Termination of Open Higher-Order Programs

VOIROL NICOLAS*, EPFL
MADHAVAN RAVICHANDHRAN†, EPFL
KUNCAK VIKTOR‡, EPFL

We study the problem of proving termination of open, higher-order programs with recursive functions and datatypes. We identify a new point in the design space of solutions, with an appealing trade-off between simplicity of specification, modularity, and amenability to automation. Specifically, we consider termination of open expressions in the presence of higher-order, recursive functions, and introduce a new notion of termination that is conditioned on the termination of the callbacks made by the expressions. For closed expressions our definition coincides with the traditional definition of termination. We derive sound proof obligations for establishing termination modulo callbacks, and develop a modular approach for verifying the obligations. Our approach is novel in three aspects. (a) It allows users to express properties about the environment in the form of higher-order contracts. (b) It establishes properties on the creation sites of closures instead of application sites and does not require knowing the targets of applications. (c) It uses a modular reasoning where termination (modulo callbacks) is verified for each function independently and then composed to check termination of their callers. We present the results of evaluating our approach on benchmarks comprising more than 10K lines of functional Scala code. The results show that our approach, when combined with a safety verifier, established both termination and safety of complex algorithms and data structures that are beyond the reach of state-of-the-art techniques. For example, it verifies Okasaki's scheduling-based data structures and lazy trees in under a few seconds.

1 INTRODUCTION

Termination verification, which is the problem of statically verifying whether a program terminates on all possible values that its parameters can take at execution time, is an ancient and important problem in software engineering. It plays a key role in establishing robust functioning of programs. From a broader perspective, termination verification is a method of ensuring *totality* of functions, especially those that are recursively defined. This makes it crucial for theorem proving systems (Kaufmann et al. 2000; Nipkow et al. 2002) that allow establishing theorems involving arbitrary recursive functions. Furthermore, termination verification also forms a key component of most contract-based verification techniques (Blanc et al. 2013; Jacobs et al. 2011; Leino 2010) by enabling an inductive reasoning that relies on the well-foundedness of the number of steps executed by a program on any input.

Techniques for proving termination of higher-order programs are gaining increasing relevance. This is in part due to the increase in popularity of functional programming languages as well as the adoption of higher-order functions in most languages being used in industry. Moreover, the soundness of popular higher-order interactive theorem proving and program verification systems (Bertot and Castéran 2004;

*nicolas.voirol@epfl.ch
†ravi.kandhadai@epfl.ch
‡viktor.kuncak@epfl.ch

Kobayashi et al. 2011; Nipkow et al. 2002; Vazou et al. 2014) typically relies on termination. However, most of the techniques for termination verification of higher-order programs are *whole program* analyses that work under a closed world assumption, i.e, they require that the targets of all first-class functions can be estimated (or approximated) at the time of the analysis, e.g. using a control-flow analysis. Such approaches have at least three main shortcomings. Firstly, users cannot reason at all about termination of an open library that uses higher-order features without linking the library to a client. Besides the scalability overheads this may result in, it also means that the termination of the library is established only with respect to a specific client. Hence, the library has to be *re-analyzed* for each new client. Secondly, whole program analyses often trade-off precision for scalability due to the need to analyze large pieces of code. Hence, these techniques are effective only on programs whose termination argument depends only on the invariants that can be inferred by a scalable, light-weight analysis. Since termination is an undecidable problem, there exist numerous interesting algorithms where the invariants required for termination are so complex that they are beyond the reach of existing automatic invariant inference techniques. For instance, lazy functional queues proposed by Okasaki (1998) rely on the invariant that the streams used by the queues have finite length. Defining such invariants requires recursive functions or quantifiers. (Section 5 provide many such examples.) Finally, it is difficult to integrate whole-program termination verifiers with modular verification techniques such as Vazou et al. (2014) and Voirol et al. (2015) that verify a contract (or refinement type) independently of the clients of the library.

In contrast, *modular* techniques that can reason about termination of different modules or functions of a higher-order program independently, and later compose the results of the analysis are rather scarce in literature. The primary challenge here is the need to reason about callbacks i.e, indirect calls whose targets depend on the calling contexts. In this paper, we propose a new approach for *modularly* reasoning about termination of open, higher-order functional programs with recursive functions and datatypes. One of key challenges in achieving this is devising a reasonable notion of termination for open, higher-order programs. Unlike the case of first-order programs, defining termination of open higher-order programs is not straightforward. For example, consider a map function shown below in the syntax of the Scala programming language (Odersky et al. 2008). The function takes an arbitrary function f and a non-empty list l as input, and applies f to every element of l.

```
def map[T,R](f: T ⇒ R, l: List[T]): List[R] = l match {
  case Nil() ⇒ Nil[R]()
  case Cons(h, t) ⇒ f(h) :: t.map(f)
}
```

The termination of the function map depends on the termination of the function f passed as input, since map may apply f during its execution. The parameter f can be a non-terminating function. While this suggests that map should be flagged as non-terminating, the function exhibits another interesting property that may suggest the opposite: map terminates whenever f is a terminating function. The latter is an interesting piece of information for the developer of the function, and is, in fact, what the developer expects. Even more interesting is the fact that this also enables modular reasoning, since if it is established that a caller of map passes a terminating function as f, the call to map is terminating. This raises the question of whether map should be blamed for the non-termination of a function passed as a parameter. In fact, the documentation of the Haskell stream library (Swierstra 2015) does not mention map as a non-terminating function, while it flags certain other functions as potentially diverging. Furthermore, the definition used by the related work (Giesl et al. 2006) on open program termination, namely *H-termination*, also considers the function map to be terminating.

In this paper, we provide a new perspective on the problem by proposing a novel definition for termination of open, higher-order functions namely *termination modulo callbacks*. A function is terminating modulo callbacks iff it terminates on every input for which the callbacks made by the function while executing under the input terminate. Intuitively, a callback to a function f is a call to a piece of code that is created outside the function f but is executed by the function. (These notions are formally defined in section 2.) For the map example, every call f(h) made at runtime is a callback. To check if map terminates modulo callbacks, it suffices to consider inputs to map where every f(h) is a terminating call. Since map terminates under all such inputs, it is terminating modulo callbacks. The function size shown below is an illustration of non-termination modulo callbacks. The function takes as input a node of a stream, and a next function that returns the next node of the stream given the current node (or None for the last node of the stream). The function size counts the number of elements in the stream.

```
def size[T](node: Option[T], next: T ⇒ Option[T]): Int = node match {
  case Some(x) ⇒ 1 + size(next(x), next)
  case None ⇒ 0
}
```

There exists an input to the function size under which the function is non-terminating even if all the callbacks it makes during the execution under the input are terminating. For instance, if next is defined as a function x ⇒ Some(x), which corresponds to an infinite stream of x, then size is non-terminating as it iterates on Some(x) forever. The documentation of the Haskell stream library documents every such function that iterates over a stream (e.g. filter, dropWhile) as potentially diverging. Note that if an expression does not make any callbacks, e.g. like the main function of a closed program, then the above notion coincides with the traditional definition of termination.

Advantages of Termination Modulo Callbacks. The following are the main reasons why we believe termination modulo callbacks is a useful notion for open, higher-order programs.

(a) *Intuitive appeal and blame assignment.* Based on our experience with studying and developing higher-order libraries, it appears that this definition matches the termination property that is of interest to developers of open, higher-order programs. In particular, reporting that a function does not terminate modulo callbacks is almost always an indication of a bug or an undocumented usage restriction.

This definition also seems to provides a satisfactory assignment of blame for non-termination. For instance, since map is terminating modulo callbacks, any non-termination during execution of map is the fault of the caller of map. However, non-termination during execution of size under the inputs shown above is the fault of the size function. In this case, either size should be disabled from being applicable under such environments, e.g. by a precondition or by a sanity check, or it should be modified so that it terminates for infinite streams.

(b) *Suitability for use in theorem provers and verifiers.* The assumption that the callbacks are terminating functions is well-suited for applications where all functions are required to be terminating, as in the case of

```
sealed abstract class Tree[T] { val depth: BigInt }
case class Leaf[T]() extends Tree[T] { val depth: BigInt = 0 }
case class Node[T](child: T ⇒ Tree[T], depth: BigInt) extends Tree[T] {
  require(depth > 0 && child.requires(_ ⇒ true) && child.ensures((_, res) ⇒ res.depth < depth)) }
```

Fig. 1. Lazy tree definition. The **require** keyword in the Node class states a class invariant and **requires** and **ensures** state higher-order contracts on the first-class function child (namely that child must be defined for all inputs and the depth of its output must be lower than the current depth).

```
def fold[C](tree: Tree[Alphabet], f: List[C] ⇒ C): C = {
  decreases(tree.depth)
  tree match {
    case Node(child, _) ⇒ f(Alphabet.values.map(a ⇒ fold(child(a), f)))
    case Leaf() ⇒ f(Nil()) }}
```

Fig. 2. Fold definition on a lazy tree. The **decreases** keyword specifies the ranking function of the fold.

interactive theorem provers and contract-based software verifiers. In these applications, every function is guaranteed to be used only under the environment where all the callbacks provably terminate.

(c) *Environment Agnostic Definition.* As opposed to whole-program analyses, checking whether a function terminates modulo callbacks can be accomplished independently of the calling context. This gels well with higher-order contract verification techniques that verify a function independently of the calling context.

(d) *Simplicity of the definition.* In comparison to the other existing definitions for termination of open, higher-order functions such as H-termination (Giesl et al. 2006) which is defined cyclicly using H-terminating terms, our definition is constructed out of the unambiguous notion of termination under a specific input and does not have any cyclic dependencies on itself.

1.1 Modular Verification of Termination Modulo Callbacks

In this paper, we propose a modular approach to verifying termination modulo callbacks of open programs and establish its soundness. The main distinguishing aspect of the approach compared to other existing related techniques is the ability of the approach to exploit user-provided contracts on functions and user-provided ranking functions to establish termination. The main challenge addressed by our approach is in making such heavy-weight reasoning scale by controlling the proof obligations (or verification conditions) that are generated, and by reducing the annotation overhead for users.

In particular, we derive novel and concise sufficient conditions, referred to as proof obligations, for checking termination of functions modulo callbacks. Interestingly, the sufficient conditions we derive allow us to verify termination of programs even without requiring the knowledge of the targets of the applications (or indirect calls). In fact, our approach does not even generate any proof obligations for checking indirect calls, instead it proves termination by establishing sufficient properties at closure creation sites. Since closure application sites far outnumber closure creation sites in practice, the technique generates far fewer proof obligations and scales to complex programs.

We now present an overview of the approach using the example shown in Fig. 1 and Fig. 2. Fig. 1 defines a datatype `Tree` with two constructors `Leaf`, which represents an empty tree, and `Node` which represents a non-empty tree. The datatype Tree defines an abstract field depth, which is initialized to zero in a leaf, and is defined at the time of construction for a node. The children of a node are given by a field child of function type T ⇒ Tree. child maps an index of type T to the corresponding children of the node. The construct **require** is used to specify function preconditions, as well as invariants of datatypes. Notice that the invariant of the type Node asserts that the function stored in the field child is such that the depth field of the tree returned by child is smaller than the depth field of the node. This is the key property that ensures that the tree has finite depth.

Consider now the fold function shown in Fig. 2 that defines a fold operation on a tree: Tree[Alphabet], where Alphabet is a class that denotes a finite sequence of distinct symbols. The children of the tree are indexed by the alphabet symbols. The fold function applies the function f recursively on the children of the tree, and combines the results of folding the children by applying f again. The **decreases** construct is used to specify a ranking function or measure that will be used by the termination checking algorithm, described shortly. The function map is defined as described earlier and is terminating modulo callbacks. We are interested in checking if the fold function is terminating modulo callbacks.

In principle, establishing termination of fold modulo callbacks depends on termination of the call to map and vice-versa. This is because first we need to show that the call Alphabet.values.map terminates (unconditionally). To prove this we need to know that the argument to the map terminates, which in turn requires knowing that the function fold terminates on the arguments passed to it. However, this cyclic dependency is not an impediment to our approach, which relies on creation-site-based reasoning. Our approach modularly establishes the termination (modulo callbacks) of fold and map by proving that there exists a measure that decreases across direct recursive calls and the recursive calls made within the *closures* created by the functions. For instance, it establishes that the measure of fold namely tree.depth decreases for the call to fold(child(a),f) made within the closure created by the function. This property is proved using the datatype invariant of the Node constructor shown in Fig. 1 that asserts that the depth of child(a) is smaller than depth of the tree. Similarly, we also establish that there exists a measure for map that decreases across the recursive calls made within the closures created by the map function, which in this case follows from the wellfoundedness of the List datatype. Though this creation-site reasoning may initially come as a surprise, we formally show in section 3 that this reasoning is sound by establishing non-trivial, previously unknown theoretical results in termination of higher-order functional expressions with recursion.

As illustrated by the above example, our approach has two main advantages: (1) functions can be independently checked for termination much as in contract-driven verification, and (2) we make no closed-world assumption, thus enabling reusability of library-level termination proofs across multiple clients. However, our algorithm can also perform non-modular reasoning, i.e analyze a function more than once, in cases where the contracts provided by the users are insufficient for establishing the required proof obligations. In our system, users have the option of helping the system scale or achieve better precision by providing contracts at the level of functions or types. But even in the absence of contracts (or insufficient contracts) they may rely on the automation in the system to establish the required obligations for termination, which may potentially (re)analyze the direct or local callees (i.e, non-callbacks) of a function. We refer to such an analysis where the users can tune the level of modularity based on their needs as a tunable modular analysis. To further improve automation, we also integrate a light-weight, non-modular static analysis to determine the set of local calls (i.e. non-callbacks) that could be transitively invoked by a function. In either case, our system ensures that the properties (1) and (2) listed above hold.

Contributions. To summarize, the main contributions of the paper are as follows:

- We present a new semantic notion for open higher-order program termination: termination modulo callbacks. We discuss the advantages of this notion and how it relates to pre-existing notions, and discuss compositionality of our notion. (Section 2)
- We discuss in Section 3 two different semantically-defined obligations that can be used to establish termination modulo callbacks. We prove that these obligations are sufficient to establish termination modulo callbacks and are compositional. We further show in Theorem 4 that the first obligation is weakly complete, in the sense that it will only fail to establish termination modulo callbacks if no compositional analysis can succeed. The obligations are based on a novel theorem (Theorem 1) on termination of closed higher-orders expressions with recursive functions under call-by-value evaluation semantics. We present detailed proofs of the theorems in the Appendix A.
- We present an approach and a system that modularly verifies the termination obligations using higher-order contracts. We show how verification and termination can mutually benefit from each other and establish the soundness of the verification approach (Section 4).
- We present the results of using our system to verify termination of numerous higher-order functional Scala programs that comprise more than 10K lines of Scala code. To our knowledge, no prior termination technique can prove termination of some of the data structures we handle such as Okasaki's scheduling-based, lazy data structures and lazy trees. These benchmarks demonstrate the effectiveness of our technique and also more broadly demonstrate the usefulness of higher-order contracts in higher-order termination verification (Section 5).

2 LANGUAGE AND SEMANTICS

$$\begin{array}{rcll}
 & & x \in \mathit{Vars} & \textit{(Variables)} \\
 & & f \in \mathit{Fids} & \textit{(Function identifiers)} \\
\tau \in \mathit{Type} & ::= & \bar{\tau} \mid \tau \Rightarrow \tau \mid \mathsf{Unit} & \\
v \in \mathit{Value} & ::= & \lambda x.f\ \bar{w} & \text{where } \bar{w} \in (\mathit{Value} \cup \{x\})^* \\
e_\lambda \in \mathit{Lambda} & ::= & \lambda x.f\ \bar{y} & \text{where } \bar{y} \in (\mathit{Value} \cup \mathit{Vars})^* \\
e \in \mathit{Expr} & ::= & x \mid e_\lambda \mid e\ e & \\
\mathit{Fdef} & ::= & \textbf{def}\ f(\bar{x}) : \tau \Rightarrow \tau := e & \\
L \in \mathit{Library} & ::= & 2^{Fdef} &
\end{array}$$

Fig. 3. Syntax of expressions, functions and libraries.

