Abstract

We present a new approach for specifying and verifying resource utilization of higher-order functional programs that use lazy evaluation and memoization. In our approach, users can specify the desired resource bound as templates with numerical holes e.g. as steps \( \leq ? \cdot \text{size}(l) + ? \) in the contracts of functions. They can also express invariants necessary for establishing the bounds that may depend on the state of memoization. Our approach operates in two phases: first generating an instrumented first-order program that accurately models the higher-order control flow and the effects of memoization on resources using sets, algebraic datatypes and mutual recursion, and then verifying the contracts of the first-order program by producing verification conditions of the form \( \exists \forall \) using an extended assume/guarantee reasoning. We use our approach to verify precise bounds on resources such as evaluation steps and number of heap-allocated objects on 17 challenging data structures and algorithms. Our benchmarks, comprising of 5K lines of functional Scala code, include lazy mergesort, Okasaki’s real-time queue and deque data structures that rely on aliasing of references to first-class functions; lazy data structures based on numerical representations such as the conqueue data structure of Scala’s parallel library, cyclic streams, as well as dynamic programming algorithms such as knapsack and Viterbi. Our evaluations show that when averaged over all benchmarks the actual runtime resource consumption is 80% of the value inferred by our tool when estimating the number of evaluation steps, and is 88% for the number of heap-allocated objects.

Categories and Subject Descriptors D.2.4 [Software]: Program Verification; D.3.1 [Software]: Formal Definitions

General Terms Languages, Performance, Reliability, Verification

Keywords complexity, lazy evaluation, dynamic programming

1. Introduction

Static estimation of performance properties of programs is an important problem that has attracted a great deal of research, and has resulted in techniques ranging from estimation of resource usage in terms of concrete physical quantities [80] to static analysis tools that derive upper bounds on the abstract complexities of programs [1, 32, 35, 47]. Recent advances [7, 22, 32, 35, 74, 84] have shown that automatically inferring bounds on more algorithmic metrics of resource usage, such as the number of steps in the evaluation of an expression (commonly referred to as steps) or the number of memory allocations (alloc), is feasible on programs that use higher-order functions and datatypes, especially in the context of functional programs. However, most existing approaches aim for complete automation but trade off expressive power and the ability to interact with users. Many of these techniques offer little provision for users to specify the bounds they are interested in, or to provide invariants needed to prove bounds of complex computation, such as operations on balanced trees where the time depends on the height or weight invariants that ensure balance. This is in stark contrast to the situation in correctness verification where large-scale verification efforts are commonplace [34, 40, 41, 50]. Alternative approaches [18, 52] have started incorporating user specifications to target more precise bounds and more complex programs.

In this paper, we show that such contract-based approach can be extended to verify complex resource bounds in a challenging domain: higher-order functional programs that rely on memoization and lazy evaluation. By memoization we refer to caching of outputs of a function for each distinct input encountered during an execution, and by lazy evaluation we mean the usual combination of call-by-name (which can be simulated by lambdas with a parameter of unit type [66]) and memoization. These features are important as they improve the running time (as well as other resources), often by orders of magnitude, while preserving the functional model for the purpose of reasoning about the result of the computation. They are also ubiquitously used, e.g. in dynamic programming algorithms and by numerous efficient, functional data structures [59, 62], and often find built-in support in language runtimes or libraries. The challenge that arises with these features is that reasoning about resources like running time and memory usage becomes state-dependent and more complex than correctness—to the extent that precise running time bounds remain open in some cases (e.g. lazy pairing heaps described in page 79 of [59]). Nonetheless, reasoning about correctness remains purely functional making them more attractive and amenable to functional verification in comparison to imperative programming models. We therefore believe that it is useful and important to develop tools to formally verify resource complexity of programs that rely on these features.

Although our objective is not to compute bounds on physical time, our initial experiments do indicate a strong correlation be-
private case class SCons(x: (BigInt,Bool), tfun:() ⇒ SCons) {
  lazy val tail = tfun()
}

private val primes = SCons((1, true), () ⇒ nextElem(2))

def nextElem(i: BigInt): SCons = {
  require(i ≥ 2)
  val x = (i, isPrimeNum(i))
  val y = i + 1
  SCons(x, () ⇒ nextElem(y))
}

ensuring(r ⇒ steps ≤ ? * i + ?)

def isPrimeNum(n: BigInt): Bool = {
  def rec(i: BigInt): Bool = {
    require(i ≥ 1 & i < n)
    if (i == 1) true else (n % i != 0) & & rec(i - 1)
  }
  ensuring(r ⇒ steps ≤ ? * i + ?)
  rec(n - 1)
}

ensuring(r ⇒ steps ≤ ? * n + ?)

def isPrimeStream(s: SCons, i: BigInt): Bool = {
  def rec(i: BigInt): Bool = {
    require(i ≥ 2)
    s.tfun ≈ (i) ⇒ nextElem(i))
  }
  ensuring(r ⇒ steps ≤ ? * n + ?)
  rec(i + 2)
}

def takePrimes(i: BigInt, n: BigInt, s: SCons): List = {
  require(0 ≤ i & i ≤ n & & isPrimeStream(s, i+2))
  if (i < n) {
    val t = takePrimes(i+1, n, s.tail)
    if (s.x._2) Cons(s.x._1, t) else t
  }
  else Nil()
}

ensuring(r ⇒ steps ≤ ? * (n(n−i)) + ?)

def primesUntil(n: BigInt): List = {
  require(n ≥ 2)
  takePrimes(0, n-2, primes)
}

ensuring(r ⇒ steps ≤ ? * n² + ?)

Figure 2. Specifying properties dependent on memoization state.

Prime stream example. The class SCons shown in Fig. 1 defines a stream that stores a pair of unbounded integer (BigInt) and boolean, and has a generator for the tail: tfun which is a function from Unit to SCons. The lazy field tail of SCons evaluates tfun() when accessed the first time and caches the result for reuse. The program defines a stream primes that lazily computes for all natural numbers starting from 1 its primality. The function primesUntil returns all prime numbers until the parameter n using a helper function takePrimes, which recursively calls itself on the tail of the input stream (line 28). Consider now the running time of this function. If takePrimes is given an arbitrary stream s, its running time cannot be bounded since accessing the field tail at line 28 could take any amount of time. Therefore, we need to know the resource usage of the closures accessed by takePrimes, namely s.(tail).tfun. However, we expect that the stream s passed to takePrimes is a suffix of the primes stream, which means that tfun is a closure of nextElem. To allow expressing such properties we revisit the notion of **intensional or structural equivalence**, denoted ≈, between closures [5].

Structural equality as a means of specification. In our system, we allow closures to be compared structurally. Two closures are structurally equal if their abstract syntax trees are identical without unfolding named functions (formally defined in section 2). For example, the comparison at line 23 of Fig. 1 returns true if the tfun parameter of s is a closure that invokes nextElem on an argument that is equal to i. We find this equality to be an effective and low overhead means of specification for the following reasons: (a) Many interesting data structures based on lazy evaluation use aliased references to closures (e.g. Okasaki’s scheduling-based data structures [59, 62]). Expressing invariants of such data structures requires equating closures. While reference equality is too restrictive for convenient specification (and also breaks referential transparency), semantic or extensional equality between closures is undecidable. Structural equality is well suited in this case. (b) Secondly, our approach is aimed at (but not restricted to) callee-closed programs where the targets of all indirect calls are available at analysis time. (Section 2 formally describes such programs.) In such cases, it is often convenient and desirable to state that a closure has the same behavior as a function in the program, as was required in Fig. 1. (c) Structural equality also allows modeling reference equality of closures by augmenting closures with unique identifiers as they are created in the program.

While structural equality is a well-studied notion [5], we are not aware of any prior works that uses it as a means of specification. Using structural equality, we specify that the stream passed as input to takePrimes is an SCons whose tfun parameter invokes nextElem(i+2) (see function isPrimeStream and the precondition of takePrimes). This allows us to bound the steps of the function takePrimes to $O(n(n − i))$ and that of primesUntil to $O(n^2)$. For primesUntil, our tool inferred that steps ≤ 16n² + 28.

Properties depending on memoization table state. The quadratic bound of primesUntil is precise only when the function is called for the first time. If primesUntil(n) is called twice, the time taken
by the second call would be linear in \( n \), since every access to tail
within \texttt{takePrimes} will take constant time as it has been cached
during the previous call to \texttt{takePrimes}. The time behavior of
the function depends on the state of the memoization table (or cache)
making the reasoning about resources imperative. To specify such
resources we support a built-in operation \( \text{cached}(f(x)) \) that can
query the state of the cache. This predicate holds if the function
\( f \) is a memoized function and is cached for the value \( x \). Note
that it does not invoke \( f(x) \). The function \texttt{concrUntil}(s, i) shown
in Fig. 2 uses this predicate to state a property that holds iff the
first \( i \) calls to the tail field of the stream \( s \) have been cached.
(Accessing the lazy field \( s\text{.tail} \) is similar to calling a memoized
function \texttt{tail}(s)\.) This property holds for \( \text{primes} \) stream at the end
of a call to \texttt{primesUntil}(n), and hence is stated in the postcondition
of \texttt{primesUntil}(n) (line 7 of Fig. 2). Moreover, if this property holds
in the state of the cache at the beginning of the function, the number
of steps executed by the function would be linear in \( n \). This is
expressed using a disjunctive resource bound (line 8). Observe that
in the postcondition of the function, we need to refer to the state
of the cache at the beginning of the function, as it changes during
the execution of the function. For this purpose, we support a built-
in construct \texttt{inSt} that can be used in the postcondition to refer
to the state at the beginning of the function, and an \texttt{in} construct
which can be used to evaluate an expression in the given state.
These expressions are meant only for use in contracts. We need
these constructs since the cache is implicit and cannot be directly
accessed by the programmers to specify properties on it. On the
upside, the knowledge that the state behaves like a cache can be
exploited to reason functionally about the result of the functions,
which results in fewer contracts and more efficient verification.

Veriﬁcation Strategy. Our approach, through a series of trans-
formations, reduces the problem of resource bound inference for
programs like the one shown in Fig. 1 to invariant inference for
a strict, functional ﬁrst-order program, and solves it by applying
an inductive, assume-guarantee reasoning. The inductive reasoning
assumes termination of expressions in the input program, which is
verified independently using an existing termination checker. We
use the Leon termination checker in our implementation \cite{78}, but
other termination algorithms for higher-order programs \cite{31, 37, 68}
are also equally applicable. Note that memoization only affects
resource usage and not termination, and lazy suspensions are in fact
lambdas with unit parameters. This strategy of decoupling termin-
ation checks from resource veriﬁcation enables checking termin-
ation using simpler reasoning, and then use proven well-founded
relations during resource analysis. This allows us to use recursive
functions for expressing resource bounds and invariants, and en-
ables modular, assume-guarantee reasoning that relies on induction
over recursive calls (previously used in correctness veriﬁcation).

Contributions. The following are the contributions of this paper:

- We propose a speciﬁcation approach for expressing resource
  bounds of programs and the necessary invariants in the presence
  of memoization and higher-order functions (section 2).

- We propose a system for verifying the contracts of programs
  expressed in our language by combining and extending existing
techniques from resource bound inference and software veri-
fication (sections 3 and 4).

- We use our system to prove asymptotically precise resource
  bounds of 17 benchmarks, expressed in a functional subset
  of Scala \cite{57}, implementing complex lazy data structures and
dynamic programming algorithms comprising 5K lines of Scala
  code and 123 resource templates (section 5).

- We experimentally evaluate the accuracy of the inferred bounds
  by rigorously comparing them with the runtime values for the
  resources on large inputs. Our results show that while the
  inferred values always upper bound the runtime values, the run-
  time values for steps is on average 80% of the value inferred
  by the tool, and is 88% for alloc (section 5).

2. Language and Semantics

Fig. 3 show the syntax of a simple, strongly-typed functional lan-
guage extended with memoization, contracts and speciﬁcation con-
structs, that we will use to formalize our approach. Every expres-
sion has a static label belonging to \texttt{Vars}, \texttt{Cst} and \texttt{Fids}
respectively. We use \( e \) to denote an expression with its label. To reduce clutter, we
omit the label if it is not relevant to the context. \texttt{Tdef} shows the
syntax of user-deﬁned algebraic datatypes and \texttt{Fdef} shows the syn-
tax of function deﬁnitions. A program \( P \) is a set of functions deﬁni-
tions in which every function identiﬁer is unique, every direct call
invokes a function deﬁned in the program, and the labels of expres-
sions are unique. As a syntactic sugar, we consider tuples as a spe-
cial datatype, and denote tuple construction using \((x_1,...,x_n)\),
and selecting the \( i \)-th element of a tuple using \( x_i \).

In particular, our language supports a structural equality op-
erator \( e_1 \equiv e_2 \). Direct calls to named functions: \( f(x) \), and indirect
calls or lambda applications: \( x \). We also deﬁne an if-else operation
\texttt{if}\ \texttt{cond} \( e_1 \) \texttt{else} \( e_2 \) that is similar to a match construct with two
cases. The annotation \texttt{@memoize} serves to mark functions that have to
be memoized. Such functions are evaluated exactly once for each
distinct input passed to them at run time. The language uses call-by-
value evaluation strategy. Nonetheless, lazy suspensions can be im-
plemented using lambdas with unit parameter and memoized func-
tions. Expressions that are bodies of functions can have contracts
(or speciﬁcations). Such expressions have the form \( \{ e_1 \} \ e_2 \) where
\( e_1 \) and \( e_2 \) are the pre-and post-condition of \( e \) respectively.
The syntax of speciﬁcation expressions is given by \( E_{spec} \). The
postcondition of an expression \( e \) can refer to the result of \( e \) us-
ing the variable \( res \), and to the resource usage of \( e \) using steps and
alloc. Users can specify upper bounds on resources as templates
\( \epsilon \) that is on average 80% of the value inferred
by the tool, and is 88% for alloc (section 5).

Figure 3. Syntax of types, expressions, functions, and programs. The rule \texttt{Blk} is parametrized by the subscript \( \alpha \).

\begin{verbatim}
x, y ∈ Vars, \bar{x} ∈ Vars, c ∈ Cst (Variables & Constants)  
a ∈ TVars (Template Variables) 
f ∈ Fids, C_i ∈ Cids, i ∈ N (Function & Constructor ids) 
Tdef ::= type d := {C_1, \tau, \cdots, C_n \bar{\tau}} 
τ ∈ Type ::= Unit | Int | Bool | \tau ⇒ \tau | d 
Blk_α ::= let x := e_0 in e_1 | x match\{(C \bar{x} ⇒ e_{α_1})\} 
pr ∈ Prim ::= | + | | * | | \cdot | | \cdot | | \cdot | | ∃ 
ep ∈ Esrc ::= x | c | pr x | x eq y | f x | C \bar{x} | e_λ | x y | Blk_α 
ep ∈ Espec ::= e_s | Blk_α | cached(f x) | inSt | in(e_p, x) | res 
| steps ≤ ub | alloc ≤ ub 
ub ∈ Bnd ::= e_p | e_λ 
e_1 ∈ Etemp ::= a \cdot x + e_1 | a 
Fdef ::= (@memoize)\? \{ \epsilon \} x := \{ e_p \} e_s (e_p)
\end{verbatim}
\( \lambda x. f(x, y) \) where \( f \) is a named function whose argument is a pair (a two element tuple) and \( y \) is a captured variable.

**Notation and Terminology.** Given a domain \( A \), we use \( a \in A^n \) to denote a sequence of elements in \( A \), and \( a_i \) to refer to the \( i \)th element. (Note that this is different from tuple selector \( x_i \), which is an expression of the language). We use \( A \to B \) to denote a partial function from \( A \) to \( B \). Given a partial function \( h \), \( h(\bar{x}) \) denotes the function that applies \( h \) point-wise on each element of \( \bar{x} \), and \( h[a \mapsto b] \) denotes the function that maps \( a \) to \( b \) and every other value \( x \) in the domain of \( h \) to \( h(x) \). We use \( h[a_i \mapsto b_i] \) to denote \( h[a_1 \mapsto b_1][a_2 \mapsto b_2]\ldots[a_n \mapsto b_n] \). We omit \( h \) in the above notation if \( h \) is an empty function. We define a partial function \( h_1 \uplus h_2 \) as \((h_1 \uplus h_2)(x) = \text{if} \( x \in \text{dom}(h_2) \) then \( h_2(x) \) else \( h_1(x) \). Let \( \lambda f \cdot (e) \) denote the type of an expression \( e \) in a program \( P \). Given a lambda \( e_\lambda \), we use \( FV(e_\lambda) \) to denote the free variable captured by \( e_\lambda \) and \( \mathsf{target}(e_\lambda) \) to denote the function called in the body of the lambda. The operation \( e'[x'/e'] \) denotes the syntactic replacement of the free occurrences of \( x \) in \( e \) by \( e' \). We use \([a, b] \) to denote a closed integer interval from \( a \) to \( b \). Given a substitution \( : TVars \to Z \), we use \( e \) to represent substitution of the holes by the values given by the assignment. We also extend this notation to formulas later. We refer to programs and expressions without holes as *concrete* programs and expressions.

We now define the semantics of the language (Fig. 4) and subsequently define the problem of contract and resource verification. We use a big-step semantics (similar to Lauchby’s semantics for lazy evaluation [46]) as it naturally leads to a compositional reasoning, which is used by our approach. We also define a reachability relation on top of the big-step semantics to reason about environments that are reachable during an evaluation.

**Semantic domains.** Let \( \mathsf{Adr} \) denote the addresses of heap-allocated structures namely closures and data types. The state of an interpreter evaluating expressions of our language is a quadruple consisting of a cache \( C \), a heap \( H \), an assignment of variables to values \( \sigma \), and a set of function definitions, defined as follows:

\[ u, v \in \mathsf{Val} = \mathsf{Val}_{\mathsf{Z}} \cup \mathsf{Bool} \cup \mathsf{Adr}\]

\[ \mathsf{FVal} \equiv \mathsf{Fids} \times \mathsf{Val} \]

\[ \mathsf{DVal} \equiv \mathsf{Cids} \times \mathsf{Val}^* \]

\[ \mathsf{Clo} \equiv \mathsf{Lam} \times \mathsf{Store} \]

\[ \mathsf{Heap} \equiv \mathsf{Adr} \to (\mathsf{DVal} \cup \mathsf{Clo}) \]

\[ \mathsf{Store} \equiv \mathsf{Vars} \to \mathsf{Val} \]

\[ \mathsf{Cache} \equiv \mathsf{FVal} \to \mathsf{Val} \]

\[ \mathsf{Env} \subseteq \mathsf{Cache} \times \mathsf{Heap} \times \mathsf{Store} \times 2^{\mathsf{Val}^*} \]

We define a few helper functions on the semantic domains. Let \( \mathsf{fresh}(H) \) denote an element \( a \in (\mathsf{Adr} \setminus \text{dom}(H)) \). Let \( \mathsf{body}_1(f) \) and \( \mathsf{param}_1(f) \) denote the body and parameter of a function \( f \) defined in the environment \( \Gamma \), and \( \mathsf{Mem} \subseteq \mathsf{Fids} \) denote the set of memoized functions in the function definitions in \( \Gamma \).

