s must hold in M.

Consider the subsets of $\phi_i \cup \psi_i \cup \rho_i$, A_i and F_i that derive from $\mathcal{P}_{\text{INV}}(s)$ and note that they correspond to the sequence $U(\neg\text{INV})$ where $\mathcal{V}(\mathtt{x})$ has been substituted by s. We can therefore construct a model $M_s = M\{\, \mathcal{V}(\mathtt{x}) \mapsto M[\![s]\!] \,\}$ such that for some $(\phi_j, A_j, F_j) \in U(\neg\text{INV})$ with associated ψ_j, ρ_j, we have $M_s \models \phi_j \cup \psi_j \cup \rho_j$. The proof of Theorem 20 then ensures that $\mathcal{C}_{M_s}(\text{vars}(\text{INV}))[\![\text{INV}]\!] \xrightarrow{*} \mathbf{true}$.

Finally, as $\mathtt{x} \in \text{vars}(\text{P})$ when an instance $\text{V}_{\mathtt{T}}$ is required for $\mathcal{C}_M(\text{vars}(\text{P}))$, $\mathcal{C}_M(\mathtt{x})$ will produce a value on which INV holds. Note that when no value $\text{V}_{\mathtt{T}}$ exists, either M doesn't exist, or $\mathcal{C}_M(\mathtt{x})$ won't terminate, thus ensuring soundness of produced inputs. □

Theorem 26. *If inputs* m *exist such that* INV holds on m *and* $m[\![\text{P}]\!] \xrightarrow{*} \mathbf{false}$*, then there exists* $(\phi_i, A_i, F_i) \in U_{\text{INV}}(\text{P})$ *with associated* ψ_i, ρ_i *such that* $M \models \phi_i \cup \psi_i \cup \rho_i$ *exists.*

Proof. Closely follows the proof of Theorem 21 where m is converted into m_k such that all first-class functions in m_k correspond to results of $\mathcal{C}_M$. Note that as INV *holds on* m, any V_{res} that depends on $\mathtt{T}$ can be obtained by evaluating the corresponding first-class function in m at an arbitrary point. □

Lemma 27. *For* $Q(A_p) = (\mathcal{X}_0, \_), (\mathcal{X}_1, \_), \ldots, \bigcup_i \mathcal{X}_i = \mathcal{X}(A_p)$.

Proof. Equivalence for INIT-G and INIT-X rules is given by definition. For EXPAND rule to apply, there must exist $app \in A_0$ such that app is *not* quantifier-free. The EXPAND rule application then corresponds exactly to introducing $(b \wedge b_s, \theta_s[\![c]\!], tpe, \theta_s[\![a^n]\!])$ into A_1 for each $app \in A_0$ that is not quantifier-free and applying INIT-G. The proof then follows by induction on i. □

Theorem 28. *Given* $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$*, if* $\phi_i \in$ *Unsat, then there exist no inputs* m *on which* $\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}$ holds.

Proof. Follows from the Compactness Theorem as each $\mathcal{P}_{\text{QUANT}}(x_i)$ corresponds to a quantifier-free instantiation of PROP. □

Lemma 29. *For* $(\mathcal{X}_i, A_i) \in Q(A_p)$ *and model* M*, if* M finitely interprets $\mathcal{X}_i$*, then* M finitely interprets $\mathcal{X}_{i+1}$.

Proof. Condition No. 1 clearly carries over to all $x \in \mathcal{X}_{i+1} - \mathcal{X}_i$. If condition No. 2 holds on $x_{i+1} \in \mathcal{X}_i$ selected for A_{i+1}, then given $(b_{i+1}, s_{i+1}) \in X_k(A_{i+1})$, there exists $(b_i, s_i) \in X_k(A_i)$ such that $M \models b_{i+1} \iff b_i$ and $M \models s_{i+1} \simeq s_i$. Therefore, all $x \in \mathcal{X}_{i+1} - \mathcal{X}_i$ will satisfy condition No. 2. □

Theorem 30. *Given* $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$ *with associated* ψ_i, ρ_i, γ_i*, and* $M \models \phi_i \cup \psi_i \cup \rho_i \cup \gamma_i$*, if* $\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}$ *is in a semi-decidable fragment, then* $\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}$ holds on $\mathcal{C}_M(\text{vars}(\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}))$.