In this section, we formally introduce our notion of termination for open higher-order libraries. For perspicuity, we present this notion with respect to a toy functional language with (recursive) named functions, lambda expressions and call expressions. In the later sections, we enrich the language with type polymorphism, recursive datatypes and higher-order contracts, which takes it closer to the subset of the Scala language admitted by our tool. Fig. 3 shows the syntax and Fig. 4 the static semantics of the toy language. In the figure, $\bar{a}$ denotes a sequence of elements belonging to a domain A^*, a_i refers to the i^{th}

$$\{(f,\tau) \mid \textsf{def}\ f(\bar{x}) : \tau := e \in L\} \subseteq \Gamma \subseteq (\mathit{Vars} \cup \mathit{Fids}) \times \mathit{Type}$$

$$\textsc{Var}\ \frac{u \in \mathit{Vars} \cup \mathit{Fids} \qquad (u,\tau) \in \Gamma}{\Gamma \vdash u : \tau} \qquad \textsc{Lambda}\ \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \Rightarrow \tau_2} \qquad \textsc{DirectCall}\ \frac{\Gamma \vdash f : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash f\ e : \tau_2}$$

$$\textsc{Seq}\ \frac{|\bar{e}| = |\bar{\tau}| \qquad \forall i.\ \Gamma \vdash x_i : \tau_i}{\Gamma \vdash \bar{e} : \bar{\tau}} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \textsc{FunDef}\ \frac{|\bar{x}| = |\bar{\tau_1}| \qquad \Gamma[\bar{x} \mapsto \bar{\tau_1}] \vdash e : \tau_2}{\Gamma \vdash \textsf{def}\ f(\bar{x}) : \bar{\tau_1} \Rightarrow \tau_2 := e : \textsf{Unit}}$$

Fig. 4. Static typing rules for expressions with respect to a library L, following simply typed lambda calculus with recursion.

element of the sequence, and $|\bar{a}|$ refers to the length of the sequence. The language allows direct calls to named functions of the form $f\ e$ and also indirect calls (or applications) of the form $e\ e$. The domain *Value* defines the set of closed lambda terms (or closures), and the domain *Lambda* defines lambda terms with free variables. Without loss of generality, we enforce that the bodies of all lambdas are direct calls to named functions applied over variables or values. This syntactic restriction ensures that the computation performed by the lambdas are given by named functions in the program. This allows a succinct presentation of the termination conditions checked by our approach. (It is straightforward to relax this by introducing more explicit checks for lambdas which is incorporated in our implementation.) Every expression in our language is associated with a static label. The label is generally omitted but when needed is denoted as $\ell : e$. A library $L \in 2^{Fdef}$ is a finite set of function definitions that are closed with respect to named function calls. That is, every named function called in the bodies of functions (defined in L) are defined in L.

The static typing rules for the expressions are shown in Fig. 4. Since expressions can refer to named functions, an expression can be type checked only with respect to a library L. As shown in Fig. 4, every type environment Γ binds all functions in the library L to their declared types. This enables recursive function definitions since the body of a function f is allowed to refer to any named function defined in the library (including itself), as shown by the rule FunDef. Other typing rules shown in Fig. 4 are similar to that of *simply typed lambdas calculus*. Below we introduce a few notations used in the rest of the paper.

Basic Notation and Terminology. We use $A \mapsto B$ to denote a partial function from A to B. $h[a \mapsto b]$ denotes the function that maps a to b and every other value x in the domain of h to $h(x)$. We use $h[\bar{a} \mapsto \bar{b}]$ to denote $h[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. Given an expression e, let $FV(e)$ denote the set of free variables of e. We refer to an expression without free variables as a closed expression. Given a value $\lambda x.f\ \bar{w}$, we refer to the elements of $\bar{w}$ that are different from x as the *captured values* of the closure. Given a function $f \in L$, we use $\mathit{body}(f)$ and $\mathit{param}(f)$ to refer to the body and parameters of f. Note that these operations are always defined with respect to the library L.

The operation $e[e'/x]$ denotes the syntactic replacement of the *free occurrences* of x in e by e'. This operation replaces expressions along with their static labels and also performs alpha-renaming of bound variables, if necessary, to avoid variable capturing. We extend this operation to replacing a sequence of variables and denoted it by $e[\bar{e'}/\bar{x}]$. A substitution $\sigma : \mathit{Vars} \mapsto \mathit{Expr}$ is a partial function from variables to expressions. We use $e\ \sigma$ to denote $e[\sigma(x_1)/x_1] \cdots [\sigma(x_n)/x_n]$, where $\mathit{dom}(\sigma) \supseteq \{x_1, \cdots x_n\}$.

$$\textit{(Evaluation contexts)} \quad \mathcal{C} \in \mathit{EContext} = [\,] \mid \mathcal{C}\ e \mid v\ \mathcal{C}$$

$$\textsc{Context}\ \frac{e \rightsquigarrow e'}{\mathcal{C}[e] \rightsquigarrow \mathcal{C}[e']} \qquad \textsc{UnmarkedCall}\ \frac{v_1 = \lambda x.f\ \bar{w} \qquad b = \mathit{body}(f)}{v_1\ v_2 \rightsquigarrow b[\bar{w}/\mathit{param}(f)][v_2/x]} \qquad \textsc{MarkedCall}\ \frac{v_1 = (\lambda x.f\ \bar{w})+ \qquad mb = \mathit{markAll}(\mathit{body}(f))}{v_1\ v_2 \rightsquigarrow mb[\bar{w}/\mathit{param}(f)][v_2/x]}$$

Fig. 5. Operation semantics of the expressions. The function $\mathit{markAll}(e)$ marks every lambda in the expression e.

We define an *expression context* $\mathcal{C}$ as an expression with a hole, which can be substituted by another (type-compatible) expression e to obtain an expression denoted $\mathcal{C}[e]$. We say e_1 is a sub-expression of e_2 (denoted $e_1 \sqsubseteq e_2$) iff there exists a context $\mathcal{C}$ such that e_2 is syntactically equal to $\mathcal{C}[e_1]$. We say an expression e is well-typed with respect to a library L iff the expression type checks under an environment Γ that binds all functions in L to their types (as shown at the top of Fig. 4), and the static labels of expressions in e are unique.

Operational Semantics. Fig. 5 presents the small-step operational semantics for *closed expressions* that are well-typed with respect to a library L. The semantics are defined using a set of reduction rules similar to beta-reduction rules. However, we instrument the rules to also propagate *marks* (or taints) on lambda terms encountered during evaluation. A marked lambda term is denoted by $(\lambda x.f\ e)+$. Every lambda term is unmarked unless explicitly stated. The markers on lambdas will be used shortly to define the notion of callback, which in turn will be used to define the notion of termination modulo callbacks of open expressions.

Consider the reduction rules shown in Fig. 5. The evaluation context $\mathcal{C}$ and the reduction rule Context enforce a *call-by-value* evaluation strategy. The rules UnmarkedCall and MarkedCall perform the usual beta-reduction but with a minor extension. Since (by our syntactic restriction) the body of every lambda term is a direct call to a function (say f), the direct call is first applied it to its arguments before the beta-reduction. Note that the rule MarkedCall additionally marks every lambda term in $\mathit{body}(f)$ using the helper function *markAll*. This results in a property that every lambda term created by applying a marked lambda term is also marked, which is essential for defining callbacks. As is evident, the marks on lambda terms have no effect on the evaluation. In other words, the evaluation of an expression simulates the usual call-by-value evaluation strategy. We use $\rightsquigarrow^*$ to refer to the reflexive, transitive closure of $\rightsquigarrow$. Below we present a conventional semantic definition of termination of closed expressions.

Definition 2.1 (Termination). A closed expression e that is well-typed with respect to a library L terminates iff the evaluation sequence $e \rightsquigarrow e' \rightsquigarrow e'' \cdots$ is finite.

Semantics of Open Expressions and Termination Modulo Callbacks. We now define the semantics of *open expressions* (i.e, expressions with free variables). Subsequently, we define the notion of callbacks and termination modulo callbacks for open expressions.

Definition 2.2 (Closing Environment). Let e be a (possibly open) expression that is well-typed with respect to a library L. A closing environment of (e, L) is a pair (σ, L') where $L' \in 2^{Fdef}$ and $\sigma : \mathit{Vars} \mapsto \mathit{Value}$ is a substitution such that

(a) $L \cup L'$ is a library,

(b) $e\ \sigma$ is a well-typed, closed expression with respect to $L \cup L'$,

(c) The lambda terms in $\mathit{range}(\sigma)$ are marked.

The first condition ensures that the library $L \cup L'$ is well-formed i.e, has unique static labels for expressions and is closed with respect to named functions definitions. The second condition ensures that the substitution σ binds all free variables of e to type-compatible values. The third condition ensures that the lambdas that are bound to free variables are marked. By the definition of the operational semantics, this in turn implies that every lambda term created by applying the lambdas in $\mathit{range}(\sigma)$ are also marked. We refer to every application of a marked lambda that arises during the evaluation of $e\ \sigma$ as a callback of e. Formally,

Definition 2.3 (Callbacks). Let e be an expression well-typed with respect to a library L. Let (σ, L') be a closing environment of (e, L). $\mathit{Callbacks}(L, L', e, \sigma) = \{(v_1\ v_2) \mid \exists \mathcal{C} \in \mathit{EContext}, \{v_1, v_2\} \subseteq \mathit{Value}.\ (e\ \sigma) \rightsquigarrow^* C[v_1\ v_2] \wedge v_1$ is marked $\}$

Dual to callbacks, we define $\mathit{LocalCall}(L, L', e, \sigma)$ as the set of applications of lambda terms that are not marked. That is, $\mathit{LocalCall}(L, L', e, \sigma) = \{(v\ u) \mid \exists \mathcal{C} \in \mathit{EContext}.(e\ \sigma) \rightsquigarrow^* C[v\ u] \wedge v \in \mathit{LocalClosure}(L, L', \sigma, e)\}$, where $\mathit{LocalClosure}(L, L', e, \sigma)$ is the set of unmarked values that arise during the evaluation of $e\ \sigma$, i.e, $\{v \mid \exists \mathcal{C} \in \mathit{EContext}, v \in \mathit{Value}.\ (e\ \sigma) \rightsquigarrow^* C[v] \wedge v$ is unmarked $\}$.

In essence, callbacks are applications of lambda terms that are defined by the environment in which e is invoked. Thus, their behavior depends on the closing environment. Whereas, local calls apply local closures created by the expression e (possibly via other local calls) whose definition is available in e or in L. Intuitively, the non-termination of any callbacks made by e is the fault of the client of e and should not be blamed on e. Below we present the definition of termination modulo callbacks that is based on this intuition.

Definition 2.4 (Termination Modulo Callbacks). Let e be an expression that is well-typed with respect to a library L. e terminates modulo callbacks iff for all closing environments (σ, L') of (e, L),

$$(\forall cb \in \mathit{Callbacks}(L, L', e, \sigma).\ cb \text{ terminates}) \Rightarrow e\ \sigma \text{ terminates.}$$

Interestingly, if e does not make any callback, termination modulo callbacks reduces to plain old termination. In other words, for closed expressions or expression where the free variables do not have function types, termination modulo callbacks and termination coincide. We define non-termination modulo callbacks as the complement of termination modulo callbacks. We say a function is terminating (or non-terminating) modulo callbacks if its body expression is terminating (or non-terminating) modulo callbacks.

Tunable Modular Analysis. We are interested in reasoning techniques where functions (or expressions) are verified for a property $\mathcal{P}$ by utilizing the fact that their callees (or sub-expressions) satisfy the property $\mathcal{P}$, together with a few additional checks. Such techniques are called modular or compositional. For instance, type checking is modular because the well-typedness of a function (or expression) can be determined from the well-typedness of the callees (or sub-expressions) with an additional check for the compatibility of the types. A main advantage of modularity is that it allows reasoning about parts of the program independently of others while guaranteeing that the correctness of the parts will imply the correctness of the whole. This is particularly important for analyzing open libraries as it is desirable to avoid as much re-analysis of the library as possible for each new client. To make things more concrete we formally define a modular analysis or verifier as follows.

Modularity. An analysis $\mathcal{A} : \mathit{Library} \rightarrow \mathit{Summary}$, for some domain $\mathit{Summary}$, is modular (or library modular) iff for any library L and $L' \in 2^{Fdef}$ such that $L \cup L'$ is a library, $\mathcal{A}(L \cup L')$ is a function of $\mathcal{A}(L)$

```
def f(u): Unit ⇒ Unit := g (λx.h x)              def h(x): (Unit ⇒ Unit) ⇒ Unit := x Unit
def g(z): ((Unit ⇒ Unit) ⇒ Unit) ⇒ Unit := z (λy.g z)
```

Fig. 6. A challenging example for checking termination modulo callbacks modularly.

and L'. In other words, $\mathcal{A}(L \cup L')$ does not depend on the implementation of the functions in L but only on their summaries.

Despite the intuitive appeal of modular analyses, in practice, sacrificing modularity could be beneficial at times e.g. to reduce the user annotations required on function or library interfaces. We are interested in analyses where users are able to tune the degree of modularity in verification to achieve the right balance between user annotation overheads and performance. Users should have the option of helping the system scale or achieve better precision by providing specifications on function interfaces or modules when required, but otherwise may rely on the automation in the system to establish the required properties (even if it requires re-analyzing libraries). We believe this property allows for a truly scalable solution especially when heavy-weight techniques such as SMT solving or theorem proving techniques are employed, which are powerful but exhibit unpredictable behaviors.

Non-composability of Termination Modulo Callbacks. Termination modulo callbacks is inherently a non-modular property, which makes it challenging to design a modular reasoning technique. For instance, consider the functions shown in Fig. 6 expressed in the syntax of the toy language. The bodies of functions h and g are trivially terminating modulo callbacks since they just apply the input parameters x and z respectively, which are callbacks in any closing environment. However, the body of the function f, which is a closed expression, is non-terminating (and also non-terminating modulo callbacks) due to the cyclic reduction sequence: $g\ (\lambda x.h\ x) \rightsquigarrow^* (\lambda x.h\ x)\ (\lambda y.g\ (\lambda x.h\ x)) \rightsquigarrow^* g\ (\lambda x.h\ x)$. In other words, composing expressions that are terminating modulo callbacks does not preserve termination modulo callbacks. However, this does not imply that termination modulo callbacks cannot be verified modularly. It only implies that one needs to infer/verify some property in addition to (or stronger than) termination modulo callbacks for every function in a library to perform a modular verification.

3 TERMINATION OBLIGATIONS

In this section, we reduce the problem of verifying termination modulo callbacks to checking a set of semantically-defined safety assertions called *termination obligations*. The goal is to deduce assertions that are amenable to efficient, modular verification on the functions in a library L. (As shown by Fig. 6, the definition of termination modulo callbacks as such is not amenable to modular verification.) In section 4, we describe an algorithm for efficiently checking these conditions using a state-of-the-art safety verification technique for higher-order programs, based on higher-order contracts.

3.1 A Sound and Complete Condition for Termination

First, we present a new and important theorem about the closed evaluation of expressions in our language that are well-typed with respect to a library L. The termination obligations are deduced based on this theorem. The proofs of all theorems that follow are presented in detail in Appendices A and B. Before we present the theorem, we introduce a couple of simple definitions.

Definition 3.1 (Incomplete Call Sequence). Let e be a closed expression. A sequence of applications $S : (v_1\ u_1),\ (v_2\ u_2),\ \cdots$ is an *incomplete call sequence* in the evaluation of e iff the call $(v_i\ u_i)$ happens before $(v_{i-1}\ u_{i-1})$ completes. That is, there exists contexts $\{\mathcal{C}_i \mid i \geq 1\}$ such that $e \rightsquigarrow^* \mathcal{C}_1[(v_1\ u_1)]$ and for all $i \geq 2$, $(v_{i-1}\ u_{i-1}) \rightsquigarrow^* \mathcal{C}_i[(v_i\ u_i)]$.

Definition 3.2 (Parameter Substitution). Let e be a closed expression. Let $e \rightsquigarrow^* (\lambda x.f\ \bar{w})\ u$. Define $\sigma_{param}((\lambda x.f\ \bar{w})\ u)$ as the substitution that maps the parameters of the function f to the actual arguments passed to the function f, and where all lambda terms in the arguments are *marked.* That is, $\sigma_{param}((\lambda x.f\ \bar{w})\ u) = [param(f) \mapsto markAll(\bar{w}[u/x])]$.