**Structural equivalence.** We define a structural equivalence relation \( \approx \) on the values \( \mathsf{Val} \) with respect to a \( \mathsf{Heap} \in \mathsf{Heap} \), as explained below. We say two addresses are structurally equivalent iff they are bounded to structurally equivalent values in the heap. Two datatypes are structurally equivalent iff they use the same constructor and their fields are equivalent. Two closures are structurally equivalent iff their lambdas are of the form \( \lambda x. f(x, y) \) and \( \lambda x. f(u, z) \) and the captured variables \( y \) and \( z \) are bound to structurally equivalent values. Formally, (script omitted below for clarity)

\[ \forall a \in \mathbb{Z} \cup \mathsf{Bool}. a \approx a \]

\[ \forall (a, b) \subseteq \mathsf{Adr}. a \approx b \iff \mathsf{Heap}(a) \approx \mathsf{Heap}(b) \]

\[ \forall f \in \mathsf{Fids}. (a, b) \subseteq \mathsf{Val}. (f a) \approx (f b) \iff a \approx b \]

\[ \forall c \in \mathsf{Cids}. (a, b) \subseteq \mathsf{Val}. (c a) \approx (c b) \iff \forall i \in [1, n]. a_i \approx b_i \]

\[ \forall (e_1, e_2) \subseteq \mathsf{Lam}. \forall (\sigma_1, \sigma_2) \subseteq \mathsf{Store}. (e_1, e_2) \approx (e_2, \sigma_2) \iff \mathsf{target}(e_1) = \mathsf{target}(e_2) \land \sigma_1(FV(e_1)) \approx \sigma_2(FV(e_2)) \]
This equivalence satisfies congruence properties with respect to the result and resource usage of expressions (formalized in Appendix A).

Judgements. We use judgements of the form $\Gamma \vdash e \Downarrow_p v, \Gamma'$ to denote that under an environment $\Gamma \in \mathit{Env}$, an expression $e$ evaluates to a value $v \in \mathit{Val}$ and results in a new environment $\Gamma' \in \mathit{Env}$, while consuming $p \in \mathbb{Z}$ units of a resource. When necessary we expand $\Gamma$ as $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F)$ to highlight the individual components of the environment. We omit any component of the judgement that is not relevant to the discussion when there is no ambiguity. In Fig. 4, we omit the function definitions from the environment as they do not change during the evaluation.

Resource parameterization. We parametrize the operational semantics in a way that it can be instantiated on multiple resources using the following parameterization functions: (a) A cost function $c_{\text{op}}$ that returns the resource requirement of an operation op such as cons or app. $c_{\text{op}}$ may possibly have parameters. In particular, we use $c_{\text{match}}(i)$ to denote the cost of a match operation when the $i$th case was taken, which should include the cost of failing all the previous cases. (b) A resource combinator $\oplus : \mathbb{Z}^* \rightarrow \mathbb{Z}$ that computes the resource usage of an expression by combining the resource usage of the sub-expressions. Typically, $\oplus$ is either $+$ or $\max$.

We specifically consider two resources in this paper: (a) the number of steps in the evaluation of an expression denoted steps, and (b) the number of heap-allocated objects (viz. a closure, datatype or a cache entry) created by an expression denoted alloc. In the case of steps, $c_{\text{let}}$ and $c_{\text{var}}$ are zero as the operations are normally optimized away or subsumed by a machine instruction. $c_{\text{op}}$ is 1 for every other operation except $c_{\text{miss}}$ and $c_{\text{match}}(i)$. We consider datatype construction and primitive operations on big integers as unitary steps. We define $c_{\text{match}}(i)$ proportional to $i$ as we need to include the cost of failing all the $i - 1$ match cases. In the case of alloc, $c_{\text{op}}$ is 1 for datatype and closure creations and also for a cache miss since it allocates a cache entry. It is zero otherwise. For both resources, the operation $\oplus$ is defined as addition ($+$). Our implementation, however, supports other resources such as abstract stack space usage and number of recursions.

Memoized Call Semantics. For brevity, we skip the discussion of straightforward semantic rules shown in Fig. 4 and focus on rules that are atypical. The semantics of calling a memoized function is defined by the rules: MEMOCALL_HIT and MEMOCALL_MISS. Calling a memoized function involves as a first step querying the cache for the result of the call. In case the result is not found, the callee is invoked, and the cache is updated once (and if) the callee returns a value. Querying the cache involves comparing arguments of the call for equality. We define a lookup relation $\mathcal{E}_P$ that uses structural equivalence to lookup the cache as follows: $(f u) \in \mathcal{E}_P \Leftrightarrow \exists u' \in \mathit{Val}. (f u') \in \mathit{dom}(\mathcal{C}) \land u' \approx u$. We parameterize the cost of searching and updating the cache using the parameters $c_{\text{hit}}$ and $c_{\text{miss}}$. To calculate the steps resource, we consider lookup and update as unitary steps, and hence define $c_{\text{miss}} = 2$ (as it involves a lookup and an update operation) and $c_{\text{hit}} = 1$. In general, $c_{\text{miss}}$ and $c_{\text{hit}}$ may depend on the values of the arguments.

Specifications. The construct $\text{cached}(f x)$ evaluates to true in an environment $\Gamma$ iff the call $f x$ is cached for the value of $x$ in $\Gamma$. Observe that the resource consumption of this construct is zero. This is because the construct is syntactically excluded from being part of the implementation of functions (see Fig. 3) which renders its resource usage irrelevant. The rule CONTRACT defines the semantics of an expression $\tilde{e}$ of the form $\{\text{pre} \} e \{\text{post} \}$. The expression evaluates to a value $v$ only if $\text{pre}$ holds in the input environment and $\text{post}$ holds in the environment resulting after evaluating $e$. Observe that the value, cache effects, and resource usage of $\tilde{e}$ are equal to that of $e$. Also note that the resource variables steps and alloc are bound to the resource consumption of $e$ before evaluating the postcondition. The construct inSt is used by expressions in the postcondition to refer to the state of the cache at the beginning of the function, and $\text{in}(e, x)$ evaluates an expression $e$ in the cache state given by $x$, as illustrated by the example shown in Fig. 2. For brevity, we omit the formal semantics of the constructs in and inSt.

Appendix D formalizes their semantics along with a match construct fsmatch based on structural equality. In the rest of the section, using the big-step semantics, we introduce a few concepts that are used in this paper, and formally define the problem of resource verification for open programs.

Reachability Relation. We define a relation $\rightsquigarrow$ (similar to the calls relation of Sereni, Jones and Bohr [37, 68]) that characterizes the environments that may reach an expression during an evaluation. For every semantic rule shown in Fig. 4 with $n$ antecedents: $A_1 \cdots A_m B_1 \cdots B_n$, where $A_1 \cdots A_m$ are not big-step reductions, and each $B_i, i \in [1, n]$, is a big-step reduction of the form: $\Gamma_i \vdash e_i \Downarrow_{p_i} v_i, \Gamma_i'$, we introduce $n$ rules for each $1 \leq i \leq n$:

$$A_1 \cdots A_m \cdot B_1 \cdots B_{n-1} \underbrace{\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma, e_1 \rangle}_{1} \cdot \cdots \cdot \underbrace{\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma, e_n \rangle}_{n}$$

Let $\rightsquigarrow$ represent the reflexive, transitive closure of $\rightsquigarrow$. We say that an environment $\Gamma'$ reaches $e'$ during the evaluation of $e$ from $\Gamma$ iff $(\Gamma, e) \rightsquigarrow^* (\Gamma', e')$. We say that the evaluation of $e$ under $\Gamma$ diverges iff there exists an infinite sequence $(\Gamma_1, e_1) \rightsquigarrow (\Gamma_2, e_2) \rightsquigarrow \cdots$. We say an expression $e$ (or a function $f$) terminates iff there does not exist a $\Gamma \in \mathit{Env}$ under which $e$ (or body$_f(f)$) diverges [68].

Valid environments. In reality, the environments under which an expression is evaluated satisfies several invariants which are ensured either by the runtime (like the invariant that the cached values of function calls correctly represent their results), or by the program under execution. Similar to prior works on data structure verification [39], we define the problem of contract/resource verification only with respect to such valid environments under which an expression can be evaluated. Let $P_e = P' \mid P$ denote a closed program obtained by composing a client $P'$ with an open program $P$. The evaluation of a closed program $P_e$ starts from a distinguished entry expression $e_{\text{entry}}$ (such as a call to the main function) under an initial environment $\Gamma_{P_e} : (\emptyset, \emptyset, \emptyset, F)$ where $F$ is the set of function definitions in the program $P_e$. We define the valid environments of an expression $e$ belonging to an open program $P$, denoted $\mathit{Env}_P, e$, as $\Gamma : \exists P'. (\Gamma \uparrow \Gamma \mid P, e_{\text{entry}}) \rightsquigarrow (\Gamma, e) \}$. When an expression belonging to a type correct program is evaluated under a valid environment, there are only two reasons why its evaluation may be undefined as per the operational semantics (provided the primitive operations are total): (a) the evaluation diverges, or (b) there is a contract violation during the evaluation.

Contract verification problem. Given a program $P$ without templates. The contract verification problem is to decide for every function defined in the program $P$ of the form $\text{def } f x := e$, where $\tilde{e} = \{\text{pre} \} e \{\text{post} \}$, whether in every valid environment that reaches $\tilde{e}$ in which $\text{pre}$ does not evaluate to false, $e$ evaluates to a value. Formally, $\forall \mathcal{V} : (\mathcal{C}, \mathcal{H}, \sigma, F) \in \mathit{Env}_P, e \vdash v. \left( (\Gamma \uparrow \Gamma \mid \text{pre} \Downarrow \text{false}) \vee \Gamma \uparrow \tilde{e} \Downarrow \tilde{v} \right)$ (We omit the quantification on $v$ when there is no ambiguity.) Since contracts in our programs can specify bounds on resources, the above definition also guarantees that the properties on resources hold.

Resource inference problem. Recall that we allow the resource bounds of functions to be templates. In this case, the problem is to find an assignment $\tilde{e}$ for the holes such that in the program
obtained by substituting the holes with their assignment, the contracts of all functions are verified, as formalized below. Let \( e \downarrow \) denotes substituting the holes in an expressions \( e \) with the assignment given by \( \downarrow \). The resource bound inference problem is to find an assignment \( \downarrow \) such that for every function \( \text{def } f x := \{\text{pre}\} \in \{\text{post}\} \) where \( \text{post} \) may contain holes, \( \forall \Gamma \in \text{Env}. \left( \Gamma \vdash \text{pre} \Downarrow \right) \lor \Gamma \vdash \{\text{pre}\} \in \{\text{post}\} \Downarrow v \).

**Encapsulated Calls.** Our approach is primarily aimed at programs where the targets of all indirect calls that may be executed are available at the time of the analysis. This includes whole programs that take only primitive valued inputs/parameters, and also data structures that use closures internally but whose public interfaces do not permit arbitrary closures to be passed in by their clients such as the program shown in Fig. 1 and lazy queues [59, 62]. We formalize this notion below. We say an indirect call \( e = y \) belonging to a program \( P \) is an **encapsulated call** if in every environment \( \Gamma : (C, \mathcal{H}, \sigma, F) \in \text{Env}_{x.P} \), if \( \mathcal{H}(e(\sigma)) \) is a closure \( (e_\Gamma^{\downarrow}, \sigma') \), \( l \in \text{labels}_P \). A program \( P \) is call encapsulated iff every indirect call in \( P \) is encapsulated. In our implementation, we perform a type-level static analysis that leverages access modifiers like private to identify encapsulated calls. E.g. for the program shown in Fig. 1 our tool infers that the type \( () \Rightarrow \text{SCons} \) should be assigned a closure created within the program based on the fact that no parameter of public constructors or methods has this type or any of its subtype. Therefore, it identifies that the call `tfunc()` at line 2 of Fig. 1 is an encapsulated call.

### 3. Generating Model Programs

In the following sections, we describe our approach in two phases: **model generation phase** (discussed in this section) and **verification phase** (discussed in section 4). The goal of the model generation phase is to generate a first-order program with recursion that accurately models the resource usage of the input program without any abstraction, only using theories suitable for automated reasoning. We refer to output of this phase as the **model**. In particular, there are three reductions that are handled by this phase: (a) Defunctionalization of higher-order functions to first-order functions [64], (b) Encoding of cache as an expression that changes during the execution of the program, and (c) Instrumentation of expressions with their resource usage while accounting for the effects of memoization. We formally establish the soundness and completeness of the translation with respect to the operational semantics shown in Fig. 4 by establishing a bisimulation between the input program and the model (Theorem 2). In contrast to related works [7], which use defunctionalization as a means to estimate the resource usage of input programs, here we are only interested in the values (and not resources) of expressions of the model. The expressions of the model themselves track the resource usages.

**Model Language.** The model language is similar to the source language without higher-order features, memoization, and special specification constructs (i.e., \( E_{\text{spec}} = E_{\text{pre}} \)). However, we introduce two features that were not a part of the source language: (a) set values and set primitives such as union \( \cup \) and inclusion \( \subseteq \), and (b) an error construct that halts the evaluation. The values of the model language includes \( \text{Val} \) and also sets of values of the source language (\( \text{Set} = 2^{\text{Val}} \)). The environments of the model do not have the cache component, i.e., \( \Gamma \in \text{Env} = \text{Heap} \times \text{Store} \times 2^{\text{def}.} \).

**Illustrative Example.** We use the constant-time take operation on a stream shown in Fig. 6 to illustrate the construction of the model, and later in section 4 to illustrate the verification of the model. Fig. 6(a) shows the take operation in the toy language used in the formalism, and Fig. 6(b) shows the model program explained in this section. In a real language, the function tail would be implemented as a lazy field of the SCons constructor as shown in Fig. 1. But for the purpose of verification, we treat it as a memoized function with a single argument as shown here. The function `containsUntil`, which is omitted, is similar to the Scala function shown in Fig. 2 that checks if the tail function is memoized for the first \( n \) suffixes of a stream. Observe that the lazy take operation (unlike `takePrimes`) returns a (finite) stream with the first element and a suspension of take, which when accessed constructs the next element. It requires that the input stream is memoized at least until \( n \) in order to achieve a constant time bound. Otherwise, the call to tail at line 26 may result in a cascade of calls to take (via app). The challenge here is to verify that such cascade of calls cannot happen. The take operation with these contracts is in fact used by the Okasaki’s persistent `Deque` data structure ([59] Page 111) that runs in worst-case constant time.

**Closure encoding.** We represent closures using algebraic datatypes in a way that preserves the structural equivalence of closures. We say two lambdas \( e_\lambda = \lambda x. f(x, y) \), and
\(\text{type Stream := (SCons (BigInt, Unit ⇒ Stream), SNil)}\)
\[@\text{memoize}\]
def tail s = s match \{ SNil ⇒ SNil; SCons (x, tfun) ⇒ (tfun(Unit)); \}
def take (n, s) =
  \{ concurUntill (s, n) \}
if \(n \leq 0\) SNil else \(s\) match \{
  SNil ⇒ SNil;
  SCons (x, tfun) ⇒
  let t := tail s in
  let n1 := n - 1 in SCons(x, λa.take (n1, t));\}
\(\{\}\) \(\text{steps ≤ ?}\)
(a) A constant-time, lazy take operation
def tail2 (s, st) = s match \{ SNil ⇒ SNil;
  SCons (x, tfun) ⇒ app (tfun, Unit, st); \}
def app (cl, x, st) = cl match\{ Take (n1, s1) ⇒ take2 (n1, s1, st);\}
def take2 (n, s, st) =
  \{ concurUntill2 (s, n, st) \}
if \(n \leq 0\) SNil, st, 3 else \(s\) match \{
  SNil ⇒ SNil, st, 5;
  SCons (x, tfun) ⇒
  let u := tail2 (s, st) in
  let res := \(\text{tag}\{\}\) \(\text{res, 3 ≤ ?}\) \)
Figure 6. Illustration of the translation shown in Fig. 5.
\(\text{e}'_1 = \lambda x. f'(x, z)\) are compatible, and denote it as e\(_\lambda\) \(\approx e'_\lambda\), iff they invoke the same targets i.e, \(f \approx f'\). This relation is interesting because during any evaluation two closures could be structurally equivalent iff their lambdas are compatible i.e, \(e\_\lambda \approx e'\_\lambda\) iff \(\exists H, \sigma, \sigma' s.t. \(e,\sigma \approx H(\_\lambda, e',\sigma')\). In the generated model we ensure that the closures of lambdas that are compatible are represented using the same datatype. For each lambda \(e\_\lambda\), we define a representative denoted \(e\_\lambda /\_\rho\) of the equivalence class with respect to \(\approx\) that belongs to a program \(P\). (It is undefined if \(P\) does not have a compatible lambda.) For each function type \(\tau = A ⇒ B\) used in \(P\), we add a datatype \(d\_\tau\) to the model (defined shortly), and replace every use of \(\tau\) in the input program by the datatype \(d\_\tau\).
\[\{ e\_\lambda, \{ i \mid i \in [1, n] \} \} \text{ the representatives (with respect to } \approx \text{) of the lambda terms in the program \(P\) that are of type } \tau \text{, and let } \{ i, \{ i \mid i \in [1, n] \} \} \text{ be their labels. The datatype } d\_\tau\text{ has } n + 1 \text{ constructors denoted } C_{1, i} \text{, } i \in [1, n] \text{ and } C_\tau. \text{That is, } d\_\tau\text{ is of the form: the type } d\_\tau := \langle C_{1,1}, \ldots, C_{n, n}, C_\tau \rangle. \text{The } i^{th} \text{ constructor } C_{1, i} \text{ represents the closure of the } i^{th} \text{ lambda term } \lambda x\_i. \text{The parameter of the constructor represents } FV(\lambda x\_i). \text{The type } C_\tau \text{ is obtained by recursively replacing the function types by their closure datatypes in } typeP(FV(\lambda x\_i)). \text{The } (n + 1)^{th} \text{ constructor } C_\tau\text{ of } d\_\tau\text{ is a stub for a closure created outside the program under analysis and serves to handle an error case (explained shortly). In Fig. 6(b), the datatype \text{tStream defined at line 14 represents the closures of lambdas of type } \text{Unit ⇒ Stream. The constructor } \text{Take of Stream represents the closure of } \lambda a.\text{take } (n1, t) \text{ created at line 12. As shown at line 29, the lambda is replaced by an instance of Take in the model. The constructor } \text{Other represents the stub closure } C_\tau.\]
are the result of adding up all the constants in the instrumented expressions along the same branch (or match case) in the program.