Proof. Let us assume there exist no inputs m such that $\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}$ *holds on* m. Semi-decidability then tells us there exists $j > i$ such that $\phi_j \cup \psi_j \cup \rho_j \cup \gamma_j \in \mathsf{Unsat}$.

Let us consider u_{i+1} and $\Phi_{i+1} = \phi_{i+1} \cup \psi_{i+1} \cup \rho_{i+1} \cup \gamma_{i+1}$. We start by assuming u_{i+1} was obtained through $\mathcal{I}$. Note that if

$$M \not\models (app_{i+1})_b \wedge (fun_{i+1})_b \wedge (app_{i+1})_f \simeq (fun_{i+1})_f,$$

then all clauses introduced by $\mathcal{I}$ are irrelevant and the tuples in $A_{i+1} - A_i$ satisfy condition No. 1 of *finite interpretation*. Otherwise, $M \models \rho_i$ ensures that a corresponding inlining has already taken place and there must therefore exist a mapping θ from fresh variables introduced by $\mathcal{I}$ to variables in M such that $\theta[\![u_{i+1}]\!] \subseteq u_i$ and $M\{\, x \mapsto M[\![y]\!] \mid x \mapsto y \in \theta \,\} \models \Phi_{i+1}$.

If u_{i+1} was obtained through $\mathcal{P}_{\text{QUANT}}$, then $\phi_{i+1}, \psi_{i+1}, \rho_{i+1}$ are unchanged and Lemma 29 ensures that there exists model $M_{i+1} \models \Phi_{i+1}$. Finally, by induction on i we have existence of $M_k \models \Phi_k$ for all $k > i$ and thus obtain a contradiction. Validity of $\mathcal{C}_M(\text{vars}(\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}))$ is then ensured by applying the proof of Theorem 20. □

Theorem 31. *If there exist inputs* m *such that for each first-class function* $\mathtt{f} \in m$*, the set* $\{\, \text{V} \mid \mathtt{f}(\text{E}_1, \ldots, \text{E}_n) \xrightarrow{*} \text{V} \,\}$ *is finite and* $\forall \mathtt{x}_1, \ldots, \mathtt{x}_m.\text{PROP}$ *is in a semi-decidable fragment, then there exists* $(\phi_i, A_i, F_i) \in U_{\text{QUANT}}(\text{PROP})$ *with associated* ψ_i, ρ_i, γ_i*, and model* M *such that* $M \models \phi_i \cup \psi_i \cup \rho_i \cup \gamma_i$.

Proof. Follows from semi-decidability and fair exploration of relevant arguments to each $\mathtt{f}$. □

D. Examples

D.1 Lambda Calculus Evaluator

```
object LambdaCalculus {
  abstract class Term
  case class Var(x: BigInt) extends Term
  case class Abs(x: BigInt, body: Term) extends Term
  case class App(func: Term, arg: Term) extends Term

  def fv(t: Term): Set[BigInt] = t match {
    case Var(x) ⇒ Set(x)
    case Abs(x, body) ⇒ fv(body) ++ Set(x)
    case App(func, arg) ⇒ fv(func) ++ fv(arg)
  }

  // [x->u]t
  def subst(x: BigInt, u: Term, t: Term): Term = t match {
    case Var(y) ⇒ if (x == y) u else t
    case Abs(y, body) ⇒ if (x == y) t else Abs(y, subst(x, u, body))
    case App(f, a) ⇒ App(subst(x, u, f), subst(x, u, a))
  }

  // big step call-by-value evaluation
  def eval(t: Term): Option[Term] = (t match {
    case App(t1, t2) ⇒ eval(t1) match {
      case Some(Abs(x, body)) ⇒ eval(t2) match {
        case Some(v2) ⇒ eval(subst(x, v2, body))
        case None() ⇒ None[Term]()
      }
      case _ ⇒ None[Term]() // stuck
    }
    case _ ⇒ Some(t) // Abs or Var, already a value
  }) ensuring { res ⇒ res match {
    case Some(t) ⇒ isValue(t)
    case None() ⇒ true
  }}

  def isValue(t: Term): Boolean = t match {
    case Var(x) ⇒ true
    case Abs(x, body) ⇒ true
```

```
    case App(f, a) ⇒ false
  }
}
/* Termination checker (LoopProcessor) says:
   Non-terminating for call:
   eval(App(Abs(0, App(Var(0), Var(0))), Abs(0, App(Var(0), Var(0)))))
*/
```