We now present an important theorem that provides a sound and complete condition for proving termination of the evaluation of a closed expression. In simple words, the theorem states that for any closed expression e belonging to our language, its evaluation terminating if one can bound the length of every sequence of incomplete calls (i.e, calls that do not return) having the following property: there exists a function f such that every call in the sequence apply a closure whose target is f and each call in the sequence applies a closure *local* to the previous call in the sequence.

Theorem 1 (Locality of Termination)**.** *Let e be a closed expression that is well-typed under a library L. e is terminating if (and only if) for every function f, there exist a $n \in \mathbb{N}$ that upper bounds the length of every sequence of incomplete calls $S : (v_1\ u_1), (v_2\ u_2), \cdots$ where each v_i is of the form $\lambda x.f\ \bar{w}_i$ and $\forall i \geq 2.\ (v_i\ u_i) \in LocalCall(L, L, body(f), \sigma_{param}(v_{i-1}\ u_{i-1}))$.*

We now briefly explain the significance of this theorem. The only if direction of the theorem is trivial and hence is shown in parentheses, since if e is terminating then every sequence of incomplete calls in the evaluation of e should be finite. The other direction is non-trivial, since even if we know that the length of every sequence S as describe above is bounded by a number n, the number of such sequences could potentially be infinite meaning that the evaluation of e is non-terminating. The theorem asserts that this is not possible for a well-typed, closed expression belonging to our language.

This theorem is interesting because for each function f in a library L, it enables us to focus on calls that are *local* to f. The local calls only apply local closures whose definitions (i.e, corresponding lambda terms) are available within the function f or its named callees. The targets of these calls can be inferred just by traversing the abstract syntax trees of the function f and its (named) callees, as detailed in section 4, resulting in a plausible strategy for termination checking without considering the calling context. The following lemma formally states this syntactic property of local closures.

Lemma 2 (Local Closure Property)**.** *Let e be an open expression well-typed with respect to a library L, and let (σ, L') be a closing environment. Let $(\ell : v) \in LocalClosure(L, L', e, \sigma)$, where ℓ is the static label of v. There exists a lambda term $\lambda x.g\ \bar{y}$ that is a sub-expression of e such that $\exists((\lambda x.g\ \bar{w})\ u') \in LocalCall(L, L', e, \sigma)$ and $v = \lambda x.g\ \bar{w}$ or $v \in LocalClosure(L, L', body(g), \sigma_{param}((\lambda x.g\ \bar{w})\ u'))$.*

An immediate corollary of the above lemma is that if $(\ell : v)$ is a local closure of e w.r.t any closing environment then it is a closure of a lambda term which is a sub-expression of e or a sub-expression of the body of some function in L. In other words, the definition or source code of all local closures are available in the library or in the expression under analysis.

As an illustration, consider the body of the function f shown in Fig. 6: $g\ (\lambda x.h\ x)$. (Since we do not allow direct calls outside of the lamdba terms, g is a syntactic sugar for a lambda term $\lambda y.g\ y$.) The evaluation of the closed expression $g\ (\lambda x.h\ x)$ is non-terminating due to the infinite reduction sequence: $g\ (\lambda x.h\ x) \rightsquigarrow^* (\lambda x.h\ x)\ (\lambda y.g\ (\lambda x.h\ x)) \rightsquigarrow^* g\ (\lambda x.h\ x) \rightsquigarrow^* \cdots$. By theorem 1, there must exist a function, in this case g, such that the length of every sequence of incomplete calls that invoke a closure whose target is g is unbounded. The infinite sequence $S : g\ (\lambda x.h\ x) \rightsquigarrow^* g\ (\lambda x.h\ x) \rightsquigarrow^* \cdots$ serves as a witness for the fact that there does not exist an upper bound on the length of all such sequences (as it doesn't even exist for a single sequence).

We are not aware of similar theorems for higher-order programs that derives sound and complete conditions for non-termination based on the locality of the closures that are invoked. The complete proof of the theorem is shown in Appendix A. At a high-level, we prove this by showing that if there exists a bound n on all such call sequences stated in the the Theorem 1, then the recursions in the functions in the library L can be embedded into a strongly normalizing fragment of lambda calculus, by unfolding the functions upto a depth of n. For this purpose, we extend the Tait (1967) and Girard (1990) proof of strong normalization of simply-typed lambda calculus to functions with recursion that are known to be terminating.

Below we use the Theorem 1 to deduce sufficient conditions (proof obligations) for checking termination modulo callbacks of open expressions.

3.2 Obligations for Termination Modulo Callbacks

Based on Theorem 1, we now present the proof obligations that ensure termination modulo callbacks of any open expression belonging to a library L. We present the obligation checked by our algorithm in two steps in order to throw light on the trade-offs involved in the proof obligation and to throw light on the correctness. We first introduce a weak obligation, which is an intermediate condition that helps derive the second obligation namely the strong obligation. Our system only uses the strong obligation to establish termination modulo callbacks.

The first obligation, referred to as weak obligation, requires that there exist a bound on every call sequence of the form stated by Theorem 1, and thereby ensures termination. For this purpose, we associate a *measure* or *ranking function* with every function f in the library. The measure defines a well-founded ordering on the set of assignment of expressions to the parameters of f. We assert that every time a call local to *body*(f) that applies f is invoked, the measure decreases. (We assume that the measure function is oblivious to the marks on the lambda terms, which is only an instrumentation artifact.)

Definition 3.3 (Weak Termination Obligation). Let e be an expression well-typed with respect to a library L. There exists a measure function $\mathcal{M}_f : (\mathit{param}(f) \mapsto \mathit{Expr}) \to \mathbb{N}$ for every function f belonging to L, and a measure function $\mathcal{M}_e$ for e such that the following holds.

For all $e' \in \{e\} \cup \{\mathit{body}(f) \mid f \in L\}$, for every closing environment (σ, L') of (e', L),

$$((\lambda x.f\ \bar{w})\ u) \in \mathit{LocalCall}(L, L', e', \sigma) \Rightarrow \mathcal{M}_f(\sigma_{\mathit{param}}((\lambda x.f\ \bar{w})\ u)) < \mathcal{M}(\sigma)$$

where $\mathcal{M}$ is $\mathcal{M}_f$ if e' is *body*(f) for some f, and $\mathcal{M}_e$ otherwise.

Theorem 3 (Soundness of Weak Obligation)**.** *Let e be an open expression well-typed with respect to a library L. If the weak termination obligation given by Definition 3.3 holds for L and e, then e terminates modulo callbacks.*

The formal proof of soundness is presented in Appendix B. Intuitively, the reason why this rules out call sequences of the form stated in Theorem 1 is as follows. If the above obligation holds, every local call c of $e\ \sigma$ invoking a closure whose target is f must have a measure strictly smaller than the measure of $e\ \sigma$. Similarly, every call local to it, invoking a closure whose target is f, must also have a strictly smaller measure compared to the measure of c. Thus, any chain of local calls starting from $e\ \sigma$ invoking a given function f must have length at most $\mathcal{M}_e(\sigma)$.

The weak obligation is not complete for termination modulo callbacks. That is, if the obligation fails for a library L, it may still be possible for an expression well-typed with respect to L to be terminating modulo callbacks. Nevertheless, we have a weaker completeness guarantee stated below. In simple words, the following theorem states that if the weak obligation fails then there exist expressions of the library that are terminating modulo callbacks, yet when composed with each other will lead to an expression that is no longer terminating modulo callbacks. This suggests that it is necessary to ensure this obligation for a library L if one has to guarantee termination under all possible clients that are terminating modulo callbacks.

Theorem 4 (Weak Completeness of Weak Obligation)**.** *If the weak termination obligation does not hold for a library L, there exists a library $L \cup L'$ and expressions e and e' well-typed under the library $L \cup L'$ such that e and e' are terminating modulo callbacks but $e\ e'$ is not terminating modulo callbacks.*

We refer to this obligation as weak as it is quite permissive in comparison to the conditions presented later. However, the obligation is also difficult to check (precisely) without knowledge of the definition of callbacks in the closing environment. This is because, while we only need to check calls local to $\mathit{body}(f)$, those calls could be made within callbacks of f (through a local closure passed to the callback). Hence, it is difficult to know the arguments with which the local closures would be invoked under a closing environment, and even whether it would be invoked at all. In other words, it is difficult to know the value u used in the definition of the obligation.

One conservative way to soundly guarantee this obligation is to determine all closures local to f that could ever be applied during the evaluation of $\mathit{body}(f)$ (under some closing environment), and establish that the measure decreases irrespective of the argument on which it is applied. This is the idea behind the strong termination obligation described below.

Definition 3.4 (Strong Termination Obligation). Let e be an expression well-typed with respect to a library L. There exists a measure function $\mathcal{M}_f : (\mathit{param}(f) \mapsto \mathit{Expr}) \mapsto \mathbb{N}$ for every function f in L and a measure function $\mathcal{M}_e$ for e such that the following holds.

For all $e' \in \{e\} \cup \{\mathit{body}(f) \mid f \in L\}$, for any closing environment (σ, L') of (e', L) and for all lambda term $\ell : (\lambda x.g\ \bar{y})$ that is a *sub-expression* of e',

$$\forall u.(\ell : \lambda x.g\ \bar{w}) \in \mathit{LocalClosure}(L, L', e', \sigma) \wedge \mathit{Applicable}(L, L', e', \sigma, \ell, u) \implies$$
$$\mathcal{M}_g([\mathit{param}(g) \mapsto \bar{w}[u/x]]) < \mathcal{M}(\sigma)$$

where $\mathcal{M}$ is $\mathcal{M}_f$ if e' is $\mathit{body}(f)$ for some f, and $\mathcal{M}_e$ otherwise.

The operation *Applicable*$(L, L', e', \sigma, \ell, u)$ returns true if a closure $\ell : \lambda x.g\ \bar{w}$ is applied with argument u during the evaluation of e' under the closing environment (σ, L'). Note that when *Applicable*$(L, L', e', \sigma, \ell, u)$ *iff* $((\ell : \lambda x.g\ \bar{w})\ u) \in$ *LocalCall*(L, L', e', σ), then the strong termination obligation corresponds exactly to the weak version. However, the strong termination obligation can adjust the precision of the analysis used to compute *Applicable* as needed. The soundness of the strong termination obligation is incumbent only on the requirement that this operation conservatively returns true on all lambda terms that may be invoked before e' completes (for example, by always returning true).

Theorem 5 (Soundness of Strong Termination Obligation)**.** *Let e be an open expression well-typed with respect to a library L. If the strong termination obligation given by Definition 3.4 holds for L and e then e terminates modulo callbacks.*

We formally detail the proof of this theorem in Appendix B. In practice, a good first approximation of *Applicable*$(L, L', e', \sigma, \ell, u)$ can be given by the function *Invocable*(L, e', ℓ) which returns true if a closure labeled ℓ is applied during the evaluation of e' under some closing environment. We also use any available user-provided preconditions on the possible values of the parameter u (e.g. expressed using a contract on the lambda term labelled ℓ). The purpose of this operation is to relieve the algorithm from having to consider lambda terms of e' that cannot be invoked before e' completes evaluation (under any closing environment). Obviously, such lambda terms cannot result in local calls of e'. We will shortly provide a semantic definition of *Invocable*(L, e', ℓ). In our implementation we use a static analysis to approximate this semantic definition.

The strong termination obligation is more amenable to modular verification. Firstly, note that σ is an assignment of parameters of f to some values. To consider all possible values of the parameters, we could leave the parameters as unconstrained, free variables. Secondly, the obligation only considers closures of a sub-expression $\ell : e_\lambda$ of *body*(f) or e. Thus the definition of all relevant closures that need to be checked is available in the body of f (or e). Assuming that the measures of each function are already known (either inferred automatically or provided by the user), the only unknown in the obligation is the constraint on the captured values of the closures corresponding to e_λ, i.e, the substitution σ' from the free variables of the lambda to values that is possible at the label ℓ. This is constrained by the path starting from the entry point of f leading to the lambda term $\ell : e_\lambda$.

Thus, to check the strong obligation it suffices to estimate the substitutions reachable at the point of lambda creations in the functions in L. In essence, the strong obligation has reduced termination modulo callbacks of an open library L to the familiar problem of determining reachable states within each function, albeit in the presence of first-class functions. To be able to constrain the reachable states modularly and precisely on open programs, we turn to higher-order contracts. The following section details our algorithm for checking the strong termination obligation using modular, property verifiers for higher-order programs.

Another interesting aspect of the strong obligation is that it only deals with closure creation sites and establishes properties that must hold for all arguments with which the closures are invoked (except for *Invocable* function which could analyze application sites, but is not required to). Any incompleteness due to this creation site reasoning can be circumvented through additional user annotations, as it is always possible for a user to express the properties that hold at the application sites of lambda terms as preconditions on the arguments of the lambda and captured variables. Such preconditions would be taken into account by our algorithm presented in the following section. Below we present a sound semantic definition of the *Invocable* function, which is used .

3.3 Semantic Definition of Invocable

A straightforward and precise way to define *Invocable* is by defining $\mathit{Invocable}(L, e, \ell)$ to be true iff there exists a closing environment (σ, L') such that $(\ell : v)\ u \in \mathit{LocalCall}(L, L', e, \sigma)$. This definition of *Invocable* makes the strong and the weak obligations almost identical, and thereby also reintroduces the non-modularity problem we sought to address in the weak obligation. For instance, the call $(\ell : v)\ u$ may happen within a callback of e whose implementation depends on the closing environment. Therefore, we resort to an over-approximate yet sound definition of *Invocable* described below that conservatively assumes that a callback may invoke any closure (local to e) that is *accessible* from its arguments.

Given an expression e and library L, we define a label ℓ belonging to L or e as invocable iff either (a) there exists a call site in the library L or e that applies a local closure (referred to as *LocalCallSite*) labeled ℓ, or (b) a closure labeled ℓ is local to another local closure labeled ℓ' passed as an argument (or captured by an argument passed) to a call site in the library L or e that is a callback under some closing environment.

Definition 3.5 (Invocable). Let e be an expression well-typed with respect to a library L. Let Label_L denote the labels of e and expressions of the library L.

$$\mathit{Invocable}(L, e, \ell) \ \mathit{iff}\ \exists\{r, u\} \subseteq \mathit{Value}.((\ell : r)\ u) \in \mathit{LocalCallSite}(L, e) \vee (\ell : r) \in \mathit{AccessibleClo}(L, e)$$

$$\begin{aligned}
\text{where } \mathit{LocalCallSite}(L, e) &= \{(\ell : v\ u) \mid \ell \in \mathit{Label}_L \wedge \\
&\qquad \exists \text{ closing environment } (\sigma, L').\ (\ell : v\ u) \in \mathit{LocalCall}(L, L', e, \sigma)\} \\
\mathit{NonLocalCallSite}(L, e) &= \{(\ell : v\ u) \mid \ell \in \mathit{Label}_L \wedge \\
&\qquad \exists \text{ closing environment } (\sigma, L').\ (\ell : v\ u) \notin \mathit{LocalCall}(L, L', e, \sigma)\} \\
\mathit{AccessibleClo}(L, e) &= \textstyle\bigcup_{v \in \mathit{EscapingClo}(L,e)} \mathit{LocalClosure}(L, v) \\
\mathit{EscapingClo}(L, e) &= \textstyle\bigcup_{(v\ u) \in \mathit{NonLocalCallSite}(L,e)} \mathit{LocalCaptures}(L, e, v) \\
\mathit{LocalCaptures}(L, e, v) &= (\{v\} \cup \{r \mid \text{r is captured by } u\}) \cap \mathit{LocalClosure}(L, e) \\
\mathit{LocalClosure}(L, e) &= \{v \mid \exists \text{ closing environment } (\sigma, L').\ v \in \mathit{LocalClosure}(L, L', e, \sigma)\}
\end{aligned}$$

In the above definition, $\mathit{LocalClosure}(L, e)$ denotes the set of closures local to e under some closing environment. $\mathit{LocalCallSite}(L, e)$ denotes the call sites in L or e that apply a local closure under some closing environment and $\mathit{NonLocalCallSite}(L, e)$ the call sites that apply some non-local closure. $\mathit{EscapingClo}(L, e)$ is the set of local closures that are passed as arguments to non-local call sites, or are captured by arguments passed to them. $\mathit{AccessibleClo}(L, e)$ is the set of closures that are local to $\mathit{EscapingClo}(L, e)$. This set captures the set of closures that may be invoked by a callback under some closing environment.