**Defunctionalization.** We translate an indirect call: \( x y \) to a guarded disjunction of direct calls through a process known as defunctionalization [64]. We replace every indirect call \( x y \) with label \( l \) by a call to a dispatch function \( App_l \) constructed as follows. The parameters of the function are (a) a closure \( cl \) of type \( d_r \), where \( \tau = \text{type}_P(x) \), (b) the argument of the call \( w \), and (c) a state parameter \( st \) denoting the state of the cache at the entry of the function. The dispatch function matches the closure \( cl \) to each possible constructor and in each case \( C_{l} \), where \( l \) is the label of the lambda \( \lambda a_i.e_i \) represented by the constructor, invokes the expression \( [e_i]_P \bar{a} st \) where \( e_i \) is the result of replacing in \( e_i \), the parameter of the lambda \( a_i \) with \( x \) and the free variable of the lambda with the field of \( C_{l} \). If the closure matches \( C_{l} \), the model halts with an error as this case corresponds to the scenario where a function not defined within the program \( P \) is applied to an argument. Such a function, being arbitrary, may either not terminate or can have a precondition that is violated by the arguments it is applied to. The model soundly flags this case as an error. We eliminate this case if we can statically infer (based on type encapsulation) that the targets of the closures are strictly within the program under analysis. Observe that in Fig. 6 the call to \( \text{tfn} \) inside the function \( \text{tail} \) is translated to a call to the dispatch function \( \text{app} \). (The case Other is omitted in \( \text{app} \) as we assume that the call is encapsulated.) Even though the set of possible cases in the function \( \text{App}_l \) could be large, many of those cases that are not feasible at runtime are not explored by our underlying verifier (section 4) which uses targeted unfolding [52] to unfold calls only along satisfiable (abstract) paths.

**Soundness and Completeness of the Model.** We now establish the soundness and completeness of the model for verification of contracts of an input program \( P \). The proofs of all theorems that follow are presented in Appendix B. Let \( P \) be a program. Let \( \{ H, H^p \} \subseteq \text{Heap} \). Define a relation \( \sim \) on the semantic domains as follows: (subscripts omitted below for clarity)

1. \( v a \in \mathbb{Z} \cup \text{Bool} \), \( a \sim a \)
2. \( v c \in \text{Cls}, \{ a_i \} \subseteq Vals^p \), \( c \sim c \) \( \forall v \in \{1, \ldots, n\}, a_i \sim b_i \)
3. \( \forall (e, \sigma) \in \text{Closure} \), \( \forall v \in \text{Vals}, l \in \text{labels}_p(e, \sigma) \sim C_l \) \( \forall v \) \( \sigma(FV(e)) \sim v \lor \sigma(x) \in \text{labels}_p(e, \sigma) \) is defined and has label \( l \)
4. \( \forall f \in \text{Fids} \) defined in \( P, \{ a, b \} \subseteq \text{Vals} \), \( f a \sim C_f b \) \( \forall f a \sim b \)
5. \( \forall c \in \text{Cache}, S \subseteq C \sim S \) \( \forall c \in \text{Cache} \), \( S \subseteq C \sim S \)
6. \( \forall a, b \) \( \subseteq \text{Adr} \), \( a \sim b \) \( \forall h(a) \sim H(a) \)
7. \( \forall \sigma, \sigma^t \subseteq \text{Store} \), \( \sigma \sim^t \sigma \) \( \forall \sigma, \sigma^t \subseteq \text{Store} \), \( \sigma \sim^t \sigma \)

The relation formally captures that a cache is simulated by a set of \( \text{Set} \).

**Theorem 1 (Bisimulation).** Let \( P \) be a program. Let \( e, st \) and \( e' \) be expressions such that \( Let e' = \llbracket e \rrbracket_P st \). Let \( \Gamma \in \text{Env} \) and \( \Gamma^p \in \text{Env}^p \) be such that \( \Gamma \vdash s \downarrow \downarrow v, S \) and \( \Gamma \sim_P (\Gamma^S, S) \).

(a) If \( \Gamma \vdash e \Downarrow v, \Gamma_o \) then \( \exists \Gamma_o \in \text{Env}^p, u \in \text{DVal} \) such that \( \Gamma \vdash e' \Downarrow u, \Gamma^p \) and

\[
\Gamma_o \sim_P (\Gamma^Z_o, u, 1, 1) \quad \Gamma \sim u, 1, 1 \quad p = u, 1
\]

(b) If \( \Gamma \vdash e' \Downarrow u, \Gamma_o \) then \( \exists \Gamma_o \in \text{Env}, v \in \text{Val}, p \in \mathbb{N} \) such that \( \Gamma \vdash e \Downarrow v, \Gamma, \Gamma \) and

\[
\Gamma_o \sim_P (\Gamma^Z_o, u, 1, 1) \quad \Gamma \sim u, 1, 1 \quad p = u, 1
\]

Using the above theorem, we now establish that for every function \( f \) in the program \( P \), verifying the contracts of its translation \( f^2 \) will imply that the contracts of \( f \) hold and vice-versa. A tricky aspect here is that there exist valid environments \( \Gamma \in \text{Env} \) that binds addresses to lambdas not in the scope of the program \( P \) under which \( f \) evaluates to a value. Such environments do not have any counterparts (with respect to \( \sim_P \)) in \( \text{Env}^p \). The following theorem holds despite this because if such lambdas are invoked by \( P \), the contracts of \( f \) and \( f^2 \) do not hold for all environments as there exists an environment each in \( \text{Env} \) and \( \text{Env}^p \) that results in a contract violation in \( f \) and enforces the error condition in \( f^2 \) respectively.

**Theorem 2 (Model Soundness and Completeness).** Let \( P \) be a program and \( P^p \) the model program. Let \( \bar{e} = \{ p \} e (s) \) and \( \bar{e}' = \{ p' \} e' (s') \). Let def \( f := \bar{e} \) be a function definition in \( P \), and let def \( f^2 (x, st) := \bar{e}' \) be the translation of \( f \), where \( st \) is the state parameter added by the translation.

\[
\forall \Gamma^2 \in \text{Env}^p, \forall p, \exists v. \forall \Gamma \vdash \Gamma^2 \Downarrow false \lor \Gamma \vdash \bar{e}' \Downarrow u \iff \\
\forall \Gamma \in \text{Env} \downarrow p, \exists v. \forall \Gamma \vdash \Gamma^2 \Downarrow false \lor \Gamma \vdash \bar{e} \Downarrow v
\]

Appendix B has the proofs of the above theorems. A corollary of the above theorem is that the model is sound and complete for the inference of resource bounds. That is, for any assignment to \( \bar{e} \), \( \forall \Gamma^2 \in \text{Env}^p, \forall p, \exists v. \forall \Gamma \vdash \Gamma^2 \Downarrow false \lor \Gamma \vdash \bar{e}' \Downarrow u \iff \\
\forall \Gamma \in \text{Env} \downarrow p, \exists v. \forall \Gamma \vdash \bar{e} \Downarrow v
\]

## 4. Model Verification and Inference

In this section, we discuss our approach for verifying contracts and inferring constants in the resource bounds of the model programs.

**Modular reasoning for first-order programs.** Approaches based on function-level modular reasoning for first-order programs verify the postcondition of each function \( f \) in the program under the assumption that the precondition of \( f \) and the pre-and post-condition of the functions called by \( f \) (including itself) hold at all call sites. The precondition of each function is verified at their call sites independently. This assumption/guarantee reasoning is essentially an inductive reasoning over the calls made by the functions, which would be well-founded and hence sound only for terminating evaluations of the function bodies (also referred to as partial correctness). (Section 2 formally defines termination.) The termination of functions in the program is also verified independently. We now formalize this reasoning and subsequently present an extension for handling defunctionalized programs more effectively.

Let \( e_1 \) and \( e_2 \) be two properties i.e., boolean-valued expressions. Let \( e_1 \rightarrow e_2 \) denote that whenever \( e_1 \) does not evaluate to false, \( e_2 \) evaluates to true i.e., \( \forall \Gamma \in \text{Env}^2, \Gamma \vdash e_1 \Downarrow false \lor \Gamma \vdash e_2 \Downarrow true \). (The operation \( \rightarrow \) can be considered as an implication with respect to the operational semantics of the model language.) We use \( \vdash \) \( e_1 \rightarrow e_2 \) to denote that under the assumption that all functions in \( P \) terminate and that the pre-and post-condition of callees hold at all call sites in \( P \), \( e_1 \rightarrow e_2 \) is guaranteed. The modular reasoning described above corresponds to the following two rules:

**Function-level modular reasoning:**

- For each def \( f = \llbracket e \rrbracket = \llbracket e/pre \rrbracket \rightarrow \text{post} \llbracket e/\text{res} \rrbracket \)
- For each call site \( c = f \) in \( P \), \( \llbracket e/\text{path}(c) \rrbracket \rightarrow \text{pre}(c) \)
Recall that the variable res refers to the result of e in the postcondition of e. For a call c = f x, we use pre(e) to denote the precondition of f after parameter translation. The path condition path(c) denotes the static path (possibly with disjunctions and function calls) to e from the entry of the function containing c. For instance, the path condition of the call tail²((s, st)) at line 26 of the program shown in Fig. 6(b) is: concr Until²(s, n, st) ∧ n > 0 ∧ s = SCons(x,t,fun). For programs with templates, the assume/guarantee assertions generated as above would have holes (TVars).

The goal is then to find an assignment τ for holes such that all assume/guarantee assertions of all functions are valid. (For brevity, we have omitted the formal definition of the assumptions and path as they are commonly known. Appendix C presents their formal definition.)

Observe that this modular reasoning requires that the assume/guarantee assertions hold for all environments Γ ∈ Env² (by the definition of →), even though for contract verification it suffices to consider only valid environments that reach the function bodies. (However, Γ can be assumed to satisfy invariants ensured by the runtime, e.g., that the variables in the environment are bound to type-correct values etc.) This means that pre-and post-conditions of functions should capture all necessary invariants maintained by the program. This obligation dramatically increases the specification/verification overhead when applied as such to the model programs. For example, consider the call to take² within app at line 20 in the program shown in Fig. 6(b). The path condition to the call is not strong enough to imply the precondition of the call namely concr Until²(s1,n1,st). To make this example verify, it would in fact require concr Until² to hold on the arguments of every instance of Take reachable from the recursive datatype Stream, due to the mutual recursion between app, take² and tail². That is, the precondition of app would need a function pre (cl,st) defined as follows:

\[
\text{def \ pre (cl, st) = cl match} \{ \\
\text{Take(n1,s1) } \Rightarrow \text{concr Until²(s1,n1,st)} \\
\text{(s1 match} \{ \text{SCons(x,t,)} \Rightarrow \text{pre (t, st); SNil } \Rightarrow \text{true)}; \\
\}
\]

This scenario happens very often when dealing with recursive, lazy data structures [59]. Our initial attempts to synthesize a precondition such as the above for App functions resulted in formulas too complicated for the state-of-the-art SMT solvers to solve. In the sequel, we discuss an approach to alleviate this specification overhead based on the observation that the property concr Until² actually holds at the points where the closure Take is created and is monotonic with respect to the changes to the cache.

**Cache Monotonic Properties.** Informally, a property \( p \) in \( E_{\text{spec}} \) is cache monotonic iff whenever it holds in an environment with a cache \( C_1 \), it also holds in all environments where the cache has more entries than \( C_1 \). These properties are interesting because once established they can be assumed to hold at any subsequent point in the evaluation (similar to heap-monotonic type states introduced by Fähndrich and Leino [27]). We find that in almost all cases the properties that are needed to establish resource bounds are (or can be converted to) cache monotonic properties. Intuitively, this phenomenon seems to result from anti-monotonicity of resource usage i.e, the resource usage of an expression cannot increase when it is evaluated under a cache that has more entries. Below we formalize cache monotonicity and later describe how we exploit it in verification. Let \( \Gamma_1 : (C_1, \mathcal{H}_1, \sigma_1, F) \) and \( \Gamma_2 : (C_2, \mathcal{H}_2, \sigma_2, F) \). We say \( \Gamma_1 \subseteq \Gamma_2 \) iff every component of \( \Gamma_2 \) has more entries than the corresponding component of \( \Gamma_1 \), i.e., \((k, v) \in C_1 \Rightarrow (k, v) \in C_2\), where \( C \) could be \( \mathcal{H}, \mathcal{C} \) or \( \sigma \). A property \( pr \) is cache monotonic iff \( \forall (\Gamma_1, \Gamma_2) \subseteq E_{\text{env}}. (\Gamma_1 \subseteq \Gamma_2 \wedge \Gamma_1 \vdash pr \Downarrow \text{true}) \Rightarrow \Gamma_2 \vdash pr \Downarrow \text{true}. \) To check if a property \( pr \) is cache monotonic it suffices to check the following property on the translation of \( pr \) with respect to \( \llbracket p \rrbracket_\text{env} \) defined in Fig. 5:

\( \llbracket st \subseteq st_2 \wedge \llbracket pr \rrbracket_{\text{pr}, st_1} \rightarrow \llbracket pr \rrbracket_{\text{pr}, st_2} \).

**Creation-dispatch rule for encapsulated calls.** Recall that each indirect call \( x \rightarrow y \) has a set of target lambdas that are estimated at the time of model construction based on \( \text{type}(x) \). Let \( A = \{ e_i \mid i \in [1, n] \} \), where \( e_i = \lambda x.f_i(x, y_i) \), be the lambdas in the program that are the possible targets of encapsulated calls in a program \( P \) (defined in section 2). Let \( \text{Clo}^0 \{ Ci, wi \mid i \in [1, n] \} \) be the closure constructions in the model of \( P \) representing the lambdas \( A \). In the model program, the dispatch functions \( \text{App}_i \) corresponding to the encapsulated calls invoke the function \( f_i^x \) (the translation of \( f_i \)) in each case \( Ci, wi \) (see Fig. 5 and the illustration Fig 6(b)). Let \( \text{DispCalls} = \{ f_i^x (x, z_i, st) \mid i \in [1, n] \} \) be the calls invoked by such \( \text{App}_i \) functions. Let \( \text{Props} = \{ \rho_i \mid i \in [1, n] \} \) be a set of boolean-valued expressions (properties) in \( E_{\text{spec}} \) defined on the captured argument \( y_i \) of the lambda \( e_i \in A \) (i.e., \( \rho_i \) has only \( y_i \) as free variable). We augment the function-level assume/guarantee rules with the following condition: if each property \( \rho_i \) is cache monotonic, and hold at the point of creation of the lambda \( e_i \) for the state of the cache at that point, it can be assumed to hold at the point of dispatch. Formally.

**Modular reasoning with creation-dispatch rule**

I. For each def \( \text{f x := \{ pre \} pre \rightarrow \text{post|res} } \) \( \rightarrow \llbracket \text{path} \rrbracket \rightarrow \llbracket \text{pre} \rrbracket \)

II. For each call site \( c \notin \text{DispCalls} \) \( \llbracket \text{path} \rrbracket (c) \rightarrow \llbracket \text{pre} \rrbracket (c) \)

III. (Cache monotonicity) For each \( \rho_i \in \text{Props} \)

\( \llbracket \text{pre} \rrbracket (\llbracket \text{st} \subseteq \text{st}_2 \wedge \llbracket \rho_i \rrbracket_{\text{pr}, \text{st}_1} \rightarrow \llbracket \rho_i \rrbracket_{\text{pr}, \text{st}_2} \)

IV. For each closure construction site \( c = Ci, wi \) in \( \text{Clo}^0 \) \( \llbracket \text{path} \rrbracket (c) \rightarrow (\llbracket \rho_i \rrbracket_{\text{pr}, \text{st}(c)} \)

V. For each call site \( c = f_i^x (x, z_i, st) \) in \( \text{DispCalls} \)

\( \llbracket \text{pre} \rrbracket (\llbracket \text{path} \rrbracket (c) \wedge (\llbracket \rho_i [z_i/y_i] \rrbracket_{\text{pr}, \text{st}}) \rightarrow \llbracket \text{pre} \rrbracket (c) \)

In the above rules, \( \text{st}(c) \) denotes the cache-state expression propagated by the translation function \( \llbracket \cdot \rrbracket_{\text{pr}} \rightarrow \text{expr} \) to an expression \( e \) in the model program. Note that there is exactly one cache-state expression reaching every point in the model program by the definition of the translation shown in Fig. 5. For instance, the state expression reaching the line 20 of Fig. 6(b) is \( st_2 \), whereas the state expression reaching the line 29 is \( st_3 \).

While the above reasoning holds irrespective of the how the properties \( \rho_i \) are chosen for each lambda \( e_i \), we use a particular strategy in our implementation. For each \( e_i = \lambda x.f_i(x, y_i) \), we choose \( \rho_i \) to be the disjuncts of the precondition of the call \( f_i(x, y_i) \) that only refer to the captured variable \( y_i \). E.g. for the model shown in Fig. 6(b), our approach would verify that (a) \( \text{concr Until} \) is a cache monotonic property: \( \llbracket \text{pre} \rrbracket (\llbracket \text{st} \subseteq \text{st}_2 \wedge \text{concr Until}(i, i, \text{st}_1) \rightarrow \text{concr Until}(i, i, \text{st}_2), \) and (b) that the property \( \text{concr Until}(u_{1, 1} \rightarrow n_{1, 1} \rightarrow 1) \) holds at the point of creation of the closure \( \text{Take}(n \rightarrow 1, u_1) \) at line 29. The property \( \text{concr Until}(s_1, n_1, \text{st}) \) is assumed to hold while checking...
the precondition of call to take\(\overline{x}\) at line 20. With this extension we do not need any more preconditions than what is stated in the program to verify the program.

**Theorem 3 (Soundness of creation-dispatch reasoning).** Let \(P\) be a program and \(P^1\) the model program. Let \(def \overline{f}^1 \overline{x} := \overline{e}\) where \(\overline{e} = \{ p \} \in \{ s \}\) be a function definition in \(P^1\). If every function defined in \(P\) terminate and the assert/guarantee assertions (I) to (V) defined above hold, the contracts of \(f^1\) holds i.e., \(\forall \overline{x} \in \mathbb{E}_\mathbb{N}_\overline{\overline{x}}, P_{\overline{x}}.\exists u. \overline{f}^1 \overline{x} = p \Downarrow \text{false} \lor \overline{f}^1 \overline{x} = \overline{e} \Downarrow u.