To compute *Invocable* using the above definition, we need three pieces of information. (a) We need the set of closures that are local to e, under some closing environment. (b) We need to know if a call site in L or e could be a callback or could be a local call to e. (c) We need the set of local closures that are passed as (or are captured by) arguments to the call sites in L and e. As in the case of the strong obligation, to compute this information it suffices to determine reachable substitutions (or states) at the call sites and lambda creation sites in L and e. Our algorithm determines the above pieces of information using a over-approximate static control-flow analysis that treats callbacks as uninterpreted functions with arbitrary

$$\begin{array}{ll} d \in \mathit{Tids} & \mathit{Datatype\ identifiers} \\ C \in \mathit{Cids} & \mathit{Datatype\ constructor\ identifiers} \end{array}$$

$$\lambda x.\textbf{req}(c); f\ \bar{y} \in \mathit{Lambda} \qquad \textbf{req}(e); e \in \mathit{Expr} \qquad \textbf{req}(f, e) \in \mathit{Expr} \qquad \mathit{Tdef} ::= \textbf{type}\ d := C\ \tau\ \cdots\ C\ \tau$$

$$L \in \mathit{Library} ::= 2^{\mathit{Fdef}} \cup 2^{\mathit{Tdef}}$$

Fig. 7. Selection of syntax extensions to the new language.

$$\textsc{Contract}\ \frac{\Gamma \vdash c : \mathsf{Boolean} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \textbf{req}(c); e : \tau} \qquad \textsc{Domain}\ \frac{\Gamma \vdash f : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{req}(f, e) : \mathsf{Boolean}}$$

$$\textsc{Datatype}\ \frac{\begin{array}{c} \textbf{type}\ d := C_1\ \tau_1\ \cdots\ C_n\ \tau_n \in L \\ \exists i \in [1, n], e \in \mathit{Expr}.\, \Gamma \vdash e : \tau_i \land (C_i\ e) \in \mathit{Value} \\ d \text{ only appears in positive position in } \tau_1, \cdots, \tau_n \end{array}}{\Gamma \vdash \textbf{type}\ d := C_1\ \tau_1,\ \cdots,\ C_n\ \tau_n : \mathsf{Unit}}$$

Fig. 8. Selection of static typing rules of the extended language with respect to a library L.

behavior. However in principle, one can use a higher-order (state or property) reachability analysis (such as symbolic execution) to determine this information.

4 TERMINATION VERIFICATION ALGORITHM

In this section, we discuss our algorithm for verifying the strong termination obligation discussed in the previous section. Termination checking is performed in the context of a verification framework and relies on higher-order contracts for effectiveness. At a high level, the verification and termination procedures of a library proceed as follows:

(1) First establish that all contracts defined within library functions hold (assuming termination modulo callbacks of all library functions).
(2) Next, check that the library satisfies the strong termination obligation.
(3) If all checks in steps 1 and 2 were successful, output that the library is verified and terminates modulo callbacks.

Note that neither the soundness of verification nor that of termination can be claimed unless both procedures succeed. Although this dependency may seem to imply cyclic reasoning, we show in the following that the procedure is sound.

4.1 Higher-Order Contracts and Datatypes

We extend the toy language presented previously to support if-expressions, type polymorphism, recursive datatypes and higher-order contracts. We show a selection of the syntax extensions in Fig. 7. For clarity, we sometimes omit the **req**(c) contract from lambdas when it is not relevant. Fig. 8 presents a selection of the typing rules for the extended language. Note that the proposed type system ensures the existence of datatype normal forms by ensuring recursive datatypes admit a term in normal form and appear only in

$$\textit{(Evaluation contexts for contracts)} \qquad \mathcal{C} \in \mathit{EContext} = \textbf{req}(\mathcal{C}); e \mid \textbf{req}(\mathcal{C}, e) \mid \textbf{req}(v, \mathcal{C})$$

$$\begin{array}{lll} \text{CONTRACT-1} & \text{CONTRACT-2} & \text{REQ-F1} \\ \textbf{req}(\textbf{true}); e \rightsquigarrow e & (\lambda x.\textbf{req}(c); f\ \bar{w})\ v \rightsquigarrow \textbf{req}(c[v/x]); (\lambda x.f\ \bar{w})\ v & \textbf{req}(\lambda x.\textbf{req}(c); f\ \bar{w}, v) \rightsquigarrow c[v/x] \end{array}$$

Fig. 9. Small-step operational semantics of expressions containing contracts and domain requirements.

positive positions in their constructor arguments. This restriction preserves the soundness of Theorems 3, 4, and 5 in the presence of recursive datatypes. The proofs of these theorems can be expanded to the extended language (Girard 1971),(Girard 1990). The extended language follows the expected operational semantics. In this section, we assume the existence of a special Boolean datatype with constructors **true** and **false** along with the usual operators (definable as non-recursive functions).

Contracts are introduced into our language through the $\textbf{req}(c); e$ expression. The small-step operational semantics of contracts can be found in Fig. 9. Contracts within lambdas effectively transform these into partial functions. In order to specify/reason about the domain of these partial functions, we provide the $\textbf{req}(f, e)$ construct that evaluates to true iff the function f is defined on e. We are interested in verifying open programs and it is therefore useful to be able to specify assumptions at function boundaries. Given an expression $\textbf{req}(c); e$, we are generally only interested in closing environments (σ, L') such that $c\ \sigma \rightsquigarrow^* \textbf{true}$ and we thus verify properties of $e\ \sigma$ under this assumption. Note that when considering evaluation and/or verification, we can always assume an expression e is of the shape $\textbf{req}(c); e'$ as wrapping e in $\textbf{req}(\textbf{true}); e$ is idempotent with respect to evaluation. Further note that the $\textbf{req}(c); e$ construct can also be used to specify postconditions, as in the following example where *pre* and *post* correspond to the usual notions of pre- and postcondition for function f.

```
def f(x): Nat ⇒ Nat := req(pre(x)); (λy.req(post(x,y)); id y) e        def id(x): Nat ⇒ Nat := x
```

Validity of contracts is independently established by a verification framework. This operation corresponds to step 1 of the high-level procedure presented above. We say an expression $\textbf{req}(c); e$ is *verified* iff

(1) it is well-typed, and
(2) for all closing environment (σ, L') such that $c\ \sigma \rightsquigarrow^* \textbf{true}$ and evaluation of $e\ \sigma$ is terminating, for all $\ell : \textbf{req}(c_\ell); e_\ell$ encountered during evaluation of $e\ \sigma$ where $\ell \in L$ (*i.e.* $e\ \sigma \rightsquigarrow^* \mathcal{C}[\ell : \textbf{req}(c_\ell); e'_\ell]$), we have $c_\ell \rightsquigarrow^* \textbf{true}$.

Note that nothing is known about the evaluation of c_ℓ in cases where $e\ \sigma$ is non-terminating. We say library L is verified when for all $f \in L$, $\mathit{body}(f)$ is verified. It is often the case that verification procedures rely on termination for soundness when performing inductive reasoning over function recursions. The termination procedure we will be presenting relies on contracts having been checked, thus leading to a bidirectional dependency between contracts and termination. This may seem like circular reasoning, however it has been shown in (Giesl 1997) that one can soundly rely on properties that have been established through induction over function recursions during termination verification as long as the full library is checked. In other words, given a verified library L, it is sound to prove termination of L while relying on all $\textbf{req}(c); e$ encountered during evaluation to be such that $c \rightsquigarrow^* \textbf{true}$. In the rest of this section, we will therefore rely on contracts always evaluating to true when establishing our termination conditions.

4.2 Checking the Strong Termination Obligation

In practice, checking the strong termination obligation directly by relying on a verification framework is awkward due to potentially uncomputable ranking functions. We address this issue in the following section and discuss a more tractable way of establishing the strong termination obligation through the verification framework. This procedure corresponds to step 2 of the high-level algorithm presented earlier.

Let us start by presenting the notion of *dependency graph*. Given a library L, we construct a labeled directed graph called dependency graph whose nodes correspond to the functions in the program. The edges of the graph are constructed as follows. For every function $f \in L$, the graph contains edges from f to the functions invoked in the body of the lambda expressions contained in $body(f)$. Each edge is annotated by the label of the lambda expression that resulted in the edge. Unlike a conventional call-graph, the dependency graph does not have any edges connecting the indirect call sites to their targets. Thus, constructing a dependency graph requires only a *syntactic* analysis of a library.

Definition 4.1 (Dependency Graph). Let L be a library. Define a *dependency graph* $G_L = (L, E)$ where $E = \{(f, \ell, g) \mid (\lambda x.\, \ell : g\ \bar{y}) \sqsubseteq body(f)\}$.

Let us now introduce the notion of *path condition*. The path condition of a sub-expression $(\ell : e_\ell) \sqsubseteq e$ corresponds to a boolean expression that must evaluate to **true** on all paths where evaluation can reach label ℓ. The path condition provides us with a condition under which properties about ℓ will become relevant.

Definition 4.2 (Path Condition). Let $e \in Expr$ and $(\ell : e_\ell) \sqsubseteq e$. We say c is a *path condition* for label ℓ if for all closing environments (σ, L') such that $e\ \sigma \rightsquigarrow^* \mathcal{C}[\ell : e'_\ell]$, we have $c\ \sigma \rightsquigarrow^*$ **true**.

Note that there are many path conditions for a single label-expression pair (the most trivial being **true**). When dealing with higher-order functions, computing the most precise path condition of a given label is undecidable. However, higher-order contracts provide us with a flexible tool to efficiently construct precise path conditions in the presence of higher-order functions. Indeed, given a lambda $(\lambda x.\textbf{req}(c); \ell : f\ w) \sqsubseteq e$ where e is verified, we can add c to the path condition of ℓ (by definition of verification). Let us therefore define $PathCond$ as

$$
\begin{array}{lll}
\multicolumn{3}{l}{PathCond(e, \ell) := e \text{ in}} \\
\quad \textbf{if } (c)\ [\![e_t]\!]_\ell \textbf{ else } e_e & := & c \wedge PathCond(e_t, \ell) \\
\quad \textbf{if } (c)\ e_t \textbf{ else } [\![e_e]\!]_\ell & := & \neg c \wedge PathCond(e_e, \ell) \\
\quad \textbf{req}(c); [\![e']\!]_\ell & := & c \wedge PathCond(e', \ell) \\
\quad \lambda x.\textbf{req}(c); [\![f\ e']\!]_\ell & := & c \\
\quad e \quad \textbf{if} \quad [\![child_i(e)]\!]_\ell & := & PathCond(child_i(e), \ell) \\
\quad \ell : e & := & \textbf{true}
\end{array}
$$

where $[\![e]\!]_\ell$ *iff* $(\ell : e') \sqsubseteq e$ and $child_i(e)$ is the i-th (direct) child of expression e. Assuming (e, L) was *soundly* verified, it is clear that $PathCond(e, \ell)$ is a path condition.

We define a cycle $C : f, f_1, \cdots, f_n, f$ in the graph as a path that begins and ends at the same vertex. For every edge (f, ℓ, g) that belongs to some cycle in the graph, we establish that for all closing environments

(σ, L') we have

$$(\mathit{PathCond}(\mathit{body}(f), \ell)\ \sigma \rightsquigarrow^* \textsf{true}) \wedge \mathit{Invocable}(L, \mathit{body}(f), \ell) \implies \mathcal{M}_g([\mathit{param}(g) \mapsto (\bar{y}\ \sigma)]) < \mathcal{M}_f(\sigma) \quad (1)$$

where $(\lambda x.\ \ell : g\ \bar{y}) \sqsubseteq \mathit{body}(f)$. As $g\ \bar{y}$ appears within a lambda, it may seem necessary to reason about unknown targets when establishing this property. However, remember that when the lambda has the shape $\lambda x.\textsf{req}(c); \ell : g\ \bar{y}$, the condition c will appear within $\mathit{PathCond}(\mathit{body}(f), \ell)$ and can be used to constrain appearances of the lambda parameter x in $\bar{y}$. Notice that in the above definition we only consider cycles in the dependency graph (whereas the strong termination obligation considers all local closures). This is sufficient because given two functions f and g such that f precedes g in a topological ordering of the call-graph then we impose that $\mathcal{M}_f > \mathcal{M}_g$. Note that establishing that $\mathit{PathCond}(\mathit{body}(f), \ell)\ \sigma \rightsquigarrow^* \textsf{true}$ will be discharged to the verification framework.

Lemma 6. *If (e, L) is verified, and property (1) holds for all $(f, \ell, g) \in E$ and closing environments (σ, L'), then the strong termination obligation holds for expression e and library L.*

Proof. By applying results from (Giesl 1997), we can assume that verification was sound while proving termination modulo callbacks. As above, we impose that $\mathcal{M}_e > \mathcal{M}_f$ for all $f \in L$. If we take *Applicable* to be as precise as possible, then by definition of a path condition, $\mathit{Applicable}(L, L', \mathit{body}(f), \sigma, \ell, u)$ implies that $\mathit{PathCond}(\mathit{body}(f), \ell)\ \sigma'[x \mapsto u] \rightsquigarrow^* \textsf{true}$ for some σ' corresponding to to the arguments given to f when the lambda was added to $\mathit{LocalClosure}(L, L', e', \sigma)$. As discussed previously, $\mathit{Applicable}(L, L', \mathit{body}(f), \sigma, \ell, u)$ further implies $\mathit{Invocable}(L, \mathit{body}(f), \ell)$. Finally, from our language syntax and operational semantics, we know that lambda $\bar{y}$ can only consist of either values. Hence, $[\mathit{param}(g) \mapsto (\bar{y}\ \sigma)]$ subsumes $[\mathit{param}(g) \mapsto \bar{w}[u/x]]$ and property (1) therefore implies the strong termination obligation. □

As we have a verification framework at our disposal, we want to cast the above check into a higher-order contract checking problem. Let us start by considering the measure functions $\mathcal{M}_f$ and $\mathcal{M}_g$. These can correspond to arbitrary (potentially non-computable) functions taking a subtitution as input. However, the verification framework can only reason about constructs that are defined in the language it supports. This observation implies a certain limitation to the set of measures that can be used to prove termination when relying on an out-of-the-box verification procedure. In practice, this issue is alleviated by two important points.

(1) The measure functions $\mathcal{M}_f$ and $\mathcal{M}_g$ can map into any well-founded domain. Indeed, as mentioned in Section 3, the measure serves only to define a well-founded ordering on the sets of parameter assignments. By allowing more complex well-founded domains (such as tuples with lexicographic orderings for example), we can significantly simplify the measure function itself.

(2) Many well-founded domains are supported by verification frameworks. Reasoning about lexicographic orderins, as well as sets with the subset relation, datatypes with structural inclusion, and lambdas with inclusion are all well-founded domains that have some support in various verification frameworks.

Consider now a well-founded domain $(D, \prec)$ that is supported by the verification framework. We associate to each function $\textsf{def}\ f(\bar{x}) : \tau_1 \Rightarrow \tau_2 := e \in L$ a terminating function $\textsf{def}\ \textsf{measure}_f(\bar{x}) : \tau_1 \Rightarrow D := m_f$. We then use the verification procedure to establish that for every edge (f, ℓ, g) that belongs to a cycle in the dependency graph, for all closing environments (σ, L'),

$$\mathit{Invocable}(L, \mathit{body}(f), \ell) \implies (\mathit{PathCond}(\mathit{body}(f), \ell) \implies \mathsf{measure}_g(\bar{y}) \prec \mathsf{measure}_f(\bar{x}))\ \sigma \rightsquigarrow^* \textbf{true} \quad (2)$$

where $(\lambda x.\, \ell : g\ \bar{y}) \sqsubseteq \mathit{body}(f)$. Interestingly, when no analysis to approximate *Invocable* is performed and we conservatively let $\mathit{Invocable}(L, \mathit{body}(f), \ell)$ return true, this property corresponds exactly to the property that would need to be checked by the verification framework if we were to inject the contract $\textbf{req}(\mathsf{measure}_g(\bar{y}) \prec \mathsf{measure}_f(\bar{x}))$ into the lambda $\lambda x.g\ \bar{y}$. Note that we can still enforce $\mathsf{measure}_f > \mathsf{measure}_g$ given the topological sort by ensuring $(D, \prec)$ is a lexicographic ordering that guarantees this holds.

Lemma 7. *If property (2) holds for all $(f, \ell, g) \in E$ and closing environments (σ, L'), then there exists a set of measure functions $\{\, \mathcal{M}_f : (\mathit{param}(f) \mapsto \mathit{Expr}) \mapsto \mathbb{N} \mid f \in L \,\}$ such that property (1) holds for all $(f, \ell, g) \in E$ and closing environments (σ, L').*

Proof. Given $f \in L$ and closing environment (σ, L') with $\mathsf{measure}_f(\mathit{param}(f))\ \sigma \rightsquigarrow^* d_\sigma \in D$, let us start by defining $\mathit{Less}(\sigma, f) = \{\, d \in D \mid d \prec d_\sigma \,\}$. We then let $\mathcal{M}_f(\sigma) = |\mathit{Less}(\sigma, f)|$. Given $(f, \ell, g) \in E$ where $(\lambda x.\, \ell : g\ \bar{y}) \sqsubseteq \mathit{body}(f)$, clearly if $(\mathsf{measure}_g(\bar{y}) \prec \mathsf{measure}_f(\bar{x}))\ \sigma \rightsquigarrow^* \textbf{true}$, then $\mathsf{measure}_g(\bar{y})\ \sigma \rightsquigarrow^* d_g$ and $\mathsf{measure}_f(\bar{x})\ \sigma \rightsquigarrow^* d_f$ where $d_g \prec d_f$. Hence, by definition of *Less*, we have $\mathit{Less}([\mathit{param}(g) \mapsto (\bar{y}\ \sigma)], g) \subset \mathit{Less}(\sigma, f)$ and therefore $\mathcal{M}_g([\mathit{param}(g) \mapsto (\bar{y}\ \sigma)]) < \mathcal{M}_f(\sigma)$. □

Theorem 8. *Given a verified library L and expression e that is verified with respect to L, if for all (f, ℓ, g) in the dependency graph of L and all closing environments (σ, L') we have*

$$\mathit{Invocable}(L, \mathit{body}(f), \ell) \implies (\mathit{PathCond}(\mathit{body}(f), \ell) \implies \mathsf{measure}_g(\bar{y}) \prec \mathsf{measure}_f(\bar{x}))\ \sigma \rightsquigarrow^* \textbf{\textit{true}}$$

where $(\lambda x.\, \ell : g\ \bar{y}) \sqsubseteq \mathit{body}(f)$, then e terminates modulo callbacks.

Proof. Follows from Lemmas 6 and 7, as well as Theorem 5. □

Corollary 9. *Given a verification procedure for partial correctness that assumes induction hypothesis over function recursions, we can construct a verification procedure that additionally proves termination modulo callbacks.*

5 EXPERIMENTAL EVALUATION

We have implemented our termination checker as part of a verification framework for a pure higher-order functional subset of Scala with higher-order contracts. Our implementation features automated measure, ranking function and inductive invariant inference, thus presenting a relatively high level of automation. Our implementation also enables users to provide certain hints to the termination checker, either in the form of contracts or relevant measures. We have evaluated our procedure over a set of benchmarks totaling $>$ 10k LoC and show the running time on a few selected benchmarks in Table 10.

Operation	LoC	Annot.	Time (s)	Operation	LoC	Annot.	Time (s)
`NNF`	101	0	18.0	`RedBlackTree`	100	0	9.0
`Ackermann`	10	0	1.4	`McCarthy91`	10	2	0.7
`QuickSort`	62	0	8.5	`MergeSort`	63	1	7.5
`QuickSortPar`	51	2	6.5	`Patterns`	25	0	5.5
`HOTransformations`	41	0	5.5	`AmortizedQueue`	125	0	3.3
`Heaps`	147	0	11.0	`ListOperations`	104	0	12.0
`ConcRope`	468	0	48.0	`ConcTree`	320	0	26.0
`ConstantProp`	268	0	25.0				
– Okasaki Data Structures –							
`BinomialHeap`	186	2	8.5	`BottomUpMergeSort`	123	2	5.5
`Deque`	241	3	4.5	`LazySelectionSort`	65	0	1.5
`RealTimeQueue`	76	0	0.1	`LazyNumericalRep`	157	2	1.0
– Streams and Lazy Trees –							
`CyclicFibStream`	46	0	2.0	`CyclicHammingStream`	74	0	2.5
`FiniteStreams`	42	2	1.2	`InfiniteStreams`	63	0	0.9
`LazyTree`	37	1	2.1				
– Dynamic Programming –							
`PackratParsing`	135	2	4.0	`HammingMemoized`	66	1	5.0
`Knapsack`	70	0	3.5	`Viterbi`	119	3	1.2
`Regression Tests`					7525	0	188.0
Total					10920	25	419.0

Fig. 10. Summary of evaluation results for termination checker, featuring lines of code (including annotation lines), the number of necessary annotation lines, as well as running time of our tool.

Note that some involved termination proofs require certain hints for termination to go through (recorded in the (Annot.) column). It is important however to realize that these constitute a limitation of our inference engine and not the termination procedure itself.

The `NNF` benchmark deals with bringing some formula into negative normal form. The tricky part stems from `nnf`$(l \implies r)$ calling `nnf`$(\neg l \vee r)$ such that the structural size of the argument to `nnf` is only guaranteed to decrease after the *third* successive recursive call! The inferred ranking function must hence depend on the type of the input formula. `Patterns` comes from (Xi 2001) (Figure 8) and implements regular expression matching with continuations. The quicksort implementation discussed in (Giesl et al. 2006; Kuwahara et al. 2014; Xi 2001) can be proved with minimal annotation burden (note that this annotation burden improves over that of these techniques). Both this version and a simpler one (automatically proved) fail to be proved terminating by the AProVE termination checker. Our tool can also handle all benchmarks collected and presented in (Kuwahara et al. 2014). Certain benchmarks require one or two lines of annotations, however these mostly stem from us proving the more general termination property of the corresponding open program. The `ConcRope` and `ConcTree` benchmarks correspond to certain complex operations in the Scala data parallel library (Prokopec and Odersky 2015). Finally, the `ConstantProp` benchmark corresponds to a lattice-based constant propagation procedure that requires complex reasoning over structural inclusion for mutually recursive data types. (Some of the benchmarks presented here were taken from a previous work on verification of resource properties (Madhavan et al. 2017). However, the approach relies on a separate tool for checking termination of the benchmarks, which is required for its soundness. It also makes the assumption that the targets of the indirect calls are available at the time of the analysis.)

```
abstract class Stream
case class SNil() extends Stream
case class SCons(x: BigInt, tailFun: () ⇒ Stream) extends Stream {
  lazy val tail = tailFun()
}

def zipWithFun(f: (BigInt, BigInt) ⇒ BigInt, xs: Stream, ys: Stream): Stream = (xs, ys) match {
  case (SCons(x, _), SCons(y, _)) ⇒ SCons(f(x, y), () ⇒ zipWithFun(f, xs.tail, ys.tail))
  case _ ⇒ SNil()
}

val fibstream: SCons = SCons(0, () ⇒ SCons(1,
  () ⇒ zipWithFun(_ + _, fibstream, fibstream.tail)))
```

Fig. 11. Implementation of a cyclic Fibonacci stream.

We have also evaluated our approach on various lazy data structures such as streams and lazy trees. Even though Scala uses call-by-value evaluation by default, such data structures can be encoding using closures and recursive datatypes whose fields are closure, as in the example shown in Fig. 1. (It is to be noted that memoization or caching performed by lazy evaluation does not affect termination of programs but only its performance, therefore we do not consider the memoization aspect of lazy evaluation in our termination verification.) Combining termination checking with powerful higher-order contracts has enabled us to verify the termination of complex lazy data structures as described in (Okasaki 1998). The *Invocable* analysis has also enabled us to prove termination of functions producing cyclic (infinite) streams such as Fibonacci (see Fig. 11) or Hamming streams and certain stream operations such as map, drop, concatenation, etc.

6 RELATED WORK

Most approaches to termination checking of higher-order programs first construct an over-approximation of the call-graph to infer potential indirect call targets, and then apply some technique from first-order termination checking on the resulting first-order call-graph. First-order termination techniques such as Size-Change Termination (SCT) (Lee et al. 2001) and termination analysis of Term Rewritting Systems (TRS) (Giesl et al. 2004) have successfully been applied to a wide range of higher-order programs (Giesl et al. 2011). However, although the underlying first-order techniques are powerful and well-understood, it remains difficult to achieve both precision and modularity when computing an over-approximate call-graph. Furthermore, the reduction from a higher-order program to a first-order one relies on certain assumptions concerning input first-class functions as these cannot be precisely modeled in the first-order world. Our technique enables precise reasoning about the higher-order interfaces of libraries by avoiding approximations when dealing with first-class functions.

A more recent approach has emerged based on binary reachability analysis (Kuwahara et al. 2014; Ledesma-Garza and Rybalchenko 2012). These techniques forgo the creation of some over-approximate call-graph and rely instead on reachability analysis to establish the relevant proof obligations. As in our case, reachability analysis can tie in to existing verification procedures (Kuwahara et al. 2014). However, the high level of automation featured by the underlying verification techniques (predicate abstraction and CEGAR (Kobayashi et al. 2011)) limit the modularity of the approach when compared to contract-based verification.

Perhaps the most closely related to our own approach are those based on dependent-types (Vazou 2016; Xi 2001). The approach shown in (Xi 2001) mentions *creation site* checking and is modular at the type level. However, complete manual specification of the termination argument is required. Furthermore, we show that well-foundedness of datatypes is not required for termination and constitutes too strong a constraint. Relaxing this constraint enables us to run our termination checker on a number of interesting and important benchmarks stating properties of lazy datastructures and infinite streams.

An important feature of our procedure consists in providing termination checking within the scope of a verification framework. It is clear that user annotations enable more powerful termination checking, however one must also realize that many interesting termination problems cannot even be specified without the use of contracts. This limitation of approaches without good contract support such as (Giesl et al. 2011, 2004; Kuwahara et al. 2014) constitute an important differentiating factor. There is therefore an important class of our benchmarks which cannot be compared with these techniques as they rely on subtle properties such as stream finiteness (`Deque`, `BottomUpMergeSort`, `LazyNumericalRep`) or bounded depth (`LazyTree` shown in Fig. 1) to establish termination. In general, the availability of contracts (even those developped for safety verification) consitutes a significant enhancement to our termination checking procedure.

A closely related line of work are the approaches for proving resource usage of programs such as time or space complexity(Avanzini et al. 2015; Danielsson 2008; Gulwani et al. 2009; Hoffmann et al. 2012; Jost et al. 2010; Madhavan et al. 2017; Madhavan and Kuncak 2014; Simões et al. 2012; Sinn et al. 2014; Vasconcelos et al. 2015; Zuleger et al. 2011). Many of these approaches subsume termination but are aimed at first-order programs, or work under a closed-world assumption that the targets of all indirect calls are available at the time of the analysis, especially those that are fully automated. However, Madhavan et al. (2017) present a technique that can use user-provided contracts to verify resource usage of higher-order programs. The approach relies on an external termination checker for soundness and does not present a termination algorithm. Furthermore, while the approach can handle limited class of open programs that do not accept arbitrary first-class functions as parameters, our approach is aimed at completely open, higher-order library that can accept arbitrary closures passed by the clients. We included some of the challenging benchmarks used in that work and prove them to terminating modulo callbacks for arbitrary clients.

7 CONCLUSIONS

We have presented a new approach for checking termination of higher-order programs. The approach is based on a new semantic notion of termination modulo callbacks. We show that this notion can be used to establish termination. Furthermore, we present sufficient conditions for establishing termination modulo callbacks in the form of weak and strong termination obligations.

We present an effective algorithm for checking the strong termination obligation using a safety verifier. In our approach, the safety verifier both checks the termination-related obligations and benefits from the induction schema obtained from the termination proofs.

Our termination checking approach is modular at two levels: at the level of programs, and at the level of functions, as explained below. Firstly, the termination conditions for an open program P are independent from its clients (i.e, calling contexts). This is because the conditions are defined on a dependency graph that is agnostic to the calling context, unlike a traditional call-graph. Secondly, the conditions generated for each edge in the dependency graph can be checked independently of the conditions generated for the other edges in the dependency graph, which implies that two functions in the program can be verified independently (like in a modular inter-procedural analysis). Another benefit of this approach is that, in practice, lambda application sites far outnumber lambda creation sites. Hence, a termination argument (like ours) that considers creation sites instead of application sites is more scalable.

Our results show that our termination checking technique is a practical method for verifying termination of higher-order programs. We expect our result to be important for building verifiers for functional languages, as well as for building more expressive theorem proving frameworks and dependently typed languages.

A PROOF OF THEOREM 1 - LOCALITY OF TERMINATION

In this section, we detail the proof of Theorem 1. Our proof requires introducing a few additional instrumentations to the operations semantics and a few novel notions such as *creator tree*. Eventually, we prove the theorem by extending (in a minor way) Tait's proof strategy for strong normalization of System F (Girard 1990). The following discussion makes clear the intuitive notion of creators and provides a more intuitive understanding of Theorem 1.

Instrumentation for Creation Labels. We define a new class of expressions called *instrumented expressions* that associates every lambda term appearing in the expressions with a sequence of dynamically created labels called *creator labels* (*CrLabels*). We denote a lambda term v with its creator labels by v^{δ}. We define an operational semantics for these instrumented expressions in Fig. 12. These creator labels on the instrumented expressions serve two purposes. (a) They help distinguish between different instances of closures created during an evaluation, i.e, they act similar to object references (or addresses) that uniquely identify the closures created at runtime. (This is unlike the static labels on expressions whose purpose is to relate a closure to a lambda term in the input program.) (b) They also help identify the chain of creators that resulted in the creation of a closure. Intuitively, a creator of a lambda term (or closure) is the closure whose application resulted in the lambda term (or closure) appearing for the first time. The meaning of this will become clearer after the discussion of creators and creator tree (see Definition A.1).

The semantic rules shown in Fig 12 are quite straightforward. The component Δ is meant to track the set of dynamic labels that are created at each step in the evaluation. This set is necessary as every dynamic label created at any step in the evaluation is chosen to be different from the labels created previously. As shown in Fig. 12, the operation $\mathit{freshDLs}(e, \delta, \Delta)$ annotates every lambda term in e with a creator sequence consisting of δ appended with a fresh label $\ell \notin \Delta$. For conciseness, we omit the Δ component while depicting the transitions in the instrumented semantics when it is not relevant for the context.

Any expression of the source language e without the instrumentation can be evaluated using the above semantic rules by applying the function $\mathit{freshDLs}(e, \emptyset, \emptyset)$. When we say an expression e of the source language is evaluated under the instrumented semantics we imply that $\mathit{freshDLs}(e, \emptyset, \emptyset)$ is evaluation under the semantic rules shown in Fig. 12. It is quite clear from Fig 12 that the evaluation of an expression under the instrumented semantics bisimulates that under the original semantics (section 2). Intuitively, the instrumented semantics can be viewed as just decorating the intermediate expressions arising during the

$$\textsc{Context}\quad \frac{(e,\Delta) \rightsquigarrow (e',\Delta')}{(\mathcal{C}[e],\Delta) \rightsquigarrow (\mathcal{C}[e'],\Delta')} \qquad \textsc{UnmarkedCall}\quad \frac{v_1 = (\lambda x.f\ \bar{w})^{\delta} \qquad (b,\Delta') = \mathit{freshDLs}(\mathit{body}(f),\delta,\Delta)}{(v_1\ v_2,\Delta) \rightsquigarrow (b[\bar{w}/\mathit{param}(f)][v_2/x],\Delta')}$$

$$\textsc{MarkedCall}\quad \frac{v_1 = (\lambda x.f\ \bar{w})^{\delta}+ \qquad mb = \mathit{markAll}(\mathit{body}(f)) \qquad (ib,\Delta') = \mathit{freshDLs}(mb,\delta,\Delta)}{(v_1\ v_2,\Delta) \rightsquigarrow (ib[\bar{w}/\mathit{param}(f)][v_2/x],\Delta')}$$

$$\mathit{freshDLs}(e,\delta,\Delta) = \begin{cases} ((\lambda x.f\ \bar{y})^{\delta;\ell}, \Delta \cup \{\ell\}) \text{ where } \ell \notin \Delta & \text{if } e = \lambda x.f\ \bar{y} \\ ((e_1'\ \mathit{freshDLs}(e_2,\delta,\Delta')),\Delta') \text{ where } (e_1',\Delta') = \mathit{freshDLs}(e_1,\delta,\Delta) & \text{if } e = e_1\ e_2 \\ (e,\Delta) & \text{Otherwise} \end{cases}$$

Fig. 12. Instrumented operational semantics that tracks creation labels of lambda terms. In the figure, the operation $\delta;\ell$ denotes a sequence δ appended with ℓ.

evaluation of an expression e with additional information. In the sequel, we use the instrumented semantics to introduce some important notions necessary for the proof.