**Solving parametric verification conditions.** To solve the assertions generated by assert/guarantee reasoning and infer values for the holes, we extend the template inference algorithm proposed by us in previous research [51, 52] and implemented in the Leon verification and synthesis system [13, 71] (leondev.epfl.ch). Fig. 7 shows a block diagram of the inference algorithm which we briefly describe in the sequel. Given an assert/guarantee assertion \(\models P e_i \rightarrow e_\overline{g}\) the VC generation phase converts it to a quantifier-free formula (VC) of the form \(\phi(\overline{x}, \overline{a})\), where the variables \(\overline{a}\) corresponds to the numerical holes, such that the assert/guarantee assertion hold if there exists a assignment \(\iota\) for \(\overline{a}\) such that \(\phi(\iota)\) is unsatisfiable. (The VC could be thought of as a \(\exists\overline{\overline{x}}\) formula where the holes are existentially quantified, and the rest including uninterpreted function symbols are universally quantified.)

Converting an assert/guarantee assertion to a many-sorted, first-order theory formula is straightforward. The primitives types such as Int, Bool and the primitive operations are mapped to the corresponding sorts and theory operations. The user-defined datatypes are mapped to algebraic datatypes. Match expressions are converted to disjunctions, and let expressions to equalities. The function calls in the expressions are unfolded upto a certain depth and treated uninterpreted. The pre-and post-conditions of the function calls are assumed (and hence conjoined) at their call sites. Nonlinear operations over \(\overline{x}\) are axiomatized in the VC. The VCs thus generated belong to the theory \(T\) of uninterpreted functions, algebraic datatypes, sets, and nonlinear arithmetic. But, due to the syntactic restrictions on the templates (shown in Fig. 3), the VCs would be linear parametric formulas [51] in which every nonlinear term is of the form \(a \cdot x\) for some \(a\) belonging to \(\overline{a}\) and \(x\) belonging to \(\overline{x}\). Each VC is solved using a counter-example guided algorithm (discussed shortly). If the solving fails, a new VC is generated by further unfolding recursive functions and instantiating nonlinear axioms, and the process is repeated until a solution is found or a timeout is reached.

**Solving linear parametric formulas with sets.** Given a linear parametric VC of the form: \(\phi(\overline{x}, \overline{a})\), the solution for \(\overline{a}\) that will make \(\phi\) unsatisfiable is computed using an iterative but terminating algorithm that progresses in two phases: an existential solving phase (phase I), and a universal solving phase (phase II). Phase I discovers candidate assignments \(\iota\) for the free variables \(\overline{a}\). It initially starts with an arbitrary guess, and subsequently refines it based on the counter-examples produced by Phase II. Phase II checks if the candidate assignment \(\iota\) makes \(\phi\) unsatisfiable. That is, if \(\phi(\iota)\) is unsatisfiable. If not, it chooses a disjunct \(d(\overline{x}, \overline{a})\) satisfiable under \(\iota\) that has only numerical variables by axiomatizing uninterpreted functions and algebraic datatypes in a complete way [52]. This numerical disjunct is then given back to phase I. Phase I generates and solves a quantifier-free nonlinear constraint \(C(\overline{a})\), based on Farkas’ Lemma [20], to obtain the next candidate assignment for \(\overline{a}\) that will make \(d(\overline{x}, \overline{a})\) and other disjuncts previously seen unsatisfiable. Each phase invokes the Z3 [25] and CVC4 [8] SMT solvers in portfolio mode on quantifier-free formulas. This algorithm was shown to be complete for linear parametric formulas belonging to the combined theory of real arithmetic, uninterpreted functions and algebraic datatypes [52]. Below we extend this result to include sets. (Proof detailed in Appendix C.)

**Theorem 4.** Given a linear parametric formula \(\phi(\overline{x}, \overline{a})\) with free variables \(\overline{x}\) and \(\overline{a}\), belonging to a theory \(T\) that is a combination of quantifier-free theories of uninterpreted functions, algebraic datatypes, and sets, and either integer linear arithmetic or real arithmetic, finding a assignment \(\iota\) such that \(\exists \overline{x} (T, \iota)\) is \(T\)-unsatisfiable is decidable.

**Encoding Runtime Invariants and Optimizations.** For improving automation and performance, we explicitly encode certain invariants (described below) ensured by the runtime that are not captured by the model, during VC generation. (a) We encode the referential transparency of the functions in the input program (namely, that the result of the function is independent of the cache state) in the VC in the following way. In principle, this corresponds to the axiom \(\forall x, st_1, st_2. (f^1(x, st_1)), \models (f^1(x, st_2)),\) for every function \(f^1\) in the model. We encode this axiom efficiently by adding the predicate \(\overline{f}^1(\overline{x}, st_1) = U_T(\overline{x})\) for every application of \(f^1\) in the VC, where \(U_T\) is a unique uninterpreted function for \(f^1\). This helps achieve a completely functional reasoning for correctness properties needed for proving resource bounds. (b) We encode the monotonic evolution of the cache by adding the predicate: \(st \subseteq \overline{f}^1(\overline{x}, st),\) for every application of \(f^1\) in the VC. (c) Also, whenever the counter-example guided solving fails, we unfold only calls along the disjuncts \(d(\overline{x}, \overline{a})\) encountered during the solving phase (referred to as targeted unfolding [52]). This prevents unfolding along paths known to be unsatisfiable in the VC thus mitigating the overheads due to defunctionalization.

**5. Evaluation.** We implemented the approach described in the previous sections (leondev.epfl.ch), and used our system to verify resource bounds of many algorithms. In this section, we summarize the results of our experiments. All evaluations presented in this section were performed on a machine with a 4 core, 3.60 GHz, Intel Core i7 processor, 32GB RAM, running Ubuntu operating system.

**Benchmark statistics.** Fig. 8 shows selected benchmarks that were verified by our approach. Each benchmark was implemented and specified in a purely functional subset of Scala extended with our specification constructs. We carefully picked some of the most challenging benchmarks from the literature of lazy data-structures and dynamic programming algorithms. For instance, the benchmark rtq has been mentioned as being outside the reach of prior works (section Limitations of [22]). For each benchmark, the figure shows the total lines of Scala code and the size of the compiled JVM bytecode in columns LOC and BC. The benchmarks comprise a total of 4.5K lines of Scala code and 1.2MB of bytecodes. The column \(T\) shows the number of functions with resource bound templates, and the column \(S\) the number of specification functions. We do not verify resource bounds of specification functions but only verify their termination [78]. The column \(AT\) shows the time taken by our system rounded off to minutes to verify the specifications and infer the constants. As shown by the figure, all benchmarks were verified within a few minutes. The column \(Resource\) shows a sample bound for steps and alloc resource. The constants in the bound were automatically inferred by the tool. We verified a total of 123 bounds each for steps and alloc. Many bounds used recursive functions, and almost 20 bounds had nonlinear operations. (Nonlinear operations like \([\log]\) are expressed as a recursive function that uses integer division: \(\log(x) = \text{if}(x > 2) \log(x/2) + 1\) else (base cases). Their properties like monotonicity are manually proved and instantiated.) A few bounds were disjunctive (like the bound shown in Fig. 1, and cong).
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>BC</th>
<th>T</th>
<th>S</th>
<th>AT</th>
<th>steps ≤</th>
<th>Resource bounds alloc ≤</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy data-structures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lazy Selection Sort (sel)</td>
<td>70</td>
<td>36kb</td>
<td>4</td>
<td>1</td>
<td>1m</td>
<td>15k ⋅ l.size + 8k + 13 2k ⋅ l.size + 2k + 2</td>
<td></td>
</tr>
<tr>
<td>Prime Stream (prims)</td>
<td>95</td>
<td>51kb</td>
<td>7</td>
<td>2</td>
<td>1m</td>
<td>16n^2 + 28 6n – 11</td>
<td></td>
</tr>
<tr>
<td>Fibonacci Stream (fib)</td>
<td>199</td>
<td>59kb</td>
<td>5</td>
<td>5</td>
<td>2m</td>
<td>45n + 4 4n</td>
<td></td>
</tr>
<tr>
<td>Hamming Stream (hams)</td>
<td>223</td>
<td>78kb</td>
<td>8</td>
<td>6</td>
<td>1m</td>
<td>129n + 4 16n</td>
<td></td>
</tr>
<tr>
<td>Stream library (slib)</td>
<td>408</td>
<td>0.1mb</td>
<td>22</td>
<td>5</td>
<td>1m</td>
<td>25l.size + 6 3l.size</td>
<td></td>
</tr>
<tr>
<td>Lazy Mergesort (msort)</td>
<td>290</td>
<td>0.1mb</td>
<td>6</td>
<td>8</td>
<td>1m</td>
<td>36k[log l.size] + 53l.size + 22  6k[log l.size] + 6l.size + 3</td>
<td></td>
</tr>
<tr>
<td>Real time queue (rtq)</td>
<td>207</td>
<td>69kb</td>
<td>5</td>
<td>6</td>
<td>1m</td>
<td>40 7</td>
<td></td>
</tr>
<tr>
<td>Deque (deq) [58, 59]</td>
<td>426</td>
<td>0.1mb</td>
<td>16</td>
<td>7</td>
<td>5m</td>
<td>893 78</td>
<td></td>
</tr>
<tr>
<td>Lazy Numerical Rep.(num)[59]</td>
<td>546</td>
<td>0.1mb</td>
<td>6</td>
<td>25</td>
<td>1m</td>
<td>106 15</td>
<td></td>
</tr>
<tr>
<td>Conqueue (conq) [61, 62]</td>
<td>880</td>
<td>0.2mb</td>
<td>12</td>
<td>33</td>
<td>5m</td>
<td>29</td>
<td>x.sizel - y.sizel</td>
</tr>
</tbody>
</table>

**Dynamic Programming**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>BC</th>
<th>T</th>
<th>S</th>
<th>AT</th>
<th>steps ≤</th>
<th>Resource bounds alloc ≤</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCS (lcs) [21]</td>
<td>121</td>
<td>37kb</td>
<td>4</td>
<td>4</td>
<td>1m</td>
<td>30mn + 30m + 30n + 28 2mn + 2m + 2n + 3</td>
<td></td>
</tr>
<tr>
<td>Levenshtein Distance(levd) [24]</td>
<td>110</td>
<td>37kb</td>
<td>4</td>
<td>4</td>
<td>1m</td>
<td>36mn + 36m + 36n + 34 2mn + 2m + 2 + 3</td>
<td></td>
</tr>
<tr>
<td>Hamming Numbers (hm) [12]</td>
<td>105</td>
<td>44kb</td>
<td>3</td>
<td>3</td>
<td>3m</td>
<td>66n + 65 3n + 4</td>
<td></td>
</tr>
<tr>
<td>Weight Scheduling (ws) [21]</td>
<td>133</td>
<td>44kb</td>
<td>3</td>
<td>5</td>
<td>1m</td>
<td>20jobi + 19 2jobi + 3</td>
<td></td>
</tr>
<tr>
<td>Knapsack (ks) [21]</td>
<td>122</td>
<td>48kb</td>
<td>5</td>
<td>4</td>
<td>1m</td>
<td>17(w ⋅ i.size) + 18w + 17i.size + 18 2w + 3</td>
<td></td>
</tr>
<tr>
<td>Packrat Parsing (pp) [30]</td>
<td>249</td>
<td>73kb</td>
<td>7</td>
<td>5</td>
<td>1m</td>
<td>61n + 58 10n + 10</td>
<td></td>
</tr>
<tr>
<td>Viterbi (vit) [76]</td>
<td>191</td>
<td>63kb</td>
<td>6</td>
<td>7</td>
<td>1m</td>
<td>34k^2t + 34k^2 – 6kt + 14k + g7t + 26 2kt + 2k + 4t + 5</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.** Selected benchmarks comprising of ~5K lines of Scala code and 123 resource bounds each for steps and alloc.

<table>
<thead>
<tr>
<th>B</th>
<th>I</th>
<th>(dynamic/static) * 100 steps</th>
<th>(optimal/static) * 100 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>sel</td>
<td>10k</td>
<td>99</td>
<td>100</td>
</tr>
<tr>
<td>prims</td>
<td>1k</td>
<td>60</td>
<td>82</td>
</tr>
<tr>
<td>fibs</td>
<td>10k</td>
<td>99</td>
<td>100</td>
</tr>
<tr>
<td>hams</td>
<td>10k</td>
<td>86</td>
<td>98</td>
</tr>
<tr>
<td>slib</td>
<td>10k</td>
<td>65</td>
<td>85</td>
</tr>
<tr>
<td>rtq</td>
<td>2^{20}</td>
<td>93</td>
<td>97</td>
</tr>
<tr>
<td>msort</td>
<td>10k</td>
<td>90</td>
<td>96</td>
</tr>
<tr>
<td>deq</td>
<td>2^{20}</td>
<td>48</td>
<td>59</td>
</tr>
<tr>
<td>num</td>
<td>2^{20}</td>
<td>94</td>
<td>96</td>
</tr>
<tr>
<td>conq</td>
<td>2^{20}</td>
<td>72</td>
<td>82</td>
</tr>
<tr>
<td>lcs</td>
<td>1k</td>
<td>88</td>
<td>100</td>
</tr>
<tr>
<td>levd</td>
<td>1k</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>hmcm</td>
<td>10k</td>
<td>79</td>
<td>92</td>
</tr>
<tr>
<td>ws</td>
<td>10k</td>
<td>99</td>
<td>100</td>
</tr>
<tr>
<td>ks</td>
<td>1k</td>
<td>94</td>
<td>100</td>
</tr>
<tr>
<td>pp</td>
<td>10k</td>
<td>77</td>
<td>88</td>
</tr>
<tr>
<td>vit</td>
<td>100</td>
<td>42</td>
<td>86</td>
</tr>
</tbody>
</table>

| Avg | 81 | 88 | 91 | 94 |

**Figure 9.** (a) Mean percentage ratio of runtime resource usage to the static bounds inferred. (b) Comparison of pareto-optimal resource bounds for the runtime data to the static bounds inferred.

However, in our experience, the most challenging bounds to prove were the constant time bounds of scheduling-based lazy data structures viz. rtq, deq, num, and conq due to their complexity.

**Evaluation of accuracy of the inferred bounds.** We instrumented the benchmarks for tracking steps and alloc resources as defined by the operational semantics, and executed them on concrete inputs that were likely to expose the worst case behavior. We varied the sizes of the inputs in fixed intervals up to 10k for most benchmarks. However, for those benchmarks with nonlinear behavior we used smaller inputs that scaled within a cutoff time of 5 min, as tabulated in the column I of Fig. 9. For scheduling based data structures (discussed shortly) we varied the input in powers of two until 2^{20}, which results in their worst-case behavior. For every top-level (externally accessible) function in a benchmark, we computed the mean ratio between the runtime resource usage and the static resource usage predicted by our tool using the following formula: Mean \( \left( \frac{\text{resource consumed by the } i^{th} \text{ input}}{\text{static estimate for } i^{th} \text{ input}} \right) \times 100 \). The column dynamic/static * 100 of Fig. 9 shows this metric for each benchmark when averaged over all top-level functions in the benchmark. As shown in the figure, when averaged across all benchmarks the runtime resource usage was 81% of what was inferred statically for steps, and is 88% for alloc. In all cases, the inferred bounds were sound upper bounds for the runtime resource usage. We now discuss the reasons for some of the inaccuracy in the inferred bounds.

In our system, there are two factors that influence the overall accuracy of the bound: (a) the constants inferred by tool, and (b) the resource templates provided by the user. For instance, in the prims benchmark shown in Fig. 1 the function isPrimeNum(n) has a worst-case steps count of 11i – 7, which will be reached only if i is prime. (It varies between \( O(\sqrt{i}) \) and \( O(i) \) otherwise.) Hence, for the function primesUntil(n), which transitively invokes isPrimeNum on all numbers until \( n \), no solution for the template: \( ? + n^2 + ? \) can accurately match its worst-case, runtime steps count. Another example is the \( O(k ⋅ [log(l.size)]) \) resource bound of msort benchmark. In any actual run, as \( k \) increases the size of the stream that is accessed (which is initially \( l \)) decreases. Hence, \( [log(l.size)] \) term decreases in steps.

To provide more insights into the contribution of each of these factors to the inaccuracy, we performed the following experiment. For each function, we reduced each constant in its resource bound, keeping the other constants fixed, until the bound violated the resources usage of at least one dynamic run. We call such a bound a pareto optimal bound with respect to the dynamic runs. Note that if there are \( n \) constants in the resource bound of a function, there would be \( n \) pareto optimal bounds for the function. We measured
the mean ratio between the resource usage predicted by the pareto optimal bound and that predicted by the bound inferred by the tool. The column \textit{optimal/static * 100} of Fig. 9 shows this metric for each benchmark when averaged over all pareto optimal bounds of all top-level functions in the benchmark. A high percentage for this metric is an indication that any inaccuracy is due to imprecise templates, whereas a low percentage indicates a possible incompleteness in the resource inference algorithm, which is often due to non-linearity or absence of sufficiently strong invariants. As shown in Fig. 9, the constants inferred by the tool were 91% accurate for steps and 94% accurate for alloc, when compared to the pareto optimal values that fits the runtime data. Furthermore, the imprecision due to templates is a primary contributor for inaccuracy, especially in benchmarks where the accuracy is lower than 80% (such \textit{Viterbi} and \textit{prims}). In the sequel, we discuss the benchmarks and the results of their evaluation in more detail.

**Cyclic streams.** The benchmarks \textit{fib} and \textit{hams} implement infinite fibonacci and hamming sequences as cyclic streams using lazy \texttt{zipWith} and \texttt{merge} functions. Their implementations were based on the related work of Vasconcelos et al. [74]. In comparison to their work in which the alloc bounds computed for \textit{hams} were 64% accurate for inputs smaller than 10, our system was able to infer bounds that were 83% accurate for inputs up to 10K.

**Scheduling-based lazy data structures.** The benchmarks \textit{rtq}, \textit{deq}, \textit{num}, and \textit{conq} use lazy evaluation to implement worst-case constant time, persistent queues and deques using a strategy called scheduling. These are one of the most efficient persistent data structures. For instance, the \textit{rtq} [58] benchmark takes a few nanoseconds to persistently enqueue an element into a queue of size $2^{30}$. The \textit{conq} data structure [62] is used to implement data-parallel operations provided by the standard Scala library. Though the data structures differ significantly in their internal representation, invariants, resource usage and the operations they support, fundamentally they consists of streams called \textit{spines} that track content, and a list of references to closures nested deep within the spines: \textit{schedules}. The schedules help materialize the data structure lazily as they are used by a client. We are not aware of any prior approach that proves the resource bounds of these benchmarks. We also discovered and fixed a missing corner case of the \textit{rotateDrop} function shown in Fig. 8.4 of [59], which was unaveraled by the system.

As the results in Fig. 9 show, the inferred bounds were at least 83% accurate for \textit{rtq} and \textit{num} benchmarks, but have low accuracy for \textit{deq} and \textit{conq} benchmarks. On further analysis of \textit{deq} we found that the bounds inferred by our system for the inner functions of \textit{deq} were, in fact, 90% accurate in estimating the worst-case usage for the dynamic runs. But the worst-case manifested only occasionally (about once in four calls) when invoked from the top-level functions. The low accuracy seems to result from the lack of sufficient invariants for the top-level functions that prohibit the calls to inner functions from consistently exhibiting worst-case behavior.

**Other lazy benchmarks.** The benchmark \textit{slib} is a collection of operations over streams such as \texttt{map}, \texttt{scan}, \texttt{cycle} etc. The operations were chosen from the Haskell stream library [72]. We excluded functions such as \texttt{filter} that can potentially diverge on infinite streams. The bounds presented are for a specific client of the library. The benchmarks \textit{msort} and \textit{set} implement lazy sorted streams that allows accessing the $k^{30}$ minimum without performing the entire sorting. In particular, \textit{msort} uses a lazy bottom-up merge sort [4] wherein a logical tree of closures of the merge function is created and forced on demand.

**Dynamic programming algorithms.** We verified the resource bounds of dynamic programming algorithms [21, 24] shown in Fig. 8 by expressing them as memoized recursive functions. In particular, the benchmark \textit{pp} is a memoized implementation of a packrat parser presented by Ford [30] for the \textit{parsing expression grammar} used in that work. As shown in Fig. 9, the inferred bounds for \textit{pp} are on average 90% accurate for the dynamic programming algorithms except \textit{pp} and \textit{vit}, and is 100% accurate in the case of \textit{alloc} for all benchmarks except \textit{pp}. In the case of \textit{vit}, the main reason for inaccuracy stems from the \textit{cubic} template (shown in Fig. 8), as highlighted by the results of comparison with the pareto optimal bound shown in Fig. 9. In the case of \textit{pp}, the evaluations were performed on random strings as were unable to precisely deduce the worst-case input. Nevertheless, the bounds inferred were 100% accurate for the inner functions: \textit{pAdd}, \textit{pMul}, and \textit{pPrim}.

6. Related Work

**Static Resource Analysis for Lazy Evaluation.** Danielsson [22] present a lightweight type-based analysis for verifying time complexity of lazy functional programs and applied it to \textit{implicit queues}. As noted in the paper, the approach is limited in handling aliasing of lazy references, which is crucial for our benchmarks. Vasconcelos et al. [69, 74] present a typed-based analysis for inferring bounds on memory allocations of Haskell programs. They evaluated their system on cylic hamming and fibonacci stream, which were included in our benchmarks, and discussed in section 5. In contrast to the above works, our approach is targeted at verifying user-specified bounds, and has been evaluated on more complex, real-world programs for relatively large input sizes.

**Static Resource Bounds Analysis.** Automatic static inference of resource bounds of programs has been an actively researched area. Some of the recent works include [1, 2, 7, 17, 29, 35, 38, 47, 53, 70, 84]. Being fully automated, these approaches target simpler programs and bounds that depend on less complex invariants compared to our approach. Another related line of work include semi-automatic formal frameworks amenable to deriving machine-checked proofs of resource bounds [9, 23, 66, 67]. In particular, Sands [66, 67] present a theoretical framework for reasoning about lazy evaluation. We are not aware of any machine-checked proofs for the resource bounds of the lazy data structures considered in our study. Recent works on resource analysis have started incorporating user specifications. Alonso et al. [3] presented an approach where resource bounds are specified by users as templates. Carbonneaux et al. [18] presented a system to verify stack space bounds of C programs written for embedded systems using a quantitative Hoare logic. Previously, we proposed an approach [52] for inferring resource bounds using user-defined templates and specifications for first-order, non-lazy functional programs with algebraic datatypes.

**Coinductive datatypes.** Leino and Moskal [49] use coinduction to verify programs with possibly infinite lazy data structures. They do not consider resource properties of such programs. Blanchette et al.[14, 15] present a formal framework for soundly mixing recursion and corecursion in the context of interactive theorem provers.

**Imperative and Higher-order Verification.** Verification Systems such as [16, 26, 28, 36, 48, 60, 65, 83] and interactive theorem provers [10, 19, 56] have been used to verify complex, imperative programs. Automation in our system appears above the one in interactive provers, and could be further improved using quantifier instantiation, induction, and static analysis [11, 33, 63]. While most approaches for imperative programs target a homogeneous, mutable heap, in this work we consider an almost immutable heap except for the cache, and use a set representation to handle mutations to the cache efficiently. We believe that similar separation of heap into mutable and immutable parts can benefit other forms of restricted mutation like \textit{write-once} fields [6].
Works such as [28, 42–44, 54, 55, 73, 75, 77–79, 81, 82] target correctness verification of higher-order, functional programs. Many of these systems allow users to write contracts on function-valued parameters, or refinement predicates on function types [28, 75]. We are not aware of any contract-based verifiers for higher-order programs that allow specifying resource properties, as in our approach. Our approach allows named functions, with contracts and resource templates, to be used inside lambdas. However, it disallows contracts on function-valued parameters and instead provides intensional-equality-based constructs to specify their properties. Though this makes the contracts very specific to the implementation, it has the advantage of reducing specification burden for closed or encapsulated programs. Supporting contracts on function-valued parameters that can refer to resource bounds would be an interesting future direction to explore.

References
...
components $C$ of the environment, namely $(C, \mathcal{H}, \sigma, F)$, using $C_i$ (or $C^i$), respectively.

**Reachability relation.** Fig 10 shows the complete, formal definition of the reachability relation $\leadsto$.

**Terminating Evaluations.** An evaluation is non-terminating iff there exists an infinite sequence: $(\Gamma, e) \rightsquigarrow (\Gamma_1, e_1) \rightsquigarrow \ldots$. An evaluation is terminating iff there are no infinite sequences starting from $(\Gamma, e)$. For a terminating evaluation, there is a natural number $n$ such that the length of every chain is upper bounded by $n$. That is, $\exists n \in \mathbb{N}. \neg(\exists k > n, e, \Gamma', (\Gamma, e) \rightsquigarrow^k (\Gamma', e))$. This is because the number of distinct chains is finite, as for every $(\Gamma, e)$ there exists at most three different successors (see Fig. 4).

**Structural Induction over Big-step Semantic Rules.** We now establish an induction strategy to prove properties of the operating semantics. To prove that a property $\rho(\Gamma, e, v, \Gamma', p)$ holds for an evaluation $\Gamma \vdash e \Downarrow_p v, \Gamma'$ we perform induction over the depth of the evaluation. That is, we inductively establish that $\forall n \in \mathbb{N}. \neg(\exists k > n, e, \Gamma'', (\Gamma, e) \rightsquigarrow^k (\Gamma'', e)) \Rightarrow \rho(\Gamma, e, v, \Gamma', p)$. This boils down to the following strategy. For every semantics rule $\text{RULE}$, we assume that the property holds for the big-step reductions in the antecedent and establish that it holds in the consequent.

**Structural induction over $\approx$ and $\leadsto$.** Recall that the relations $\approx$ and $\leadsto$ are defined recursively. As usual, we define their semantics using least fixed points. Let $R \subseteq A$ be a relation defined by a recursive equation $R = h(R)$ where $h$ is some function that uses the relation $R$. The relations $\approx$, $\leadsto$ and $\sim$ can be viewed as being in this form. The solution for the above equation is the least fixed point of $h$. Since relations are sets of pairs, there exists a natural partial order on the relations namely $\subseteq$. The ordered set $(2^A, \subseteq)$ is a complete lattice, which implies that there exists a unique least fixed point for every Scott-continuous function (by Knaster-Tarski theorem). Also, the least fixed point can be computed using Kleene iteration. Let $R^0 = \emptyset$ and $R^i = h(R^{i-1})$. The least fixed point of $h$, and hence the solution to $R$, is $\bigcup_{i=0} R^i$. This definition of $R$ naturally lends itself to an inductive reasoning: to prove a property on $R$, we establish that (a) the property holds for $\emptyset$, and (b) that if it holds for $R^{i-1}$ it holds for $R^i$. In the context of $\approx$ and $\leadsto$, assuming that the property holds for $R^{i-1}$ means that the relation can be assumed to hold in the right hand sides of the definition of the relation. We refer to this as structural induction over $R$.

**Determination of the semantics.** The semantics shown in Fig. 4 has a source of non-determinism namely the function $\text{fresh}(\mathcal{H})$ that arbitrarily chooses a fresh address not belonging $\text{dom}(\mathcal{H})$. We make this function deterministic by fixing a well-ordering on the elements of $\text{Adr}$ and requiring that $\text{fresh}(\mathcal{H})$ always returns the smallest address not bound in the heap $\mathcal{H}$. That is, $\text{fresh}(\mathcal{H}) = \min(\text{Adr} \setminus \text{dom}(\mathcal{H}))$.

**Acyclic Heaps.** We say a heap $\mathcal{H} \in \text{Heap}$ does not have any cycles iff there exists a well-founded, irreflexive (i.e strict) partial order $< \text{on } \text{dom}(\mathcal{H})$ such that for every $(a, v) \in \text{Heap}$, either $v \in Z \cup \text{Bool}$, or $v = \text{cons} \ u$ and $\forall i \in [1, |u|], u_i \in \text{Adr} \Rightarrow u_i < a$, or $v = (e, \sigma)$ and $\forall a' \in \text{range}(\sigma) \cap \text{Adr} \Rightarrow a' < a$. The relation $<$ is well-founded.

**Lemma 5.** Let $\mathcal{H}$ be an acyclic heap. The structural equivalence relation $\approx_\mathcal{H}$ is reflexive, transitive and symmetric. That is,

- $(a) x \approx_\mathcal{H} y \wedge y \approx_\mathcal{H} z \Rightarrow x \approx_\mathcal{H} z$
- $(b) x \approx_\mathcal{H} y \Rightarrow y \approx_\mathcal{H} x$
- $(c) x \approx_\mathcal{H} x$

**Proof.** The transitivity and symmetry properties follow from a simple structural induction (due to the transitivity and symmetry of equality over integers and booleans). The reflexivity property trivially holds for integers and booleans. To prove the property for addresses, we induct over the well-founded relation $\leadsto$. The base case consists of addresses in the heap that are mapped to values that do not use other addresses. The reflexivity property clearly holds in this case. The inductive case consists of addresses that are mapped to values, namely data or closure or function values, that may use addresses satisfying the reflexivity property. Here again it is easy to see that the claim holds, since for two closure/data/function values to be structurally equal they have to invoke the same function or use the same constructor.

In the rest of the paper, we consider only acyclic heaps even if not explicitly mentioned.

**Containment ordering on partial functions.** Given two partial functions $(h_1, h_2) \subseteq A \rightarrow B$, we say $h_1 \sqsubseteq h_2$ iff $h_2$ has more entries than $h_1$. That is, $a \in \text{dom}(h_1)$ implies $h_1(a) = h_2(a)$. The ordering $\sqsubseteq$ satisfies reflexivity, transitivity and anti-symmetry, and hence is a partial order. We extend this partial order to the environments as defined below:

$$(C_1, \mathcal{H}_1, \sigma_1, F) \sqsubseteq (C_2, \mathcal{H}_2, \sigma_2, F) \iff C_1 \subseteq C_2 \land \mathcal{H}_1 \sqsubseteq \mathcal{H}_2 \land \sigma_1 \sqsubseteq \sigma_2$$

**Structural Simulation Relation.** Similar to structural equivalence, we define a structural simulation $\sim_\mathcal{H}_1, \mathcal{H}_2, \sigma_1, \sigma_2$, with respect to two heaps, between the elements of the semantic domains as follows: (The subscripts $\mathcal{H}_1, \mathcal{H}_2$ are omitted below for clarity.)

$$\forall a \in Z \cup \text{Bool}. \ a \approx a$$
$$\forall \{a, b\} \subseteq \text{Adr}. \ a \approx b \iff \mathcal{H}_1(a) = \mathcal{H}_2(b)$$
$$\forall f \in \text{Fids}, \ (a, b) \subseteq \text{Val}. \ (f \ a) \approx (f \ b) \iff a \approx b$$
$$\forall c \in \text{Cids}, \ (a, b) \subseteq \text{Val}^n. \ (c \ a) \approx (c \ b) \iff \forall i \in [1, n], a_i \approx b_i$$
$$\forall \{e_1, e_2\} \subseteq \text{Lam}. \forall (\sigma_1, \sigma_2) \subseteq \text{Store} \cdot \forall (e_1, e_2) \approx (e_1, e_2) \iff \text{target}(e_1) = \text{target}(e_2) \land \sigma_1(FV(e_1)) \approx \sigma_2(FV(e_2))$$

Notice that the only change compared to $\approx_\mathcal{H}$ is the rule for addresses which now uses different heaps. The following are some properties preserved by structural simulation. (We omit the proof of the following properties as they are straightforward to derive from the definitions.)

- if $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$, $\sim_\mathcal{H}_1, \mathcal{H}_2$ reduces to $\sim_\mathcal{H}_1, \mathcal{H}_2$
- (Symmetry) $x \approx_\mathcal{H}_1, \mathcal{H}_2 y$ implies $y \approx_\mathcal{H}_1, \mathcal{H}_2 x$
- (Transitivity) $x \approx_\mathcal{H}_1, \mathcal{H}_2 y$ and $y \approx_\mathcal{H}_1, \mathcal{H}_2 z$ implies $x \approx_\mathcal{H}_1, \mathcal{H}_2 z$
- If $u \approx_\mathcal{H}_1, \mathcal{H}_2 v$ then $(u \approx_\mathcal{H}_1, \mathcal{H}_2 v' \equiv v \approx_\mathcal{H}_1, \mathcal{H}_2 v')$ and $(u' \approx_\mathcal{H}_1, \mathcal{H}_2 v \equiv v' \approx_\mathcal{H}_1, \mathcal{H}_2 v')$

**Structural Abstraction Relation.** Using the structural simulation relation, we now define a structural abstraction relation $\approx_\mathcal{H}$ between
two environments.
\[(C_1, H_1, \sigma_1, F) \subseteq (C_2, H_2, \sigma_2, F) \triangleq C_1 \subseteq C_2 \land \sigma_1 \subseteq \sigma_2, \text{ where}, \]
\[\sigma_1 \subseteq \sigma_2 \iff \forall x \in \text{dom}(\sigma_1), \sigma_1(x) \approx_{H_1, H_2} \sigma_2(x), \text{ and} \]
\[C_1 \subseteq C_2 \iff \forall k \in \text{dom}(C_1), \exists k' \in \text{dom}(C_2).k \approx_{H_1, H_2} k' \land C_1(k) \approx_{H_1, H_2} C_2(k').\]

Note that \(\subseteq\) is a stronger relation than \(\approx\).

Structural Equivalence of Environments. We say two environments are structurally equivalent iff \(\Gamma_1 \leq \Gamma_2\) and vice-versa. That is,
\[\Gamma_1 \approx \Gamma_2 \iff (\Gamma_1 \subseteq \Gamma_2 \land \Gamma_2 \subseteq \Gamma_1).\]

Congruence and substitutability of \(\approx\). In any given environment, substituting a value of a variable by a structurally equivalent value preserves the result as well as the resource usage of the evaluation of any expression \(e\).

\[\text{Lemma 6. For all } \{\Gamma_1, \Gamma_2\} \subseteq \text{Env such that } \Gamma_1 \approx \Gamma_2, \text{ for all expression } e,\]
\[\Gamma_1 \vdash e \Downarrow \Gamma_2' \Rightarrow \exists v, \Gamma_2' \vdash e \Downarrow \Gamma_2' \land \Gamma_1 \approx \Gamma_2' \land u \approx v \land p = q\]

\[\text{Proof. The claim directly follows by structural induction over the operation semantic rules shown in Fig. 4.}\]

Immutable Heap Properties. Below we present two lemmas that establish the immutable nature of the heap using the operational semantic rules.

\[\text{Lemma 7. Let } \Gamma : (C, H, \sigma, F), \Gamma_1 : (C_1, H_1, \sigma_1, F) \text{ and } e \text{ be an expression. If } (\Gamma, e) \rightsquigarrow (\Gamma_1, e_1) \text{ or } \Gamma \vdash e \Downarrow \Gamma_1, \text{ then } H \subseteq H_1 \land C \subseteq C_1. \text{ That is, the evaluation can only add more entries to the heap and cache, and cannot update existing entries.}\]

\[\text{Proof. This directly follows from the semantic rules shown in Fig. 4. Every time an address is added to the heap, it is chosen to be a fresh address that is not already bound in the heap. A function value is added to a cache iff a structurally equivalent value does not belong to the domain of the cache. As proved in Lemma 5, the structurally equivalence relation is reflexive. Thus, a function value is added to the cache only if it does not already have a binding in the cache.}\]

\[\text{Lemma 8. Let } \Gamma : (C, H, \sigma, F) \in \text{Env and } e \in \text{expression. Let } \Gamma_1 = (C_1, H_1, \sigma, F) \text{ where } H_1 \subseteq H. \text{ If } \Gamma \vdash e \Downarrow \Gamma_1, \text{ then } \Gamma_1 \vdash e \Downarrow \Gamma_2 : (C_2, H_2, \sigma_2) \text{ and } \sigma_2 \approx_{H_1} \sigma. \text{ That is, adding more entries to the heap preserves the result of the evaluation with respect to the structural equivalence relation } \approx.\]

\[\text{Proof. This follows from a structural induction over the big-step semantic rules shown in Fig. 4 and Lemma 7. The theorem follows from two facts (a) all of the semantics rules access the heap } H \text{ via the store } \sigma \text{ e.g. } H(\sigma(x)), \text{ and (b) there are no rules that can change the value of an address bounded in the heap.}\]

Domain Invariants. The environments that arise during an evaluation of a program \(P\) satisfies several invariants that are ensured by the runtime. Below we characterize the invariants. Let \(P\) be a type-correct program. Let \(\text{Env}_P \subseteq \text{Env}\) be the set of environments such that for every \(\Gamma : (C, H, \sigma, F) \in \text{Env}_P\) the following properties hold.