Creator Relation. We now define a creator relation between applications that arise during an evaluation of a closed expression under the instrumented semantics. We say an expression e' is a creator of an application $a : (v^{\delta}\ u)$ if a lambda term with label δ (which eventually becomes the closure v) is obtained by the reduction of e' during the evaluation of e. Formally,

Definition A.1 (Creator). Let e be a closed expression. We say that an expression e' is a creator of an application $a : (v^{\delta}\ u)$, denoted $e' \underset{e}{\gg} a$ iff for some evaluation contexts $\mathcal{C}$ and $\mathcal{C}'$, $(e,\emptyset) \rightsquigarrow^* (\mathcal{C}[e'],\Delta') \rightsquigarrow (\mathcal{C}[e''],\Delta'') \rightsquigarrow^* (\mathcal{C}'[v^{\delta}\ u],\_)$, and $\delta \notin \Delta'$ and $\delta \in \Delta''$.

Recall from instrumented semantic rules that creation of new dynamic labels happen only during applications (i.e, beta-reductions). Therefore, if e' is a creator of an application a then either e' is the initial closed expression e or e' is another application. Furthermore, by the definition of instrumented semantics, every application has a unique creator because every dynamic label is created exactly once during some beta-reduction step (see Fig. 12). We ignore the subscript of $\underset{e}{\gg}$ if it is clear from the context. Let $\gg^*$ denote the reflexive, transitive closure of $\gg$.

Importantly, the creator relation has a strong correlation with the *LocalCall* relation namely that if c is a transitive creator of c' during some evaluation of an expression e, and if c does not complete before c' then c' is a local call of c (under a well-defined closing environment). This is formally stated by the following lemma.

Lemma 10 (Creator Locality)**.** *Let e be a closed expression well-typed with respect to a library L. Let c and c' be two applications such that $c \gg^* c'$ and $e \rightsquigarrow \mathcal{C}[c] \rightsquigarrow \mathcal{C}[\mathcal{C}'[c']]$. That is, c is a transitive creator of c' and c' is invoked before c completes. Let c be of the form $(\lambda x.\ f\ \bar{w})\ u$. $c' \in \mathit{LocalCall}(L, L, \mathit{body}(f), \sigma_{param}((\lambda x.\ f\ \bar{w})\ u))$.*

Proof. It is given that $c \gg^* c'$. Therefore, there must exist a $k \in \mathbb{N}$ such that $c \gg^k c'$. The proof is straightforward to establish by induction on k. Say $c \gg^{k-1} b \gg c'$. By hypothesis, $b \in$

$\mathit{LocalCall}(L, L, \mathit{body}(f), \sigma_{param}(c))$. Let the closure invoked by b be $(\lambda x.g\ \bar{w})^{\delta}$. Since b is the creator of c', by the definition of creator, b introduces a lambda term with δ. As per the semantics this means that $\mathit{body}(g)$ has a lambda term which is labeled δ, where g is the function invoked by b. The lambda term will be unmarked iff the closure invoked by b is unmarked. By hypothesis, b invokes an unmarked closure in the evaluation $\mathit{body}(f)\ \sigma_{param}(c)$. Thus, c' will also invoke an unmarked closure in the evaluation $\mathit{body}(f)\ \sigma_{param}(c)$. Thus, $c' \in \mathit{LocalCall}(L, L, \mathit{body}(f), \sigma_{param}(v\ u))$. ◻

Creator Tree. We now define a graph called creator tree for a closed expression e well-typed with respect to a library L. This structure simplifies the presentation of the proof and also provides useful insights into the evaluation of an expression. The nodes of the graph consist of the expression e and the applications arising during the evaluation of e. There is a directed edge from a node p to a node a iff $p \gg a$.

This graph is a tree rooted at e since (a) every application except the root (which are nodes) have a creator and hence a parent in the tree, (b) every application is only connected to its parent and hence has unique path to the root. The following are some of the properties of the creator tree.

Property 11. *Let e be a closed expression well-typed with respect to a library L. A creator tree T of e satisfies the following properties.*

I. Every path in the tree corresponds to a chain of creators.
II. For a node $(v^{\delta}\ u)$ in the tree (other than the root) the length of the sequence δ is equal to the depth of the node.
III. The tree is infinite iff the evaluation of e is non-terminating.
IV. The tree may be infinitely branching if e is non-terminating.

We omit the proofs for the above properties as they are straightforward to derive from the definition of the creator tree. We now present one of the interesting, non-trivial property of the creator tree.

Theorem 12. *Let T be a creator tree for a closed expression e that is well-typed with respect to a library L. e is non-terminating iff the tree T has unbounded depth, i.e, there does not exist a $d \in \mathbb{N}$ such that for every node, the length of the path from root to the node is at most d.*

Proof. The if-direction is trivial: if the creator tree has unbounded depth then it should have infinite size which implies that evaluation of e results in infinite application and hence non-terminating. Therefore, consider the only-if direction. Let e be an closed expression that is non-terminating. We prove this direction by contradiction. Assume that there exists a $d \in \mathbb{N}$ such that depth of the creator tree is at most d. We now derive a contradiction that e should be terminating.

Error Construct. For the purpose of this proof, we introduce a construct $\mathsf{error}[\tau]$ that has the type τ to our toy language. This construct cannot reduced any further and hence halts an evaluation if during the evaluation it appears in a position that needs to be reduced. That is, $\forall \mathcal{C} \in \mathit{EContext}. \neg\exists e. \mathcal{C}[\mathsf{error}[\tau]] \rightsquigarrow e$. Clearly, adding the $\mathsf{error}[\tau]$ construct does not result in new sources of non-termination. In particular, if a language is *strongly or weakly normalizing* then adding the $\mathsf{error}[\tau]$ construct to the language preserves this property. The $\mathsf{error}[\tau]$ construct is used in our proof to reduce expressions belonging to the toy language to those that belong to a strongly normalizing fragment of lambda calculus.

Construction of a Acyclic Library. We construct a library L' from L by unfolding all function in L upto depth d as explained below. For every function $f \in L$, we introduce $d+1$ functions $\{f^k \mid 1 \leq k \leq d+1\}$ in L' constructed as follows. The body of each function f^k, $k \leq d$, is constructed by replacing in the $body(f)$ every direct call of the form $f\ \bar{y}$ by $f^{k+1}\ \bar{y}$. That is, every function f^k, $k \leq d$ only refers to functions f^{k+1} in the library L'. If **def** $f(\bar{x}) : \tau_1 \Rightarrow \tau_2 := e$ is the definition of f in L, the function f^{d+1} is defined in L' as **def** $f^{d+1}(\bar{x}) : \tau_1 \Rightarrow \tau_2 := \mathsf{error}[\tau_2]$. That is, the body of f^{d+1} is the error construct. Hence if the f^{d+1} is ever invoked during an evaluation the evaluation halts. Let e' be an expression obtained by replacing in e every reference to a function f by f^1.

We now claim that the evaluation of e (with respect to L) *bisimulates* the evaluation of e' (with respect to L') under the instrumented semantics. We prove this by defining a bisimulation relation between the expressions that arise during the valuation as follows. Let $\sim \subseteq Expr \times Expr$ be defined as follows.

$$\begin{array}{l}
\forall x \in Vars, \{\bar{y}, \bar{y}'\} \subseteq (Vars \cup Value)^*, f \in Fids, \{\delta, \delta'\} \subseteq CrLabels. \\
\qquad (\lambda x.f\ \bar{y})^{\delta} \sim (\lambda x.f^{|\delta|}\ \bar{y}')^{\delta'}\ \textit{iff}\ \bar{y} \sim \bar{y}' \\
\forall \{e_1, e_2, e_1', e_2'\} \subseteq Expr.\ e_1\ e_2 \sim e_1'\ e_2'\ \textit{iff}\ e_1 \sim e_1' \wedge e_2 \sim e_2' \\
\forall x \in Vars.\ x \sim x \\
\forall n \in \mathbb{N}. \{\bar{y}, \bar{y}'\} \subseteq (Vars \cup Value)^n.\ \bar{y} \sim \bar{y}'\ \textit{iff}\ \forall i \in [1,n].y_i \sim y_i' \\
\forall \tau \in Type.\ \mathsf{error}[\tau] \sim \mathsf{error}[\tau]
\end{array}$$

In the above definition, $|\delta|$ denotes the length of the sequence δ of dynamic labels. To prove that $\sim$ is a bisimulation we need to show that (a) initially $freshDLs(e, \emptyset, \emptyset) \sim freshDLs(e', \emptyset, \emptyset)$ holds and (b) whenever $e_1 \sim e_2$ and $e_1 \rightsquigarrow e_1'$ then there exists a e_2' such that $e_2 \rightsquigarrow e_2'$ and $e_1' \sim e_2'$ (and vice-versa). The part (a) holds since every lambda term in $freshDLs(e, \emptyset, \emptyset)$ is of the form: $(\lambda x.f\ \bar{y})^{\ell}$, where ℓ is a dynamic label, and e' should have a lambda term $\lambda x.f^1\ \bar{y}$ by construction.

Now consider the part (b). Say $e_1 \rightsquigarrow e_1'$ and $e_1 \sim e_2$. Say the reduction invokes one of the application rules (otherwise the claim holds by induction on the expression syntax). That is say $e_1 = \mathcal{C}[(\lambda x.f\ \bar{y})^{\delta}\ u]$. Since $e_1 \sim e_2$, e_2 should be of the form $\mathcal{C}[(\lambda x.f^{|\delta|}\ \bar{y})\ u']$ and $u \sim u'$. First, there has to exist a e_2' since a call reduction rule shown in Fig. 12 applies. Secondly, every newly introduced lambda term in e_1' would have a creator sequence of length $|\delta| + 1$. Now if $|\delta| < d$ then by the construction of L' every lambda term in $body(f^{|\delta|})$ would refer to the function $f^{|\delta|+1}$, implying that $e_1' \sim e_2'$. If $|\delta| \geq d$ then it means that during the evaluation of e, there exists an application c with length of the dynamic label greater than d. By Property 11(II), the depth of the node c in the creator tree of e is greater than d. This is not possible since we assumed that tree has depth at most d. Therefore, $|\delta| \geq d$ is not possible. Hence, the evaluation of e is simulated by the evaluation of e'. The converse can be proven similarly.

Termination under Acyclic Library. Now, consider the expression e' and the library L'. By construction, no function in L' uses itself (mutually) recursively. Let e_{norm} be obtained from e by transitively *inlining* every call to a named function with their body. e_{norm} refers to no named function and belongs to *simply typed lambda calculus* (STLC) fragment augmented with an $\mathsf{error}[\tau]$ construct. Since this is a strongly normalizing fragment, e_{norm} reduce to a normal form under any reduction strategy (including call-by-value). Since e_{norm} is a closed expression, the normal form it reduces to should be a value. In other words, e_{norm}

is terminating. Therefore, e' should also be terminating. Thus, e should be terminating since evaluations of e' and e are bisimulations, which is a contradiction. □

Theorem 13. *Let T be a creator tree for a closed expression e that is well-typed with respect to a library L. e is non-terminating iff there does not exist a $d \in \mathbb{N}$ such that for every node, the number of incomplete calls in the path from root to the node is at most d.*

Proof. The if-direction directly follows form Theorem 12. Therefore consider the only-if direction. Let e be a closed expression that is non-terminating. We prove this direction by contradiction. Assume that there exists a $d \in \mathbb{N}$ that bounds the number of incomplete calls in every path from root. In other words, if $e \rightsquigarrow^* \mathcal{C}[v^\delta\ u]$ and $|\delta| > d$ then $(v^\delta\ u)$ is a complete call. We now derive a contradiction that the expression e must terminate.

To establish this we proceed similar to the proof of Theorem 12 by unfolding the direct function calls upto a certain depth. But, here we cannot completely eliminate recursion in the unfolded program. We show termination of the unfolded program by encoding it into a simply typed lambda calculus extended with terminating recursive functions that can be shown to strongly normalizing.

Construction of a Library with Terminating Functions. Similar to the proof of Theorem 12, we construct a library L' by unfolding the functions in L upto the depth d. But instead of replacing the body of functions at depth $d+1$ by error[] as in the proof of Theorem 12, we make them identical to the original function definition in L.

More formally, for every function $f \in L$, we add f to L and also introduce d functions $\{f^k \mid 1 \le k \le d\}$ in L' constructed as follows. The body of each function f^k, $k < d$, is constructed by replacing in the $\mathit{body}(f)$ every direct call of the form $f\ \bar{y}$ by $f^{k+1}\ \bar{y}$. The body of f^d is the same as the body of f in L. (Note that the function f also belongs to L'.) Let e' be an expression obtained by replacing in e every reference to a function f by f^1. It is easy to see that the following relation introduces a bisimulation between the evaluations of e and e'. (The proof is similar to the one shown in Theorem 12 and is hence omitted.)

$$\begin{aligned}
&\forall x \in \mathit{Vars}, \{\bar{y}, \bar{y}'\} \subseteq (\mathit{Vars} \cup \mathit{Value})^*, f \in \mathit{Fids}, \{\delta, \delta'\} \subseteq \mathit{CrLabels}.\\
&\qquad (\lambda x.f\ \bar{y})^\delta \sim (\lambda x.f^{|\delta|}\ \bar{y}')^{\delta'}\ \mathit{iff}\ |\delta| \le d \wedge \bar{y} \sim \bar{y}'\\
&\qquad (\lambda x.f\ \bar{y})^\delta \sim (\lambda x.f\ \bar{y}')^{\delta'}\ \mathit{iff}\ |\delta| > d \wedge \bar{y} \sim \bar{y}'\\
&\forall \{e_1, e_2, e_1', e_2'\} \subseteq \mathit{Expr}.\ e_1\ e_2 \sim e_1'\ e_2'\ \mathit{iff}\ e_1 \sim e_1' \wedge e_2 \sim e_2'\\
&\forall x \in \mathit{Vars}.\ x \sim x\\
&\forall n \in \mathbb{N}.\{\bar{y}, \bar{y}'\} \subseteq (\mathit{Vars} \cup \mathit{Value})^n.\ \bar{y} \sim \bar{y}'\ \mathit{iff}\ \forall i \in [1,n].y_i \sim y_i'
\end{aligned}$$

We now claim that every application of the form $(\lambda x.f\ \bar{w}')\ u'$ in the evaluation of e' is a complete call (i.e, it terminates). This is because by the above bisimulation relation there exists a corresponding call $(\lambda x.f\ \bar{w})^\delta\ u$ such that $|\delta| > d$ in the evaluation of e. Clearly, the evaluation of both calls bisimulate each other. We are given that any application in the evaluation of e having depth greater than d in the creator tree of e is complete. Thus, $(\lambda x.f\ \bar{w})^\delta\ u$ and $(\lambda x.f\ \bar{w}')\ u'$ are complete calls. Let e_{norm} be an expression obtained by (transitively) inlining in e' every call to a named function f^k $(1 \le k \le d)$ with its body. Calls of the form $f\ \bar{y}$ (which invoke a function without a superscript) are not inlined as they could be mutually

$$\begin{array}{rcl} \tau \in \mathit{Type} & ::= & \bar{\tau} \mid \tau \Rightarrow \tau \mid \mathsf{Unit} \\ e_\lambda \in \Lambda_{fun} & ::= & \lambda x.f\ \bar{y} \qquad \text{where } \bar{y} \in \Lambda_{fun} \cup \mathit{Vars} \\ e_s \in E_{src} & ::= & x \mid e_\lambda \mid e_s\ e_s \\ e \in \mathit{Expr} & ::= & e_s \mid \lambda x.e \mid e\ e \\ \mathit{Fdef} & ::= & \mathbf{def}\ f(\bar{x}) : \tau_1 \Rightarrow \tau_2 := e_s \end{array}$$