(a) \((\text{dom}(\sigma) \cap \text{Addr}) \subseteq \text{dom}(H)\)
(b) For all variable \(x \in P,\)
\[x \in \text{dom}(\sigma) \text{ implies that } \sigma(x) \text{ inhabits type}_P(x).\]
(c) Every function definition in \(F\) has a unique function identifier.
(d) \(F\) contains every function definition in \(P\).
(e) For all \(\lambda x.\ f \ (x, y) \in \text{range}(H), f\) is defined in \(F\).
\[f \Rightarrow \exists k, k' \subseteq \text{dom}(C), k \neq k' \land k \approx_{H} k', \text{ and } C \supseteq C_1.\]
\[\text{The last invariant states that every key that is stored in the cache evaluates to the value that is cached in some smaller environment. The semantics rules shown in Fig. 4 preserve the domain invariants. That is, if } \Gamma \in \text{Env}_P \text{ and } (e, \Gamma) \rightsquigarrow (e', \Gamma') \text{ then } \Gamma' \in \text{Env}_P. \text{ Moreover, if } \Gamma \text{ is defined on all the free variables of } e, \text{ denoted } \text{fv}(e), \text{ then } \Gamma' \text{ will be defined on all the free variables of } e'.\]

\[\text{Figure 10. Definition of the reachability relation.}\]
Lemma 9. Let \( e \) and \( e' \) be expressions in a program \( P \). Let \( \Gamma \in Envp \) and \( \Gamma' \in Env. \) If \( \Gamma \vdash e \Downarrow v, \Gamma' \) then \( \Gamma' \in Envp. \)

Proof. The proof follows by structural induction over the operational semantics. The invariant (g) follows from the following fact that when a key is added to a cache by the MEMOCALLMiss rule, the property holds by definition for the input cache and heap. Say the input cache and heap of MEMOCALLMiss are \( C_1 \) and \( H_1 \) and the output heap and cache are \( C_1' \) and \( H_1' \). In this case, the invariant (g) holds for the following assignment: \( C = C'' = C_1' \) (see Fig. 4). \( C = C_1 \). Similarly, \( H = H'' = H_1' \) and \( H'' = H_1 \). Every subsequent evaluation can only increase the size of the cache and heap (by Lemma 7), and hence the invariant (g) is preserved once it holds.

Corollary 10. Let \( e \) and \( e' \) be expressions in a program \( P \). Let \( \Gamma \in Envp \) and \( \Gamma' \in Env. \) (a) If \( \langle e, \Gamma \rangle \rightarrow^* \langle e', \Gamma' \rangle \) then (a) \( \Gamma' \in Envp \) and (b) \( \text{fe}(e) \subseteq \text{dom}(\Gamma) \Rightarrow \text{fe}(e') \subseteq \text{dom}(\Gamma'). \)

Proof. Note that \( \rightarrow \) is defined using the big-step semantics rules as shown by Fig.10. The proof follows from a straightforward induction over \( k \) where \( \langle e, \Gamma \rangle \rightarrow^k \langle e', \Gamma' \rangle \) and the above lemma (Lemma 9).

Valid Environments. Recall that in section 2 we define the valid environments \( Envs, \rho \) that reach an expression \( e \) in a program \( P \) as: \( \{ \Gamma \mid \exists \Gamma', (\Gamma \triangleright p, \text{entry}) \rightarrow^* (\Gamma, e) \} \)

We also impose a constraint that the evaluation \( (\Gamma \triangleright p, \text{entry}) \) is terminating, unless the functions in program \( P \) are non-terminating. This is because for every \( \Gamma \in \text{hp}, \rho \) there always exist a program \( P'' \) such that \( (\Gamma \triangleright p', \text{entry}) \rightarrow^* (\Gamma, e) \), but the evaluation terminates (or halts) immediately after (and if) it returns from the function \( f \).

By Lemma 9, all valid environments satisfy the above domain invariants, since they are satisfied by \( \Gamma \triangleright p. \) Moreover, \( \Gamma : (C, H, \sigma, F) \in Envs, \rho \) implies that \( \text{fe}(e) \subseteq \text{dom}(\sigma) \).

Weak Referential Transparency and Weak Cache Correctness. In our language, we allow expressions to query the state of the cache using the construct cached. While this is indispensable for specifying properties about the state of the cache, this also makes the expressions of the language not referentially transparent. However, as captured by the syntax shown in Fig. 3, these constructs are restricted to the specifications (i.e. contracts). The source expressions \( E_{src} \) of our language exhibit a weak form of referentially transparency with respect to the changes to the cache. The weak referential property guarantees that if a source expression evaluates to a value \( u \) at a point in the evaluation, then if it evaluates to a value \( v \) at a later point in the evaluation then \( u \) and \( v \) are equivalent. Formally, we say that a source expression evaluated under two environments related by \( \leq \) should produce structurally similar values, provided the evaluations produce any value at all. This is stated and proved below. (Note that the heaps and caches that may arise during an execution are related by \( \leq \) by Lemma 7, and hence are also related by the weaker relation \( \leq ')\).

Lemma 11. Let \( \Gamma : (C_1, H_1, \sigma_1, F) \) in \( Envp. \) For all expression \( e_s \in (E_{src} \cup FVal), \) if \( \Gamma_1 \vdash e_s \Downarrow u, \Gamma_1' \) then \( \forall \Gamma_2 : (C_2, H_2, \sigma_2, F) \in Envp \) s.t. \( \Gamma_2 \approx \Gamma_2 \), \( \Gamma_2 \vdash e_s \Downarrow v, \Gamma_2 \Rightarrow u \approx \text{val} \)

Proof. We prove the lemma using structural induction on the evaluation \( \Gamma_1 \vdash e_s \Downarrow u, \Gamma_1' \). Say the evaluation \( \Gamma_1 \vdash e_s \Downarrow u, \Gamma_1' \) uses one of the base cases, namely the rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MEMOCALLHit. Note that the rule CACHED is not a part of the source expressions (see Fig. 3) and thus can be excluded from the base cases. Firstly, by Lemma 7, we know that \( \Gamma_1 \subseteq \Gamma_1' \). (The store components of \( \Gamma_1 \) and \( \Gamma_1' \) are identical.) Therefore, \( \Gamma_1 \subseteq \Gamma_2 \). Every case other than MEMOCALLHit uses only the heap and the store (and not the cache). Since \( \Gamma_1 \subseteq \Gamma_2 \), the free variables in the expressions are bound to structurally similar values in \( \Gamma_1 \) and \( \Gamma_2 \). It is easy to see that in each of the cases the resulting values are also structurally similar. Now say the evaluation \( \Gamma_1 \vdash e_s \Downarrow u, \Gamma_1' \) uses MEMOCALLHit. Therefore, \( e \) is of the form \( (f \ x) \) and \( \sigma_1(x) \approx k \) where \( k \) is a key in the cache \( C_1 \). Since \( \Gamma_1 \subseteq \Gamma_2 \), \( \sigma_1(x) \approx \sigma_2(x) \) and there exists a \( k' \in \text{dom}(C_2) \) such that \( k \approx k' \). By the properties of \( \approx \), \( \sigma_2(x) \approx k' \). Hence, the evaluation of \( \Gamma_2 \vdash e_s \Downarrow u \) must also use the rule MEMOCALLHit. In both cases, the value of the expression is looked up from the corresponding caches, and hence are structurally similar (by the definition of \( \leq ')\).

Now say the evaluation \( \Gamma_1 \vdash e_s \Downarrow u, \Gamma_1' \) uses one of the inductive cases: LET, MATCH, CONCRETECALL, NONMEMOCALLHit, MEMOCALLMiss and CONTRACT. If \( \Gamma_1 \vdash e_s \Downarrow u, \Gamma_1' \) uses any rule \( R \) other than MEMOCALLHit, then \( \Gamma_2 \vdash e_s \Downarrow u, \Gamma_2' \) will also use the same rule, which is determined by the syntax of the expression (see Fig. 4). Say now \( (\Gamma_1, e_s) \approx (\Gamma_3, e') \) and \( (\Gamma_1, e_s) \approx (\Gamma_4, e') \). Firstly, the environment \( \Gamma_3 \) and \( \Gamma_4 \) are obtained from a prior big-step evaluation given by an antecedent of \( R \), after possible updatings to the store component. Let \( \Gamma_3 \vdash e' \Downarrow v, \Gamma_4 \). By Lemma 7 and the given facts, \( C_1 \subseteq C_3 \subseteq C_4 \subseteq C_1' \subseteq C_2 \subseteq C_4. \) Consider the store component of \( \Gamma_3 \), which is identical to \( \Gamma_4 \), and \( \Gamma_4 \) namely \( \sigma_3 \) and \( \sigma_4 \). Any new mappings added to the store components depend on the prior big-step reductions in the antecedent of \( R \), which satisfies the induction hypothesis. Thus, the new entries added are structurally similar. Hence, \( \sigma_4 = \sigma_3 \leq \sigma_1 \). Therefore, \( \Gamma_3 \approx \Gamma_4 \) by induction hypothesis, \( e' \) evaluates to structurally similar values in \( \Gamma_3 \) and \( \Gamma_4 \). Since this holds for every antecedent of \( R \) and in all inductive cases the result of the rule is obtained directly from the result of an antecedent evaluation involving a source expression or function value (see Fig. 4, especially rule CONTRACT), both evaluations \( \Gamma_1 \vdash e_s \Downarrow v, \Gamma_1' \) and \( \Gamma_2 \vdash e_s \Downarrow v, \Gamma_2' \) produce structurally similar results. That is, \( u \approx v \).

Now say the evaluation \( \Gamma_1 \vdash e_s \Downarrow u, \Gamma_1' \) uses the MEMOCALLMiss rule. In this case, since \( C_2 \) has more entries than \( C_1 \), \( \Gamma_2 \vdash e_s \Downarrow v, \Gamma_2' \) will use the rule MEMOCALLHit, as explained below. In this case, we know that \( e_s = (f \ y) \) and \( ((f \sigma_1(y)), u) \in C_1' \), (Recall that the rule MEMOCALLMiss records the function value and the result of the evaluation in the cache.) Since \( C_1' \subseteq C_2 \), there exists an entry \( (k, v) \in C_2 \) such that \( (f \sigma_1(y)) \approx k \) and \( u \approx v \). Since \( C_1 \subseteq C_2 \), \( \sigma_1(y) \approx \sigma_2(y) \). Thus \( (f \sigma_1(y)) \approx (f \sigma_2(y)) \). By the property of \( \approx \), \( f \sigma_2(y) \approx k \). Under the definition of MEMOCALLHit, the result of the evaluation under \( \Gamma_2 \) is \( v \) since \( \nexists H_2 \subseteq \nexists H_2', u \approx v \) which implies the claim.

Weak Cache Correctness. Consider the following property on an environment \( \Gamma : (C, H, \sigma, F) \) that states that every key in the cache
Lemma 13. Let $\Gamma_1 = (C_1, H_1, \sigma_1, F)$ in $\text{Env}$. WeakCacheCorr($\Gamma_1$) $\Rightarrow$ $\exists k \in \text{dom}(C_1)$, $(\Gamma_1 \vdash k \Downarrow v, (C', H', \sigma', F')) \Rightarrow v \approx C(k)$

We now show that WeakCacheCorr is an invariant with respect to the semantic reduction.

Lemma 12. For all expression e, for all $\Gamma_1 : (C_1, H_1, \sigma_1, F) \in \text{Env}$, WeakCacheCorr($\Gamma_1$) $\Rightarrow$ $\exists u, \Gamma'_1 \Rightarrow$ WeakCacheCorr($\Gamma'_1$)

Proof: We prove the lemma using structural induction over the environment $\Gamma_1$. First consider the base cases: rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MEMOCALLHit and CACHED. In each of these cases, either the input and output environments are identical, or the output environment has one new binding in the heap. By Lemma 8, the claim holds in all the base cases.

Say the evaluation $\Gamma_1 \vdash e \Downarrow u, \Gamma'_1$. First consider the base cases: rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MEMOCALLHit and CACHED. In each of these cases, either the input and output environments are identical, or the output environment has one new binding in the heap. By Lemma 8, the claim holds in all the base cases.

Say the evaluation $\Gamma_1 \vdash e \Downarrow u, \Gamma'_1 \Rightarrow$ WeakCacheCorr($\Gamma'_1$) using structural induction over the environment $\Gamma_1$. First consider the base cases: rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MEMOCALLHit and CACHED. In each of these cases, either the input and output environments are identical, or the output environment has one new binding in the heap. By Lemma 8, the claim holds in all the base cases.

Download the full text for better readability and understanding.
We now show that CacheCorr is an invariant with respect to the semantic reduction given a valid program P.

**Lemma 15.** Let \( F \) be a set of function definitions such that for all \( \forall f \in F \), \( F \) is in \( \text{Env} \), and \( p \) and \( s \) are cache monotonic properties. For all expression \( e \), for all \( \Gamma : (C, \mathcal{H}, \sigma, F) \in \text{Env} \),

\[
\text{CacheCorr}(\Gamma) \land \Gamma \vdash e \Downarrow u, \Gamma \Rightarrow \text{CacheCorr}(\Gamma)
\]

**Proof.** The proof of this lemma is very similar to the proof of Lemma 12, except for the rule MECOMMiss. Analogous to the proof of Lemma 12, we now use the Lemma 14 to establish that the CacheCorr property holds for the output environment for the rule MECOMMiss.

Let \( k \in FVal \cap dom(C_1) \) be a key in the cache \( C_1 \). By the domain invariants, there exists a \( H_0 \subseteq H_1 \) and \( C_0 \subseteq C_1 \) such that \( \Gamma \vdash k \Downarrow u_0, \Gamma \Rightarrow \), where \( u_0 = C_2(k) \), \( \Gamma_0 = \langle C_0, H_0, \{j\} \rangle \), \( \Gamma_0 = \langle C_0, H_0, \{j\} \rangle \) and \( C_0 \subseteq C_1 \). Since \( C_0 \subseteq C_1 \) and \( \{j\} \subseteq \sigma_1, \Gamma_0 \subseteq \Gamma_1 \). Therefore, by Lemma 14,

\[
\Gamma_1 \vdash k \Downarrow u, \Gamma_1 \vdash u \Downarrow w, \Gamma_1 \vdash w \Downarrow w.
\]

Therefore, CacheCorr(\( \Gamma_1 \)).

A.2 The Intermediate Semantics I

We now present a new intermediate semantics denoted \( I \) which is used as an intermediate step in establishing the correctness of the model programs. The semantics \( I \) is defined by the same set of rules as the operational semantics shown in shown in Fig. 4, except for the rule MEMOCALHit which is replaced by the rule shown below. We denote the reduction with respect to the semantics \( I \) using \( \Downarrow^I \), whenever it is necessary to distinguish the reductions with respect to semantics \( I \) from those of the operational semantics.

**MENCALHit2**

\[
\frac{f \in \text{Mem} \quad (f \sigma(x), v) \in H \quad \Gamma \vdash f \Downarrow^I u, (C', \mathcal{H}', \sigma')}{}{\Gamma : (C, \mathcal{H}, \sigma) \vdash f \Downarrow^I u, (C', \mathcal{H}', \sigma')}
\]

The semantics \( I \) is an over-approximation of the operational semantics. That is, if the semantics \( I \) evaluates an expression \( e \) to a value \( u \) under and environment \( \Gamma \), then the operational semantics also produces an equivalent value for the expression under the environment \( \Gamma \), while consuming the same amount of resources. But, the semantics \( I \) may have more crashes compared to the operational semantics. The following lemma formalizes this property.

**Lemma 18.** Let \( P \) be a program. For every expression \( e \) and environments \( \{\Gamma_1, \Gamma_2\} \subseteq \text{Env} \) such that \( \Gamma \approx \Gamma_2 \).

\[
\Gamma_2 \vdash e \Downarrow^I v, \Gamma_2' \Rightarrow \left( \Gamma_1 \vdash e \Downarrow^I u, \Gamma_1' \downarrow u \approx v \land \Gamma_1' \approx \Gamma_2' \right)
\]

**Proof.** We prove this using structural induction on the evaluation \( \Gamma_2 \Downarrow^I v, \Gamma_2' \). The base cases are the rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MENCALHit2 and CACHED. In all cases except MENCALHit2 it is easy to see that the
claim holds since the rules in semantics I and the operational semantics are identical in these cases. Say \( \Gamma_2 \vdash e \Downarrow^I v, \Gamma_2' \) uses p \( MemoCallHit2 \). By definition \( p = c_{hit} \). Firstly, the evaluation \( \Gamma_1 \vdash e \Downarrow u, \Gamma_1' \) will use the rule \( MemoCallHit \). This is because \( C_1 \approx_{\mathcal{H}_1, \mathcal{H}_2} C_2 \) and \((f, \sigma_2(x)), v) \in H_2 \) \( C_2 \). By the definition of the rules \( MemoCallHit2 \) and \( MemoCallHit \), the cache and sigma components of \( \Gamma_1' \) and \( \Gamma_2' \) are identical to \( \Gamma_1 \) and \( \Gamma_2 \), respectively. Moreover, \( H_2 \subset H_2' \) and \( H_1 \subset H_1' \). Therefore, \( \Gamma_1' \approx \Gamma_2' \). The resource usage \( p = c_{hit} \) in both cases. We now show that \( u \approx v \).

By the definition of \( MemoCallHit2 \), we know that \( \Gamma_2 \vdash (f, \sigma_2(x)) \Downarrow u, (C', H_2', \sigma') \) for some \( C' \) and \( \sigma' \). By the inductive hypothesis, \( \Gamma_1 \vdash (f, \sigma(x)) \Downarrow u', (C'', H', \sigma') \) for some \( C'' \), \( H' \) and \( \sigma' \), and \( u' \approx_{H_1', H_2'} v \). Since \( WeakCacheCorr(\Gamma_1) \) holds, \( u' \approx u \). Hence, by the properties of \( \approx \), \( u \approx v \). But since \( u \in dom(H_1) \) and \( H_1 = H_1' \subseteq H_2', u \approx_{H_1', H_2'} v \). Hence the claim. It is easy to see in each of the inductive cases that the claim holds since they are identical in both semantics I and the operational semantics, and because \( \Gamma_1 \approx \Gamma_2 \).

The following Lemma formalizes that the semantic I is sound for contract verification.

**Lemma 19.** Let \( P \) be a program. Let \( \tilde{e} = \{ p \} e \{ s \} \) and let \( \tilde{e} \vdash \tilde{e} \) be a function definition in \( P \). If \( \forall \Gamma \in Env, p. \exists u. \Gamma \vdash e \Downarrow \tilde{e} \Downarrow u \) then \( \forall \Gamma \in Env, p. \exists u. \Gamma \vdash e \Downarrow \tilde{e} \Downarrow u \).

**Proof.** The claim directly follows from the Lemma 18, and the fact that for all \( \Gamma \in Env, \Gamma \approx \Gamma \).

Now consider the other direction namely completeness of the semantics I for contract checking. In the sequel we assume that the program \( P \) under consideration has only cache-monotonic contracts. Notice that Lemma 19 holds even without this assumption.