(a)

$$\{(f,\tau) \mid \mathbf{def}\ f(\bar{x}) : \tau := e \in L\} \subseteq \Gamma \subseteq (\mathit{Vars} \cup \mathit{Fids}) \times \mathit{Type}$$

$$\textsc{Var}\ \frac{u \in \mathit{Vars} \cup \mathit{Fids} \qquad (u,\tau) \in \Gamma}{\Gamma \vdash u : \tau} \qquad \textsc{Lambda}\ \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \Rightarrow \tau_2} \qquad \textsc{DirectCall}\ \frac{\Gamma \vdash f : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash f\ e : \tau_2}$$

$$\textsc{Seq}\ \frac{|\bar{e}| = |\bar{\tau}| \qquad \forall i.\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \bar{e} : \bar{\tau}} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_3}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \textsc{FunDef}\ \frac{|\bar{x}| = |\bar{\tau_1}| \qquad \Gamma[\bar{x} \mapsto \bar{\tau_1}] \vdash e : \tau}{\Gamma \vdash \mathbf{def}\ f(\bar{x}) : \bar{\tau_1} \Rightarrow \tau := e : \mathsf{Unit}}$$

(b)

Fig. 13. Syntax and typing rules of a simply-typed lambda calculus with named functions.

$$\textit{(Values)}\ v \in \mathit{Value} = \lambda x.e$$
$$\textit{(Evaluation contexts)}\ \mathcal{C} \in \mathit{EContext} = [\ ] \mid \mathcal{C}\ e \mid v\ \mathcal{C}$$

$$\frac{e \rightsquigarrow_r e'}{\mathcal{C}[e] \rightsquigarrow_r \mathcal{C}[e']} \quad r \in \{\beta, \mu, \mu'\} \qquad \frac{\lambda x.e \notin \Lambda_{fun} \qquad v \in \mathit{Value}}{(\lambda x.e)\ v \rightsquigarrow_\beta e[v/x]} \qquad \frac{\bar{w} \in (\mathit{Value} \cup \{x\})^* \qquad v \in \mathit{Value}}{(\lambda x.f\ \bar{w})\ v \rightsquigarrow_\mu \mathit{body}(f)[\bar{w}/\mathit{param}(f)][v/x]}$$

$$\rightsquigarrow_1\ =\ \rightsquigarrow_\beta \cup \rightsquigarrow_\mu$$

(a) System 1

$$\frac{\bar{w} \in (\mathit{Value} \cup \{x\})^* \qquad v \in \mathit{Value} \qquad (\lambda x.f\ \bar{w})\ v \in \mathit{SN}_1}{(\lambda x.f\ \bar{w})\ v \rightsquigarrow_{\mu'} \mathit{body}(f)[\bar{w}/\mathit{param}(f)][v/x]} \qquad \mathit{SN}_1 = \{e \mid \exists n \in \mathbb{N}. \neg \exists e'. e \rightsquigarrow_1^n e'\}$$

$$\rightsquigarrow_2\ =\ \rightsquigarrow_\beta \cup \rightsquigarrow_{\mu'}$$

(b) System 2

Fig. 14. (a) System 1: Call-by-value reduction rules for direct calls and applications. (c) System 2: A strongly normalizing reduction system.

recursive. We now show that e_{norm} should be terminating by defining a strongly normalizing reduction system in which the evaluation of e_{norm} could be simulated.

Termination of Expressions with Complete Calls. Consider the Simply typed lambda calculus extended with named recursive functions presented in Fig. 13 and the reduction rules shown in Fig. 14(a) referred to as System 1. The expression E_{src} corresponds to the expressions of the toy language shown in Fig. 3. The bodies of functions belong to E_{src}. However, the language also allows lambda terms whose bodies could be arbitrary expressions. Such expressions are denoted using *Expr*. The typing rules are identical to those of the toy language shown in Fig. 4.

The reduction rules of System 1 shown in Fig. 14(a) consist of a usual beta reduction rule, and a μ-reduction rules for lambda terms in Λ_{fun} that applies the body of a named function. Note that compared to Fig. 5, we ignore the marks on the lambda terms. Moreover, the β and μ reductions can happen not just on closed lambda terms (closures) but also on lambda terms with free variables (which are the values of this system). This calculus can be thought of as a generalized form of the operation semantics shown in Fig. 5 but without the additional instrumentation. It is quite obvious that System 1 bisimulates the semantics of Fig.5 and also the instrumented semantics for the expressions of the toy language. (However System 1 is applicable to expressions with a slightly more relaxed syntax as meanioned earlier.) It is evident that System 1, which is closely related to System F^{rec} of (Barthe et al. 2008), is not normalizing since it allows unrestricted recursion. We now devise a more restricted reduction system: System 2 shown in Fig. 14(b) using System 1 that is strongly normalizing while being capable of bisimulating e_{norm}.

Note that the expression e_{norm} belongs to the syntax presented in Fig. 13. Its evaluation under System 1 bisimulates the evaluation of e' under the instrumented semantics (as we only inline certain functions in e' with their body to obtain e_{norm}). Moreover, every application $(\lambda x.f\ \bar{w})\ u$ that arises during the evaluation of e_{norm} is terminating under the System 1. This is because we know that every application of the form $(\lambda x.f\ \bar{w})\ u$ is complete in the evaluation of e'. Note that System 1 has only one deterministic reduction sequence for each expression. Hence call-by-value termination and strong normalization coincide for System 1. Therefore, every application of a lambda term in Λ_{fun} arising during the evaluation of e_{norm} is strongly normalizing. Based on this observation we define a restricted (theoretical) System 2 shown in Fig. 14(b).

System 2 consists of the beta reduction rule and reduction rule μ' for applications of terms in Λ_{fun}. The rule fires only when the arguments passed to the direct call are known to be strongly normalizing under System 1 (given by the set SN_1). Though the undecidability of strong normalization of System 1 implies that it is not practical to implement such a system, System 2 serves as a theoretical basis for proving that the evaluation of e_{norm} is terminating (or normalizing).

Note that the rule μ' is a restriction of the rule μ and hence every term that is normalizing in System 1 will also do so in System 2. The following are some of the properties of System 2.

Property 14. *Every application of the form $(\lambda x.f\ \bar{w})\ e$, where e is strongly normalizing, is strongly normalizing in System 2.*

The proof of the above property is quite straightforward. Say e reduces to a normal form u. If $\bar{w} \in (\mathit{Value} \cup \{x\})^*$ and $u \in \mathit{Value}$ then the reduction μ' applies, but only if the application is normalizing in System 1 and hence in System 2. Otherwise, no reduction applies in System 2, since u is already normal and the evaluation context does not allow to reduce $\bar{w}$.

We now show that System 2 is strongly normalizing. We follow the proof strategy explained in the Chapter 6 of the book Girard et al. (1989) for establishing strong normalization of simply typed lambda calculus . The approach is based on the strong normalization proof of System F originally proposed by Tait (1967) and later refined by Girard (1972). Although System 2 does not support type polymorphism, this proof strategy also generalizes to type polymorphism as detailed in the book Girard et al. (1989) Chapter 11, and hence allows us to extend the theorems to a language with type polymorphism.

We first define a set of *reducible* terms, denoted RED_T, for each type T. Following Girard et al. (1989) we define RED_T as follows:

- A expression t of unit type is reducible (i.e, $t \in \mathit{RED}_{Unit}$) if it is strongly normalizing.
- Every lambda term of the form $\lambda x.f\ \bar{w}$ of type $U \Rightarrow V$ is reducible iff for all $i \in [1, |\bar{w}|]$, w_i is reducible.
- Every other expression t having a function type $U \Rightarrow V$, $t \in \mathit{RED}_{U \Rightarrow V}$ iff $\forall u.(u \in \mathit{RED}_U \Rightarrow t\ u \in \mathit{RED}_V)$.

The only subtlety compared to the definition of RED_T presented in section 6.1 of Girard et al. (1989) is that here we also define reducibility of a lambda term that invokes recursive functions.

We say a term is *neutral* if it is of the form: x or $e_1\ e_2$. We now have to show that the reducible definition satisfies the following four properties (detailed in section 6.3 of Girard et al. (1989)):

CR 1 If $t \in \mathit{RED}_T$, t is strongly normalizing.
CR 2 If $t \in \mathit{RED}_T$ and $t \rightsquigarrow_2 t'$, $t' \in \mathit{RED}_T$.
CR 3 If t is neutral and normal, then $t \in \mathit{RED}_T$.
CR 4 If t is neutral and $t \rightsquigarrow_2 t'$ and $t' \in \mathit{RED}_T$, then $t \in \mathit{RED}_T$.

(The original formulation in Girard et al. (1989) combines the properties 3 and 4 into a single property. We preserve the distinction for clarity.) We now consider each property in turn and show that they hold by using induction on the structure of the types.

Consider the property **CR 1**. It is a tautology for unit types. For lambda terms and variables of function type the property holds since they are normal forms. Consider an term $t \in \mathit{RED}_{U \Rightarrow V}$ Here we use the same argument used in section 6.2.3 of Girard et al. (1989). By definition of reducibility, $t\ x$ is reducible for any variable x of type U (since x is neutral and normal and hence reducible by **CR 3** hypothesis on U). $t\ x$ is of type V and by hypothesis it is strongly normalizing. Hence, t should also be strongly normalizing (since in call-by-value reduction order t should be first reduced to a normal form while evaluating $t\ x$).

Consider the property **CR 2**. Say $t \in \mathit{RED}_T$ and $t \rightsquigarrow t'$. If T is a unit type, t and hence t' should be strongly normalizing. Hence $t' \in \mathit{RED}_{Unit}$. As before, if t is a lambda term or a variable then it cannot reduce and hence the claim holds trivially. Now say t is an application having a function type $U \Rightarrow V$. Take u reducible of type U. Then $t\ u$ is reducible (since t is reducible), and $t\ u \rightsquigarrow t'\ u$ (in call-by-value evaluation order). The induction hypothesis for V gives us that $t'\ u$ is reducible. Hence, $t'\ u$ is reducible for all reducible u. Hence $t' \in \mathit{RED}_{U \Rightarrow V}$.

Consider the property **CR 3**. It is a tautology for terms of unit type. Therefore, consider a neutral term t of function type $U \Rightarrow V$. We need to show that $t\ u \in \mathit{RED}_V$ for any term $u \in \mathit{RED}_U$. Now if $t\ u$ is normal then by induction hypothesis on V (**CR 3**), $t\ u \in \mathit{RED}_V$. Therefore, say $t\ u$ is not normal. Therefore $(t\ u) \rightsquigarrow t'$. t cannot be a lambda term since t is neutral. Since t is normal, the reduction in $t\ u$ must happen within u, i.e, $t\ u \rightsquigarrow t\ u'$. But we know u is reducible and hence strongly normalizing. Thus, by inducting on the number of steps taken to reduce u to a normal form, we can establish that $(t\ u) \in \mathit{RED}_V$ (see section 6.2.3 of Girard et al. (1989) for intuition on this step).

Finally, consider the property **CR 4**. The proof of this is very similar to **CR 3**. It is trivial for expression of unit type, since if t' is reducible it is normalizing and hence t should also be normalizing and reducible. Furthermore, t cannot be variables or lambda terms as they do not reduce to a t'. Consider an application t of function type $U \Rightarrow V$. Say $t \rightsquigarrow t'$ and t' is reducible. We need to show that $t\ u \in \mathit{RED}_V$ for any expression $u \in \mathit{RED}_U$. Since $t \rightsquigarrow t'$, by call-by-value order $t\ u \rightsquigarrow t'\ u$. Since t' is reducible i.e, $t' \in \mathit{RED}_{U \Rightarrow V}$, by reducibility definition $t'\ u$ is also reducible i.e, $t'\ u \in RED_V$. Using hypothesis for V (**CR 4** property), $t\ u$ is reducible. Hence, all four properties hold for our definition of reducible terms.

Using the above properties we now establish a property about terms in Λ_{fun} which is unique to our system.

Property 15. *For every lambda term $\lambda x.f\ \bar{w} \in \Lambda_{fun}$ of type $U \Rightarrow V$ that is reducible, for all $u \in RED_U$, $(\lambda x.f\ \bar{w})\ u \in RED_V$. In other words, any $\lambda x.f\ \bar{w}$ that is reducible will satisfy the properties of reducible lambda terms not in Λ_{fun}.*

By property 14, any $(\lambda x.f\ \bar{w})\ u$ (for a reducible u) is strongly normalizing. Therefore, we prove the property by induction on the number of steps in the reduction of $(\lambda x.f\ \bar{w})\ u$ to a normal.

Base Case: If $(\lambda x.f\ \bar{w})\ u$ is normal then it is reducible. This directly follows by property **CR 3**.

Inductive Step: Assume that the claim holds upto k steps. We now show it for $k+1$ steps.

Say $(\lambda x.f\ \bar{w})\ u \rightsquigarrow \mathit{body}(f)[\bar{w}/\mathit{param}(f)][u/x]$. We now induct on the structure of the term $\mathit{body}(f)$ and prove that $\mathit{body}(f)[\bar{w}/\mathit{param}(f)][u/x]$ is reducible for any structure. Claim: for any t_{sub} that is sub-expression of $\mathit{body}(f)$, $t_{sub}[\bar{w}/\mathit{param}(f)][u/x]$ is reducible.

Let $t' = t_{sub}[\bar{w}/\mathit{param}(f)][u/x]$.

(a) If t_{sub} is a variable, then t' belongs to $\bar{w}$ or u (because there could be no other free variable in $\mathit{body}(f)$). Each w_i is reducible since $\lambda x.f\ \bar{w}$ is reducible and u is reducible by assumption. Therefore t' is reducible.

(b) If t_{sub} is a lambda term then it should be of the form $\lambda x.g\ \bar{z} \in \Lambda_{fun}$ by the syntax definition of $\mathit{body}(f)$. In this case $t' = \lambda x.g\ (\bar{z}[\bar{w}/\mathit{param}(f)][u/x])$. By hypothesis, $\bar{z}[\bar{w}/\mathit{param}(f)][u/x]$ is reducible. Thus, t' is also reducible.

(c) If t_{sub} is an application of the form $e_1\ e_2$ then $t' = t'_1\ t'_2$, where $t'_i = e_i[\bar{w}/\mathit{param}(f)][u/x]$, for $i \in \{1,2\}$. By hypothesis both t'_1 and t'_2 are reducible. Now, if $t'_1 \notin \Lambda_{fun}$, by definition of reducibility $t'_1\ t'_2$ is reducible. Now say t'_1 is of the form $\lambda x.g\ \bar{z} \in \Lambda_{fun}$. The application $(\lambda x.g\ \bar{z})\ t'_2$ should normalize by property 14. We now show that it does so in fewer steps than $(\lambda x.f\ \bar{w})\ u$.

Let $e \in E_{src}$ be an expression of the source language that is normalizing. Recall that the bodies of all lambda terms in e are lifted to named functions. Every application in the expression e must be reduced to a normal form during the evaluation of e. It is easy to see this by induction on the structure of the expression. If e is a variable or lambda term then there are no application in e (note that the bodies of all lambda terms are either variables or values i.e, closed lambda terms in E_{src}). If e is an application of the form $e_1\ e_2$ then both e_1 and e_2 have to be reduced to normal forms by call-by-value reductions. Thus by induction hypothesis, every application in e_1 and e_2 will be reduced. Finally, $e_1\ e_2$ will also be reduced to a normal form.

By this property, the application $e_1\ e_2$ which is a sub-expression of $\mathit{body}(f) \in E_{src}$ would be normalized during the evaluation of $\mathit{body}(f)$. Hence, $(e_1\ e_2)[\bar{w}/\mathit{param}(f)][u/x]$ would be normalized during the evaluation of t'. Thus, $(\lambda x.g\ \bar{z})\ t'_2$ will normalize in fewer or same number of steps as t', and hence in strictly fewer steps than $(\lambda x.f\ \bar{w})\ u$. By the hypothesis on the number of reduction steps, $(\lambda x.g\ \bar{z})\ t'_2$ is reducible. Hence the property.

Now that we have established that RED_T satisfies the above four properties, we can reproduce the lemmas necessary for proving strong normalization by Tait and Girard's strategy. Below we outline the proof for the lemmas for completeness and also to show that the lemmas are unaffected by the introduction of terms in Λ_{fun} due to the property 15.

Lemma 16. *Let t be any expression (not assumed to be reducible) well-typed with respect to a library L under the typing rules shown in Fig. 13, and let $\bar{x}$ with type $\bar{U}$ be the free variables of t. If $\bar{u}$ is a sequence of reducible terms of size $|\bar{x}|$ then $t[\bar{u}/\bar{x}]$ is reducible.*

The proof is very similar to that of theorem 6.3.3 presented in the book Girard et al. (1989). However, for completeness we present the proof below. We prove this lemma by inducting on the structure of t.