**Lemma 20.** Let \( P \) be a program in which all contracts of all function definitions are cache monotonic. For every expression \( e \) and environments \( \{ \Gamma_1, \Gamma_2 \} \subseteq Env \) such that \( \Gamma_1 \approx \Gamma_2 \).

\[
\Gamma_1 \vdash e \Downarrow u, \Gamma_1' \Rightarrow \left( \Gamma_2 \vdash e \Downarrow^I u, \Gamma_2' \wedge \Gamma' \approx \Gamma_1' \approx \Gamma_2' \right)
\]

**Proof.** We slightly adapt the structural induction strategy for this lemma. Given that \( CacheCorr(\Gamma_1) \) is a domain invariant (as proven in Lemma 15), we know that \( \forall k \in dom(C) \), \( \Gamma_1 \vdash k \Downarrow u, (C', H', \sigma', F) \wedge v \approx C(k) \). Therefore, there exists an evaluation among these, and \( \Gamma_1 \vdash e \Downarrow v, \Gamma_1' \), that has the largest depth. We induct on the depth of that evaluation. This allows us to use the hypothesis even on the evaluations such as the above.

The proof of this lemma is very similar to the proof of Lemma 18, except for the case of \( MemoCallHit \). Say \( \Gamma_1 \vdash e \Downarrow u, \Gamma_1' \) uses \( MemoCallHit \). The evaluation \( \Gamma_2 \vdash e \Downarrow u, \Gamma_2' \) will use the rule \( MemoCallHit2 \), since \( \Gamma_1 \approx \Gamma_2 \). Since \( CacheCorr(\Gamma_1), \Gamma_1 \vdash (f, \sigma(x)) \Downarrow u', \Gamma' \) and \( u \approx u' \).

By inductive hypothesis, \( \Gamma_2 \vdash (f, \sigma(x)) \Downarrow v, \Gamma'' \), and \( u' \approx_{H_1', H_2'} v \). Therefore, the antecedents of the \( MemoCallHit2 \) rule holds.

Hence, \( \Gamma_2 \vdash e \Downarrow^I v, \Gamma_2' \). By the definition of the rules \( MemoCallHit2 \) and \( MemoCallHit \), the cache and sigma components of \( \Gamma_1' \) and \( \Gamma_2' \) are identical to \( \Gamma_1 \) and \( \Gamma_2 \), respectively. Moreover, \( H_2 \subset H_2' \) and \( H_1 \subset H_1' \). Therefore, \( \Gamma_1' \approx \Gamma_2' \). The resource usage \( p = c_{hit} \) in both cases.

From the above facts, \( u' \approx_{H_1', H_2'} v \) and \( u \approx u' \) and \( H'' = H_2' \) (by the definition of \( MemoCallHit2 \)). Hence, \( u \approx_{H_1', H_2'} v \) Since \( u = C(k) \in dom(H_1) \), for some \( k \approx (f, \sigma(x)) \), and \( H_1 = H_1' \subseteq H_2', u \approx_{H_1', H_2'} v \). Hence the claim.

The following Lemma formalizes that the semantic I is complete for contract verification if the contracts in the program are cache monotonic.

**Lemma 21.** Let \( P \) be a program. Let \( \tilde{e} = \{ p \} e \{ s \} \) and let \( \tilde{e} \vdash \tilde{e} \) be a function definition in \( P \). If \( \forall \Gamma \in Env, p. \exists u. \Gamma \vdash e \Downarrow \tilde{e} \Downarrow u \) then \( \forall \Gamma \in Env, p. \exists u. \Gamma \vdash e \Downarrow \tilde{e} \Downarrow u \).

**Proof.** The claim directly follows from Lemma 20, and the fact that for all \( \Gamma \in Env, \Gamma \approx \Gamma \).

**B. Semantics of Model Programs**

In this section, we formalize the semantics of language constructs newly introduced in the model language and subsequently characterize the model environments.

**Semantics of Set Constructs.** Fig. 11 shows the semantics of the set operations that are used in the model generation. The semantics assumes expressions are in A-normal form, as in the case of Fig. 4. For brevity, the translation shown in Fig. 5 creates terms not in A-normal form. They can be lifted to A-normal form by introducing new let binders.

**Valid Model Environments.** We now formally define \( Env^\delta_{\epsilon, P} \) for an expression \( e \) belonging to a model program \( P^\delta \). Recall that to define \( Env_{\epsilon, P} \), we considered all clients \( P' \) that closes \( P \) and all the environments that may reach \( e \) in such closed programs. A similar definition for a model program \( P^\delta \) is possible. However, since the cache in the model program is an expression of the model program, considering all possible clients of \( P^\delta \) is an overkill because it may include clients that do not update the expression denoting the cache in accordance with the operational semantics of the input language. Therefore, we define the valid environments of the model \( P^\delta \) using the valid environments of program \( P ^\delta \). In other words, we only consider the clients of the model program that are consistent with the clients of the input program.

We now define a relation \( \approx \) very similar to \( \approx \) defined in section 3. Let \( hash_Y : \text{Lam} \to \mathbb{N} \) be a function that maps structurally equal lambdas in \( \Gamma \) to the same natural number. That is,

\[
\forall \epsilon, \epsilon' \subseteq \text{range}(H). \epsilon \approx_H \epsilon' \Rightarrow \text{hash}(\epsilon) = \text{hash}(\epsilon')
\]

Define \( dom_P(C) \) as the set of all keys in the cache \( C \) that refer to functions in the program \( P \). That is,

\[
dom_P(C) = \{ (f, u) \in \text{dom}(C) | f \text{ is defined in } P \}
\]
Define a relation \( \sim \) between the semantic domains of the input and the model language as follows:

1. If \( a \in \mathbb{Z} \cup \mathbb{B} \cup \mathbb{R} \), \( a \sim a \).
2. For \( c \in \mathbb{C} \), \( \{a, b\} \subseteq \mathbb{V} \), \( c \sim c \) if and only if \( \forall i \in [1, n], a_i \sim b_i \).
3. Let \( \langle e, \sigma \rangle \) be a closure, \( e \in \mathbb{L} \), \( \sigma \in \mathbb{S} \), \( \sim \) is the closure relation if \( \sigma(FV(e)) \sim \sigma \).

Proof. Let \( \Gamma \) be a program. Let \( u, v \in \mathbb{L} \) and \( H = \{x\} \). The simulation relation \( \sim \) is preserved by the structural equality relations \( \sim \) and \( \sim \) and vice versa. That is, \( (u \sim \sim v) \text{ and } (u \sim \sim v) \).

Lemma 23. Let \( P \) be a program. Let \( u, v \in \mathbb{L} \) and \( H, H' \) be two heaps. The simulation relation \( \sim \) is preserved by the structural equality relations \( \sim \) and \( \sim \) and vice versa. That is, \( (u \sim \sim v) \text{ and } (u \sim \sim v) \).

Proof. We omit the subscripts of \( \sim \) and \( \sim \) in the rest of the proof. We show the proof for one part: \( u \sim v \text{ then } (u' \sim v \Rightarrow u \approx u') \).

Say \( u \approx u' \). We now show that \( u' \sim v \) using structural induction on \( \approx \). If \( u \) is an integer or boolean, the claim follows immediately as \( u' = u \). Say \( u \) is an address of a closure \( (\sigma, \Gamma) \) i.e., \( H(u) = (C, T) \). By the definition of \( \approx \) and \( \sim \), \( u \) and \( v \) are also addresses of \( (C, T) \) and \( (C, T) \) such that for all \( i \in [1, |u|], w_i \approx u_i \) and \( w_i \sim v_i \), respectively. By inductive hypothesis, \( w_i \sim v_i \).

Now say \( u \) is an address of a closure \( (\sigma, \Gamma) \). If \( e_{/\varnothing, \varnothing} \) is defined and has label \( l, v = (C, l) \), \( \sigma'((FV(e)) \sim \sigma \).

If \( e_{/\varnothing, \varnothing} \) is not defined, \( v = (C_{\text{type}}(\sigma), h(\sigma)) \). Since \( u \approx u' \), \( u' = (\sigma', \Gamma) \) and \( e_{/\varnothing, \varnothing} \) is not defined. By the definition of \( h(\sigma), \sigma' \approx \sigma \).

Say \( u \sim v \). We now show that \( u \approx u' \) using structural induction on \( \sim \). If \( u \) is an integer or boolean the claim immediately follows as in that case \( u \sim v \). Say \( u \) is an address of a constructor \( (C, T) \). We are given that \( \forall x \in \mathbb{L}(P), \exists y \sim \mathbb{L}, x = y \).

Lemma 24. Let \( P \) be a program. Let \( \Gamma \in \mathbb{L} = (C, T) \), \( \forall x \in \mathbb{L}(P), \exists y \sim \mathbb{L}, x = y \).

Proof. The first part of the claim follows by the definition of \( \sim \).

Correctness of the model programs for contract verification. Below we establish that if \( \Gamma \sim (\Gamma', S) \), evaluating an expression \( e \) under \( \Gamma \) results in fewer crashes than evaluating the translation

\[
\text{Figure 11. Semantics of set operations used by the model.}
\]

\[
\text{Table 11. Semantics of set operations used by the model.}
\]
of e under Γ. That is, Γ progresses as long as Γ♯ progresses on the translation of e.

**Lemma 26.** Let P be a program. Let st be an expression of the model language. Let Γ ∈ Envp and Γ♯ ∈ Env♯p be such that Γ♯ ⊨ st ⊨ S and Γ ∼ (Γ♯, S). Let e be any expression. If Γ♯ ⊨ ([e]♯ p) st ⊨ u, Γ♯ then ∃o. env ∈ Env, v ∈ Val, p ∈ N such that Γ ⊨ e ♯ p v, Γ_o and

$$\bullet \quad Γ_o \sim (Γ^o, u, 2)$$

$$\bullet \quad v \sim u, 1$$

$$\bullet \quad p = u, 3$$

**Proof.** We prove this using structural induction over the evaluation

$$Γ♯ ⊨ ([e]♯ p) st ⊨ u, Γ♯$$

**Base cases.** Say the evaluation

$$Γ♯ ⊨ ([e]♯ p) st ⊨ u, Γ♯$$

uses one of the rules: CST, VAR, PRIM, EQUAL, CONS, LAMBDA and CACHED. Let e′ = [e]♯ p st. The free variables of e′ and e are identical, and by the definition of ⊳, fn(e) ⊆ dom(σ♯) = dom(σ) ∪ {st}. Hence, there is a value defined for all free-variables in σ. Since Γ satisfies all the domain invariants, the antecedent of every base case rule is defined. Therefore, Γ♯ ⊨ e ♯ p v, Γ_o for some v, p and Γ_o.

We now establish the claim: p = u, 3 in all the base cases. In all base cases, the cost of the operation op is a constant as per the semantics I, and is exactly same as u, 3 as per the translation [ ] ♯ p.

Therefore, p = u, 3 holds trivially.

Consider now the claim: Γ_o ∼ (Γ^o, u, 2). Recall that the relation

$$∼$$

is monotonic with respect to the ordering ⊆ between the heaps. In all the base cases, the cache and store components of the input and the output environments Γ and Γ♯ are identical. The heaps of Γ and Γ♯ are contained in the heaps of Γ_o and Γ^o. Moreover, as per the translation, st and u, 2 are also identical. Therefore, by Lemmas 7 and 22, Γ ∼ (Γ♯, S) directly implies

$$Γ_o \sim (Γ^o, u, 2)$$

Consider now the claim: v ∼ u, 1. In the case of CST it is easy to see that the values returned by e are identical primitive values (in N ∪ Bool) in both evaluations under Γ and Γ♯. In the case of PRIM, the arguments of the operations are integer or boolean. By the definition of ∼, the arguments are equal in both σ and σ♯. Hence, the output of PRIM is also equal under both environments. (We allow only deterministic primitive operations.)

Therefore, v ∼ u, 1 in both cases.

Consider the case of VAR. Say σ(x) = a and σ♯(x) = a′. It is given that a ∼ a′. By definition, v = a and u, 1 = a′.

Hence, the claim holds by Lemmas 7 and 22. In the case of CONS, (a → cons σ(x)) is added to H and (a′ → cons σ♯(x)) is added to H♯, for some fresh a and a′ that are not bounded in H and H♯, respectively. It is given that σ ∼ σ♯. Therefore, a ∼ a′ by the definition of ∼, which by Lemma 22 implies a ∼ a′. Therefore,

$$v \sim u, 1$$

The LAMBDA case can be similarly proved.

Consider now the CACHED case, i.e, e is cached(f x) (for some f and x). We are given that σ(x) ∼ σ♯(x). By the definition of

$$∼$$

of (f x), provided f is defined in the program P, which holds because we require that every named function used in the program are defined in the program. By Lemma 25,

$$\exists u′_f (C_f, u′) ∈ S ∧ u′ ≈ σ(x),$$

where Γ♯ ⊨ st ⊨ S, if and only if

$$\exists u_f (u) ∈ dom(C) ∧ u ≈ σ(x).$$

By the semantics of set inclusion shown in Fig. 11 and e_i(C_f, x) ∈ st evaluates to true under Γ♯ iff cache(f x) evaluates to true under Γ.

Consider now the rule EQUAL. That is, e is of the form x eq y. This evaluates to true under Γ if f(x) ≈ y.(y). It is given that σ(x) = σ♯(x) and σ(y) = σ♯(y). By Lemma 23, if σ(x) ≈ σ♯(y) then σ(y) = σ(x), which in turn by the same lemma implies that σ♯(x) = σ♯(y). Similarly, if σ(x) = σ♯(y) is false then by Lemma 23, ¬(σ(y) ∼ σ♯(x)) which in turn implies that ¬(σ♯(x) = σ♯(y)). Hence, the claim.

**Proof of Inductive Step.** Say the evaluation

$$Γ♯ ⊨ ([e]♯ p) st ⊨ u, Γ♯$$

matches one of the rules: LET, MATCH, CONCRETECALL, CONTRACT, INDIRECTCALL, MEMOCALLHit (which is an inductive case in semantics I), and MEMOCALLMiss. The last three rules list above are the most interesting ones. The rest follow by a simple inductive reasoning.

Consider now the MEMOCALLHit rule. In this case p = u, 3 = cons. Also, Γ_o ∼ (Γ^o, u, 2), since the output caches, state expressions and stores are identical to the input in both evaluations Γ and Γ♯, and the output heaps are only larger. Consider now the claim: v ∼ u, 1. Here, v and u, 1 are the result of the evaluation

$$Γ_o \sim (Γ^o, u, 2)$$

for MEMOCALLMiss case is very similar, except that in this case C♯ is defined in both new entry C♯(x) (by the semantics of set union). However, C_o is also added a new entry (f(x)) → v. Since (f(x)) ∼ C♯(x) by definition, the claim that Γ_o ∼ (Γ^o, u, 2) holds in this case as well.

Consider now the case INDIRECTCALL i.e, e = (x y)♯. The translated expression [e]♯ p st invokes the function App# defined in Fig. 6. Let σ(x) = (e, x, y)♯. The case is very similar, except that in this case C♯ is defined in the evaluated expression App# by the semantics of set union. However, C_o is also added a new entry (f(x)) → v. Since (f(x)) ∼ C♯(x) by definition, the claim that Γ_o ∼ (Γ^o, u, 2) holds in this case as well.

Now say e♯/x,y = p = (λx. f(x, z), y)♯, where dom(σ♯) = {z}. By definition of ∼, target(σ♯) = f. By the definition of ∼, σ♯(x) = (C♯, I) where σ(˜z) = ˜z. By the definition of App# (Fig. 6) and the match construct, Γ♯ ⊨ ([e]♯ p) st ⊨ u, Γ♯ reduces to Γ♯ ⊨ ([f]♯ p) st ⊨ u, Γ♯, where Γ♯ = (H♯, σ♯ ∪ (y, y)♯). Now consider Γ♯ = (C, H, σ ∪ σ♯).

Clearly, Γ♯ ∼ (Γ♯, σ♯(x)). Therefore, by induction hypothesis, Γ♯ ⊨ (y, z)♯ p, Γ_o, Γ_o ∼ (Γ_o, S), p = u, 3 and u, 1. Therefore, (y, z)♯ ∈ dom(σ) so that the calls are syntactically identical and the induction hypothesis can be applied.) By the definition of the rule INDIRECTCALL, the above implies that Γ♯ ⊨ e ♯ p v, Γ_o and hence the claim holds.

□

**Corollary 27.** Let P be a program. Let st be an expression of the model language. Let Γ ∈ Envp and Γ♯ ∈ Env♯p be such that Γ♯ ⊨ st ⊨ S and Γ ∼ (Γ♯, S). Let e be any expression. If Γ♯ ⊨ ([e]♯ p) st ⊨ u, Γ♯ then ∃o. env ∈ Env, v ∈ Val, p ∈ N such that Γ ⊨ e ♯ p v, Γ_o and

$$\bullet \quad Γ_o \sim p (Γ^o, u, 2)$$

$$\bullet \quad v \sim u, 1$$

$$\bullet \quad p = u, 3$$

**Proof.** Notice that here the environments (and the states) are related by ∼ ♯ which is stronger than ∼. Thus the only fact that is not implied by Lemma 26 is that Γ_o ∼ p (Γ^o, u, 2). It is easy to
see that this property holds from the proof argument of the above lemma.

We now show that the evaluation of an expression $e$ under an environment $\Gamma$ with respect to the semantics $I$, and the evaluation of $[e]_{\Gamma} s t$ under $\Gamma^2$ such that $\Gamma \sim (\Gamma^2, \sigma^2(st))$ bisimulate each other, provided every indirect call invoked by the expression $e$ during its evaluation under $\Gamma$ is an encapsulated call (see section 2 for the definition of encapsulated calls).

**Lemma 28.** Let $P$ be a program. Let $st$ be an expression of the model language. Let $\Gamma \in Env^P$ and $\Gamma^3 \in Env^P$ be such that $\Gamma^3 \rightarrow st \Downarrow S$ and $\Gamma \sim (\Gamma^3, S)$. Let $e$ be any expression such that if $(\Gamma, e) \rightarrow^* (\Gamma, x, y)$, $\mathcal{H}(\sigma^2(x)) = (\sigma^1, \sigma^1')$ and $l \in \text{labels}_P$. If $\Gamma \vdash e \Downarrow_P v, \Gamma_0$ then $\exists \nu^2 \in Env^P, u \in \text{DVal}$ such that $\Gamma^3 \vdash (\{e\}_P st) \Downarrow_P u, \Gamma_0^3$ and

- $\Gamma_0 \sim (\Gamma_0^3, u_2)$
- $v \Downarrow_P u_1$
- $p = u_3$

**Proof.** The proof of this lemma is very similar to the proof of Lemma 26 expect for a minor difference in the handling of the case where $e$ is an indirect call. We are given that every indirect call encountered during the evaluation of $e$ is an encapsulated call. As a result, when the expression $e$ is an indirect call $(x \ y)^{X}$ (the rule INDIRECT-CALL), we are guaranteed that $\sigma^1(x) = (\sigma^1, \sigma^1')$ and $e_2 / \gamma, \nu , p$ is defined, since $e_2$ itself belongs to the program $P$. Thus, the evaluation of $[e]_{\Gamma} s t$ under $\Gamma^3$ cannot go through the error case of Appy function, which implies that $\Gamma^3 \vdash (\{e\}_P st) \Downarrow_P u, \Gamma^3$ will be defined for the rule INDIRECT-CALL.