(1) If t is x_i then $t[\bar{u}/\bar{x}]$ is u_i which is reducible by assumption.
(2) If t is $w\ v$ by induction hypothesis $w[\bar{u}/\bar{x}]$ and $v[\bar{u}/\bar{x}]$ are reducible. Now if $w[\bar{u}/\bar{x}] \in \Lambda_{fun}$, by property 15, $w[\bar{u}/\bar{x}]\ (v[\bar{u}/\bar{x}])$, which is $t[\bar{u}/\bar{x}]$, is reducible. If $w[\bar{u}/\bar{x}] \notin \Lambda_{fun}$ by definition $w[\bar{u}/\bar{x}]\ (v[\bar{u}/\bar{x}])$ is reducible and so is $t[\bar{u}/\bar{x}]$.
(3) If t is $(\lambda x.e) \notin \Lambda_{fun}$ of type $V \Rightarrow W$, by induction hypothesis, for all v of type V, $e[\bar{u}/\bar{x}, v/y]$ is reducible. This implies that $t[\bar{u}/\bar{x}] = \lambda y.(w[\bar{u}/\bar{x}])$ is reducible. (Lemma 6.3.2 of Girard et al. (1989) has a formal proof of the last part).
(4) If t is $(\lambda x.f\ \bar{y}) \in \Lambda_{fun}$, by hypothesis, $\bar{y}[\bar{u}/\bar{x}]$ is reducible. By definition, $\lambda x.f\ \bar{y}[\bar{u}/\bar{x}] = t[\bar{u}/\bar{x}]$ is reducible.

As a corollary of the above lemma, we obtain that all expressions t are strongly normalizing in System 2 by using $\bar{u} = FV(t)$.

Termination of e. The evaluation of e_{norm} under System 2 bisimulates the evaluation of e_{norm} under System 1, which bisimulates the evaluation of e' under the instrumented semantics, which in turn bisimulates the evaluation of e under the instrumented semantics. Since e_{norm} is strongly normalizing in System 2, e should be terminating under the instrumented semantics and hence also as per the operational semantics of Fig. 5 This is a contradiction to our assumption that e is non-terminating. Thus, there does not exist a d that bounds the number of incomplete calls from the root of a creator tree of e to every node in the tree if e is non-terminating. □

Corollary 17. *Let T be a creator tree for a closed expression e that is well-typed with respect to a library L. e is non-terminating iff for some function f, there does not exist a $d \in \mathbb{N}$ such that for every node, the number of incomplete calls of the form $(\lambda x.f\ \bar{w})\ u$ (for some $\bar{w}$ and u) in the path from root to the node is at most d.*

Proof. The if-part follows directly from the main Theorem 13. Let e be non-terminating. Assume for argument sake that there exists a depth d_f for every function in the $f \in L$ such that for every node, the number of incomplete calls of the form $(\lambda x.f\ \bar{w})\ u$ (for some $\bar{w}$ and u) in the path from root to the node is at most d. Define $d = \max_{f \in L} d_f$. Consider a path in from root to an arbitrary node n in the creator tree whose length is greater than $d * |L|$. There has to exist such a path by Theorem 13. By the syntactic restriction of the toy language, every application in this path should be of the form $(\lambda x.f\ \bar{w})\ u$ for some function f, $\bar{w}$, u. Therefore at least only f should appear inside the invoked lambdas more than $d + 1$ times, which is a contradiction to our assumption. □

The locality theorem of termination presented in section 3 follows from the above corollary and Lemma 10 in a straightforward way as detailed below.

Theorem 1 (Locality of Termination)**.** *Let e be a closed expression that is well-typed under a library L. e is terminating if (and only if) for every function f, there exist a $n \in \mathbb{N}$ that upper bounds the length of every sequence of incomplete calls $S : (v_1\ u_1), (v_2\ u_2), \cdots$ where each v_i is of the form $\lambda x.f\ \bar{w}_i$ and $\forall i \geq 2.\ (v_i\ u_i) \in \mathit{LocalCall}(L, L, \mathit{body}(f), \sigma_{\mathit{param}}(v_{i-1}\ u_{i-1}))$.*

Proof. The only-if direction trivially follows since if e is a terminating expression then every call sequence in the evaluation of e should be finite. Consider the if direction. We now prove the contra-positive form of this direction. Say e is a non-terminating expression. By Corollary 17 we know that in the creator tree of e, for some function f there does not exist a $d \in \mathbb{N}$ such the number of incomplete calls applying a lambda calling f from root to any node of the tree is bounded by d. In other words, there does not exist an upper bound on the length of every sequence of incomplete calls of the form: $S : (v_1\ u_1) \gg^* (v_2\ u_2) \gg^* \cdots$ where each v_i is of the form $\lambda x.f\ \bar{w}_i$. The claim follows from this by Lemma 10. □

B PROOFS OF CORRECTNESS OF TERMINATION OBLIGATIONS

We now detail the proofs of correctness and completeness theorems of the termination obligations presented in the paper.

Lemma 2 (Local Closure Property)**.** *Let e be an open expression well-typed with respect to a library L, and let (σ, L') be a closing environment. Let $(\ell : v) \in \mathit{LocalClosure}(L, L', e, \sigma)$, where ℓ is the static label of v. There exists a lambda term $\lambda x.g\ \bar{y}$ that is a sub-expression of e such that $\exists((\lambda x.g\ \bar{w})\ u') \in \mathit{LocalCall}(L, L', e, \sigma)$ and $v = \lambda x.g\ \bar{w}$ or $v \in \mathit{LocalClosure}(L, L', \mathit{body}(g), \sigma_{\mathit{param}}((\lambda x.g\ \bar{w})\ u'))$.*

Proof. By the definition of the semantics, a local closure v of e w.r.t (σ, L') should either be a closure of a lambda term in e, in which case the claim trivially holds, or be created by an application of another local closure of e say r. By induction on the length of the evaluation steps, r satisfies the claim, and by transitivity of locality, the claim holds for v. □

Theorem 3 (Soundness of Weak Obligation)**.** *Let e be an open expression well-typed with respect to a library L. If the weak termination obligation given by Definition 3.3 holds for L and e, then e terminates modulo callbacks.*

Proof. Consider an expression e and library L for which the weak obligation holds. Let (σ, L') be a closing environment of e. Assume that every callback of e under σ is terminating, otherwise the claim holds trivially. For $e\ \sigma$ to be non-terminating, there exists a function f such that there does not exist a bound on the length of every sequence S of incomplete calls of the form: $(\lambda x.f\ \bar{w}_i)\ u_i$ as defined by Theorem 1. We now show that the length of any such sequence S is bounded by $\mathcal{M}_e(\sigma)$.

Let σ_i $(i \geq 1)$ be the parameter substitutions of every call $(\lambda x.f\ \bar{w}_i)\ u_i$ in the sequence S. The first call of the sequence should be local to $e\ \sigma$ (i.e, should be an application of an unmarked lambda). Otherwise, it is a callback of e and hence terminates by assumption. Therefore it cannot be *incomplete* and cannot be a part of S as required by Theorem 1. By part (b) of the obligation, $\mathcal{M}_f(\sigma_1) < \mathcal{M}_e(\sigma)$. Now, by the definition of the sequence, for all $i \geq 2$, S_i is local to S_{i-1}. That is, S_i is local to $\mathit{body}(f)\ \sigma_{i-1}$. By the definition of the obligation, $\mathcal{M}_f(\sigma_i) < \mathcal{M}_f(\sigma_{i-1}) < \mathcal{M}_e(\sigma)$. Since $<$ is well-founded, the length of the sequence S should be bounded by $\mathcal{M}_e(\sigma)$. □

Theorem 4 (Weak Completeness of Weak Obligation)**.** *If the weak termination obligation does not hold for a library L, there exists a library $L \cup L'$ and expressions e and e' well-typed under the library $L \cup L'$ such that e and e' are terminating modulo callbacks but $e\ e'$ is not terminating modulo callbacks.*

Proof. Let f be a function for which the weak obligation fails. Therefore, there does not exist a measure for f as required in some closing environment (σ, L'). By the definition of a closing environment, $L \cup L'$ is a library. Let e be $\lambda x.f\ (x, \sigma(x_2), \cdots, \sigma(x_n))$, where $\mathit{param}(f) = (x_1, \cdots, x_n)$. Let e' be $\sigma(x_1)$. Clearly, e and e' are terminating (and terminating modulo callbacks) since they are values. Now, $e\ e'$ does not have a bound on the lengths of sequences of chains of local calls (invoking f) starting from $\mathit{body}(f)$. This is possible only if $e\ e'$ is non-terminating, otherwise there exists a bound on every set of sequences of calls made during the evaluation of $e\ e'$ namely the number of steps in the evaluation of $e\ e'$. Since $e\ e'$ is a closed expression, it is also non-terminating modulo callbacks. □

Theorem 5 (Soundness of Strong Termination Obligation)**.** *Let e be an open expression well-typed with respect to a library L. If the strong termination obligation given by Definition 3.4 holds for L and e then e terminates modulo callbacks.*

Proof. Let us hypothesize that for any function f, if the strong obligation holds then for any call $(\lambda x.h\ \bar{w})\ s$ local to $\mathit{body}(f)$ under a closing environment (σ, L'), $\mathcal{M}_h(\sigma_{\mathit{param}}((\lambda x.h\ \bar{w})\ s)) < \mathcal{M}_f(\sigma)$. We prove this hypothesis by induction on the evaluation steps taken to invoke a call local to the body of a function f in L from $\mathit{body}(f)$. Consider a function f and a call $(\lambda x.h\ \bar{w})\ s$ that is local to $\mathit{body}(f)$ under some closing environment σ. By Lemma 2, there exists a lambda term $\ell : (\lambda x.g\ \bar{y})$ that is a sub-expression of $\mathit{body}(f)$ such that there exists a call $(\ell : \lambda x.g\ \bar{p})\ q$ that is a local call of $\mathit{body}(f)$ under σ, and $\lambda x.h\ \bar{w}$ is a local closure of the call. Let $\sigma' = \sigma_{\mathit{param}}((\lambda x.g\ \bar{p})\ q)$. Note that $(\lambda x.h\ \bar{w})\ s$ should be reachable from $\mathit{body}(g)\ \sigma'$ in fewer number of steps compared to $\mathit{body}(f)\ \sigma$. Therefore by induction hypothesis, $\mathcal{M}_h(\sigma_{\mathit{param}}((\lambda x.h\ \bar{w})\ s)) < \mathcal{M}_g(\sigma')$. By the definition of strong obligation, $\forall u.\mathcal{M}_g([\mathit{param}(g) \mapsto \bar{p}[u/x]]) < \mathcal{M}_f(\sigma)$ if $\mathit{Applicable}(L, L', \mathit{body}(f), \sigma, \ell, u)$ holds. (Notice that $\mathit{Applicable}(L, L', \mathit{body}(f), \sigma, \ell, q)$ must be true as the call $(\ell : \lambda x.g\ \bar{p})\ q$ takes place during evaluation under σ.) Instantiating u on q, $\mathcal{M}_g([\mathit{param}(g) \mapsto \bar{p}[q/x]]) = \mathcal{M}_g(\sigma') < \mathcal{M}_f(\sigma)$. (Note that the measures are oblivious to the marks on lambdas.) Hence, $\mathcal{M}_h(\sigma_{\mathit{param}}((\lambda x.h\ \bar{w})\ s)) < \mathcal{M}_f(\sigma)$, where f and h are arbitrary. Therefore, the weak obligation holds for every function $f \in L$. By a similar argument, the weak obligation also holds for e. The claim of this theorem follows from Theorem 3. □

REFERENCES

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analysing the complexity of functional programs: higher-order meets first-order. In *International Conference on Functional Programming, {ICFP}*. 152–164. DOI:http://dx.doi.org/10.1145/2784731.2784753

Gilles Barthe, Benjamin Grégoire, and Colin Riba. 2008. A Tutorial on Type-Based Termination. In *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*. 100–152. DOI:http://dx.doi.org/10.1007/978-3-642-03153-3_3

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. DOI:http://dx.doi.org/10.1007/978-3-662-07964-5

Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. 2013. An Overview of the {Leon} Verification System. In *Scala Workshop*.

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of Programming Languages, POPL*. 133–144. DOI:http://dx.doi.org/10.1145/1328438.1328457

Jürgen Giesl. 1997. Termination of Nested and Mutually Recursive Algorithms. *J. Autom. Reasoning* 19, 1 (1997), 1–29. DOI:http://dx.doi.org/10.1023/A:1005797629953

Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. 2011. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.* 33, 2 (2011), 7:1–7:39. DOI: http://dx.doi.org/10.1145/1890028.1890030

J Giesl, S Swiderski, P Schneider-Kamp, and R Thiemann. 2006. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Rewriting techniques and Applications*, Vol. 4098. 297–312.

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2004. Automated Termination Proofs with AProVE. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*. 210–220. DOI:http://dx.doi.org/10.1007/978-3-540-25979-4_15

Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types. *Studies in Logic and the Foundations of Mathematics* 63 (1971), 63–92.

Jaen-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Université Paris VII.

Jean-Yves Girard. 1990. *Proofs and types*. Vol. 7.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press, New York, NY, USA.

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages, POPL*. DOI:http://dx.doi.org/10.1145/1480881.1480898

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification (CAV)*, Vol. 7358. 781–786.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of NASA Formal Methods, NFM*. 41–55.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Principles of Programming Languages, POPL*. 223–236. DOI:http://dx.doi.org/10.1145/1706299.1706327

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore (Eds.). 2000. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers.

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI*, Mary W. Hall and David A. Padua (Eds.). ACM, 222–233. DOI:http://dx.doi.org/10.1145/1993498.1993525

Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. *Automatic Termination Verification for Higher-Order Functional Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 392–411. DOI:http://dx.doi.org/10.1007/978-3-642-54833-8{_}21

Ruslán Ledesma-Garza and Andrey Rybalchenko. 2012. Binary reachability analysis of higher order functional programs. In *International Static Analysis Symposium*. Springer, 388–404.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*. 81–92. DOI:http://dx.doi.org/10.1145/360204.360210

K Rustan M Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 348–370. DOI:http://dx.doi.org/10.1007/978-3-642-17511-4{_}20

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of Principles of Programming Languages*. ACM, 330–343.

Ravichandhran Madhavan and Viktor Kuncak. 2014. Symbolic Resource Bound Inference for Functional Programs. In *Computer Aided Verification, CAV*. 762–778. DOI:http://dx.doi.org/10.1007/978-3-319-08867-9_51

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. DOI:http://dx.doi.org/10.1007/3-540-45949-9

Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press.

Chris Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press.

Aleksandar Prokopec and Martin Odersky. 2015. Conc-Trees for Functional and Parallel Programming. In *Languages and Compilers for Parallel Computing, LCPC*. 254–268. DOI:http://dx.doi.org/10.1007/978-3-319-29778-1_16

Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *International Conference on Functional Programming, ICFP*. 165–176. DOI:http://dx.doi.org/10.1145/2364527.2364575

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification CAV*. 745–761. DOI:http://dx.doi.org/10.1007/978-3-319-08867-9_50

Wouter Swierstra. 2015. Stream: A library for manipulating infinite lists. https://hackage.haskell.org/package/Stream-0.4.7.2/docs/Data-Stream.html. (2015).

W. W. Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967), 198–212. DOI:http://dx.doi.org/10.2307/2271658

Pedro B. Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. 2015. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *European Symposium on Programming, ESOP*. DOI:http://dx.doi.org/10.1007/978-3-662-46669-8_32

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph.D. Dissertation. UNIVERSITY OF CALIFORNIA, SAN DIEGO.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *International Conference on Functional Programming, ICFP*. 269–282. DOI:http://dx.doi.org/10.1145/2628136.2628161

Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. 2015. Counter-example complete verification for higher-order functions. In *Symposium on Scala, Scala 2015*. 18–29. DOI:http://dx.doi.org/10.1145/2774975.2774978

Hongwei Xi. 2001. Dependent Types for Program Termination Verification. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. 231–242. DOI:http://dx.doi.org/10.1109/LICS.2001.932500

Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis Symposium, SAS*. 280–297. DOI:http://dx.doi.org/10.1007/978-3-642-23702-7_22