**Corollary 29.** Let $P$ be a program. Let $st$ be an expression of the model language. Let $\Gamma \in Env^P$ and $\Gamma^3 \in Env^P$ be such that $\Gamma^3 \rightarrow st \Downarrow S$ and $\Gamma \sim (\Gamma^3, S)$. Let $e$ be any expression. If $\Gamma \vdash e \Downarrow_P v, \Gamma_0$ then $\exists \nu^2 \in Env^P, u \in \text{DVal}$ such that $\Gamma^3 \vdash (\{e\}_P st) \Downarrow_P u, \Gamma_0^3$ and

- $\Gamma_0 \sim (\Gamma_0^3, u_2)$
- $v \Downarrow_P u_1$
- $p = u_3$

**Proof.** Notice that here we do not have the assumption that $e$ invokes only encapsulated, indirect calls, since this is implied by the fact that the environments are related by the stronger relation $\sim_P$. Analogous to Lemma 27, the only fact that is not implied by Lemma 28 is that $\Gamma_0 \sim_P (\Gamma_0^3, u_2)$. It is easy to see that this property holds from the proof argument of the above lemma.

**Theorem 1. (Bisimulation.)** Let $P$ be a program. Let $st$ be an expression of the model language. Let $e' = [e]_{\Gamma} s t$. Let $\Gamma \in Env^P$ and $\Gamma^3 \in Env^P$ be such that $\Gamma^3 \rightarrow st \Downarrow S$ and $\Gamma \sim_P (\Gamma^3, S)$.

(a) If $\Gamma \vdash e \Downarrow_P v, \Gamma_0$ then $\exists \nu^2 \in Env^P, u \in \text{DVal}$ such that $\Gamma^3 \vdash e' \Downarrow_P u, \Gamma_0^3$ and

- $\Gamma_0 \sim_P (\Gamma_0^3, u_2)$
- $v \Downarrow_P u_1$
- $p = u_3$

(b) If $\Gamma^3 \vdash e' \Downarrow_P u, \Gamma_0^3$ then $\exists \nu^2 \in Env^P, v \in \text{Val}, p \in \mathbb{N}$ such that $\Gamma \vdash e \Downarrow_P v, \Gamma_0$ and

- $\Gamma_0 \sim_P (\Gamma_0^3, u_2)$
- $v \Downarrow_P u_1$
- $p = u_3$

**Proof.** This theorem follows from the Corollaries 27 and 29, and the fact that the semantics $I$ bisimulates the operational semantics as established by Lemmas 18 and 20.

The following theorem establishes the soundness and correctness of the model programs. While the soundness of the model holds regardless of the cache monotonicity requirement of contracts, for completeness we expect that the contracts in the program satisfy the property (since semantics $I$ is complete under that condition as shown in Lemma 20). The fact that the translation $\{\cdot\}_P$ is sound regardless of cache monotonicity of contracts provides a way to check this property using the translation $\{\cdot\}_P$.

Also, it is to be noted that the completeness theorem is proven only for the language used in the formalism and described in section 2. In particular, the completeness proof uses the property that the fields of a constructor can be created at any point and, also that any constructor can be created at any point in the program by passing in type-correct arguments. In particular in a language that supports access modifiers like private/public, the completeness property becomes more trickier to establish.

**Theorem 2. (Model Soundness and Completeness)** Let $P$ be a program and $P^2$ the model program. Let $\bar{e} = \{p\} e \{s\}$ and $\bar{e}' = \{p'\} e' \{s'\}$. Let $def\ x := \bar{e}$ be a function definition in $P$, and let def $f x := \bar{e}'$ be the translation of $f$, where $st$ is the state parameter added by the translation.

- If $\forall \nu^2 \in Env^P, \exists \nu. \Gamma \vdash \nu^2 \Downarrow_P v$ then $\forall \nu \in V^P, \exists \nu. \Gamma \vdash \nu \Downarrow_P v$ iff

Proof. Firstly, for every $\Gamma^3 \in Env^P$ there exists an environment $\Gamma \in Env^P$ such that $\Gamma \sim_P (\Gamma^3, \sigma^2(st))$ and vice-versa. That is the relation $\sim_P$ is total with respect to the domains $Env^P$ and $Env^P$.

If $\Gamma \vdash e \Downarrow_P v, \Gamma_0$ then $\exists \nu^2 \in Env^P, u \in \text{DVal}$ such that $\Gamma^3 \vdash (\{e\}_P st) \Downarrow_P u, \Gamma_0^3$, and

- $\Gamma_0 \sim (\Gamma_0^3, u_2)$
- $v \Downarrow_P u_1$
- $p = u_3$

**Proof for call-encapsulated programs.** In this case, we consider the programs where every indirect call in the program is an encapsulated call. In this case the claim follows from the totality of $\sim_P$ relation and the Lemmas 26 and 28. Note that the requirements of the Lemma 28 hold, since the program is call-encapsulated.

**Proof for general programs.** The only-if direction (souness) directly follows from Lemma 26 and the totality of $\sim_P$ described above. Below, we prove the if direction (completeness). That is, if $\forall \nu^2 \in Env^P, \exists \nu. \Gamma \vdash \nu \Downarrow_P v$ and $\Gamma \vdash \nu \Downarrow_P v$ when $\forall \nu^2 \in Env^P, \exists \nu. \Gamma \vdash \nu \Downarrow_P v$. Now, there are three cases to consider. If $(\Gamma, e) \rightarrow^* (\Gamma, c, a)$ implies $\mathcal{H}(\sigma^2(c)) = (\sigma^1, \sigma^1')$ and $l \in \text{labels}_P$, then by Lemma 28 the claim holds.

Therefore, say $(\Gamma, e) \rightarrow^* (\Gamma, c, a)$, for some $n \in \mathbb{N}$, and $\mathcal{H}(\sigma^2(x)) = (\sigma^1, \sigma^1) \wedge \sigma^1 \notin \text{labels}_P$. That is, the evaluation of $e$ under $\Gamma$ invokes a lambda created outside the program. Without loss of generality assume that $c$ is the first such call. That is, every call reached before $n$ steps is an encapsulated call. Now, it is easy to see that $\mathcal{H}(\sigma^2(c)) = \mathcal{H}(\sigma^2(c))$. This is because if $\sigma^2(c)$ is not bound in the input heap, it has to be bound subsequently. But we know that every expression until encountering the call $c a$ belongs to the program $P$ since we assume that $c a$ is the first call-back that executes code outside $P$. Thus, any closure created during the evaluation of $e$ until $c a$ belongs to $P$. Therefore, $\sigma^2(c)$ should be bound in the input heap. Let $\sigma^2(c) = a$ and $\mathcal{H}(a) = (\lambda r. h(r, s), \sigma^1 a)$. Now, consider a new environment $\Gamma_{err}$ defined as follows: $\Gamma_{err} = (C, \mathcal{H}[\alpha \rightarrow \mathcal{M}(v)])$. Therefore, say $(\Gamma, e) \rightarrow^* (\Gamma, c, a)$, for some $n \in \mathbb{N}$, and $\mathcal{H}(\sigma^2(x)) = (\sigma^1, \sigma^1) \wedge \sigma^1 \notin \text{labels}_P$. That is, the new environment wraps the body of the lambda $H(a)$ by a contract whose precondition is false. By the totality of $\sim_P$.

23

2017/5/16
\[ \exists \eta \in \text{Env}^\tau_{e,p} \text{ such that } \Gamma \Rightarrow (\Gamma^2, \sigma^2(st)) \]. Note that firstly (a) \( \lambda r . h (r,s) \eta \) \( r,s, \eta \) will not be defined as it is external to the program \( P \). Consider now the following definition of the hash function for the newly introduced lambdas: \( \text{hash}(\langle \lambda r . h (r,s) , \sigma^2 \rangle) = \text{hash}(\langle \lambda r . h (r,s), \sigma^2 \rangle) \). Clearly, this hash function preserves structural equality, i.e., \( \forall \{ e_1, e_\eta \} \subseteq \text{range}(\mathcal{H}_{\text{err}}) \). \( e_\eta \approx H_{\text{err}} \) \( e_1 \Rightarrow \text{hash}(e_\eta) = \text{hash}(e_\eta) \), and hence is well-defined. Therefore, it is easy to see that \( \Gamma_{\text{err}} \Rightarrow (\Gamma^2, \sigma^2(st)) \) by our construction. Hence, \( \langle \Gamma_{\text{err}}, e \rangle \sim^* (\Gamma_1', f(z)) \) and \( \exists \exists \Gamma' \sim (\Gamma_{\text{err}}, S) \). Clearly, evaluating \( (c a) \) under \( \Gamma_{\text{err}} \) results in a contract violation as the precondition of \( g \) will not hold. Now, if \( \Gamma_{\text{err}} \in \text{Env}_{e,p} \) we get a contradiction to our assumption that the contract of the function \( f \) holds in all valid environments, which implies the claim. We now complete the proof by showing that \( \Gamma_{\text{err}} \in \text{Env}_{e,p} \).

Since \( \Gamma \in \text{Env}_{e,p} \), there exists a program \( P' \) such that \( \langle \Gamma_{\text{err}}, e_{\text{env}} \rangle \sim^* (\Gamma, e) \). This implies that \( \langle \Gamma_{\text{err}}, e_{\text{env}} \rangle \sim^* (\Gamma_1, f(z)) \). where \( f \) is the function whose contracts we are trying to verify, and \( \Gamma_1 = (C, H, \sigma[z \mapsto \sigma(x)]) \) is the environment before parameter translation. Let \( P'' \) be a program obtained by augmenting \( P' \) with the function \( g \) defined as above (renaming \( g \) if there already exists a function with the same name in \( P' \)). Let \( \Gamma_{\text{err}} = (\mathcal{C}_{\text{err}}, H_{\text{err}}, \sigma_{\text{err}}[z \mapsto \sigma(x)]) \), \( \Gamma_{\text{err}} \) before parameter translation. In our language, given a value \( w \) and an environment \( \Gamma = (C, H, \sigma, F) \) it is possible to create an expression \( e \) such that \( \Gamma \vdash e \Downarrow v \). \( (C, H, \sigma) \) such that \( v \approx w \). This is because a value \( v \) is, in principle, a closed expression without free variables obtained by recursively replacing each address \( a \) by its mapping in the heap \( H(a) \). The recursion will stop as the heaps are acyclic. For brevity, we ignore the formal construction of such an expression. (Since our language doesn’t support any access modifiers, every constructor can be constructed at any point in the program.)

Given this property, we construct an expression \( e_{\text{env}} \) that produces the value \( \sigma_{\text{env}}(z) \). Since \( e_{\text{env}} \) is closed it can be inserted at any point in the program. We replace the call \( (f z) \) by the expression let \( z := e_{\text{env}} \) in \( (f z) \). Let the new program thus obtained be \( P_3 \). Thus, there exists a program \( P_3 \) such that \( \langle \Gamma_{\text{err}}, e_{\text{env}} \rangle \sim^* (\Gamma, e) \). Hence, \( \Gamma_{\text{err}} \in \text{Env}_{e,p} \).

\[ \square \]

C. Correctness of Verification

Reducing Error construct to a precondition. Recall that the model programs use an error construct in the bodies of App functions corresponding to (non-encapsulated) indirect calls. Let \( \text{def App}(c, x, st) \) be one such function corresponding to an indirect call \( (y z) \). The error construct will be encountered during the evaluation of \( \text{App} \) if and only if \( c \equiv C_{\text{typ}}(y) \). In this case, the result of the evaluation is undefined. The same effect can be achieved if we add a precondition to \( \text{App} \) namely \( c \neq C_{\text{typ}}(y) \). It is obvious that the \( \text{App} \) with the precondition is equivalent to the \( \text{App} \) function with the error construct. For simplicity, in the rest of section, we assume that the model programs are free of error constructs, which have been lifted to the preconditions of \( \text{App} \) functions. This provides us the property that the Lemma 17 applies to the model programs as well.

C.1 Soundness of Assume/Guarantee Reasoning

In this section, we formalize and prove the soundness of the assume/guarantee reasoning explained in section 4 in more detail and prove its correctness. Note that we apply the assume/guarantee reasoning only on the model programs, which has only direct calls due to defunctionalization.

Let us first formally define the assume/guarantee assertion \( \models e_1 \Rightarrow e_2 \). As defined in section 4, let \( e_1 \Rightarrow e_2 \) be

\[ \forall \Gamma \in \{(C, H, \sigma, F) \in \text{Env}^\tau \mid x \in \text{dom}(\sigma) \} \]

\[ \Gamma \vdash e_1 \Downarrow \text{false} \lor \Gamma \vdash e_2 \Downarrow \text{true} \]

Note that in the above definition we only consider environments that have a binding for the parameter \( x \), since we know this is guaranteed by the semantics (Lemma 9). Let \( \text{fe}(e) \) denote the set of free variables in the expression \( e \).

We define an assumption \( \mathcal{A}_P(\Gamma, e) \) as:

\[ \bigwedge \left\{ \exists \exists v. \Gamma' \vdash (f(z)) \Downarrow v \mid \Gamma' \in \text{Calls}(\Gamma, e) \right\} \]

That is the pre-and post-conditions of all the callees transitively invoked by \( e \) are satisfied and the callees are terminating in the environment that reaches them. An assume/guarantee assertion \( \models e_1 \Rightarrow e_2 \) denotes the following:

\[ \forall \Gamma \in \{(C, H, \sigma, F) \in \text{Env}^\tau \mid \text{fe}(e_1) \cup \text{fe}(e_2) \subseteq \text{dom}(\sigma) \} \]

\[ \neg \mathcal{A}_P(\Gamma, e_1) \land \neg \mathcal{A}_P(\Gamma, e_2) \land \Gamma \vdash e_1 \Downarrow \text{false} \lor \Gamma \vdash e_2 \Downarrow \text{true} \]

Helper Functions. We define a few helper functions used by the assume/guarantee assertions. Given a function \( \text{def } f \ x := \bar{e} \) in the model program. We define \( \text{path}(c) \) as any boolean-valued expression that satisfies the following property:

\[ \forall (H, \sigma) \in \text{Env}^\tau \text{ s.t. } \text{fv}(\bar{e}) \subseteq \text{dom}(\sigma) \]

\[ (\Gamma, \bar{e}) \sim^* (\Gamma', (g y)) \Rightarrow \Gamma' \vdash \text{path}((g y)\bar{e}) \Downarrow \text{true} \]

That is, every environment that reaches the call-site makes the expression \( e \) true.

Let \( e' \) be an expression within a function definition \( \text{def } f \ x := \bar{e} \) in the model program. We define \( \text{path}(c) \) as any boolean-valued expression that satisfies the following property:

\[ \forall (H, \sigma) \in \text{Env}^\tau s.t. \text{fv}(\bar{e}) \subseteq \text{dom}(\sigma) \]

\[ (\Gamma, \bar{e}) \sim^* (\Gamma', (g y)) \Rightarrow \Gamma' \vdash \text{path}((g y)\bar{e}) \Downarrow \text{true} \]

If the above rules hold, the following property holds for all \( n \in \mathbb{N} \)

\[ \forall (\text{def } f \ x := \bar{e}) \in P \ s.t. \bar{e} = \{p,e\} \{s\} \]

\[ \forall \Gamma' \in \{(C, H, \sigma, F) \in \text{Env}^\tau \mid x \in \text{dom}(\sigma) \} \]

\[ \exists k > n, h \in \text{Fids}, y \in \text{Vars}, \Gamma', (\Gamma, \bar{e}) \sim^k (\Gamma', (h y)) \]

\[ \lor (\exists v. \Gamma \vdash p \Downarrow \text{false} \lor \Gamma \vdash e \Downarrow v) \]

\[ \lor (\exists v. \Gamma \vdash p \Downarrow \text{false} \lor \Gamma \vdash e \Downarrow v) \]

2017/5/16

24
Proof. We prove this using induction on \( n \). Intuitively, \( n \) imposes a limit on the number of direct function calls we need to consider while proving that the contract of the function \( f \) holds. The base case are evaluations that make zero direct calls. For every function \( f \) we need to prove that
\[
\forall \Gamma \in \{ (\mathcal{C}, \mathcal{H}, \sigma, F) \in \text{Env}^f \mid x \in \text{dom}(\sigma) \}, \\
\left( \exists k > 0, h \in \text{Fids}, y \in \text{Vars}, (\Gamma, \vec{e}) \rightsquigarrow^k (\Gamma', (h y)) \right) \\
\vee (\exists \vec{e}, \Gamma \vdash p \not\vdash \forall \Gamma \vdash \vec{e} \downarrow v)
\]
Consider a \( \Gamma \) such that \( \neg (\exists k > 0, h, y, (\Gamma, \vec{e}) \rightsquigarrow^k (\Gamma', (h y))) \). Otherwise the claim trivially holds. This essentially means that we do not encounter a direct call either during the evaluation of \( p \) or \( \vec{e} \) under \( \Gamma \).

\[
\text{ Calls}(\Gamma, p) \cup \text{ Calls}(\Gamma, \vec{e}) = \emptyset
\]
\[\Rightarrow \text{ A}_p(\Gamma, p) \wedge \text{ A}_p(\Gamma, \vec{e}), \text{ by the def. of } \text{ A}_p \]
\[\Rightarrow p \rightarrow s[e/\text{res}], \text{ since } \models_p p \rightarrow s[e/\text{res}] \]
\[\Rightarrow \Gamma \vdash p \not\vdash \forall \Gamma \vdash (\exists s) \vdash v \]

By the operational semantics of contract expressions Fig. 4,
\[\Rightarrow \exists \vec{e}, \Gamma \vdash p \not\vdash \forall \Gamma \vdash \{ \Gamma' \} e \{ s \} \vdash v \]

Since every call-free evaluation terminates in our language
\[\text{and by Lemma 17,} \]
\[\Gamma \vdash p \not\vdash \forall \Gamma \vdash p \not\vdash \text{true} \]
\[\text{By 6 and 7,} \exists \vec{e}, \Gamma \vdash p \not\vdash \forall \Gamma \vdash \{ p \} e \{ s \} \vdash v \]

Hence the claim holds in the base case.

Inductive step: Assume that the claim holds for all evaluations with \( m+1 \) calls. We now show that the claim holds for all evaluations with \( m+1 \) calls. That is, we need to prove that
\[\forall \Gamma \in \{ (\mathcal{C}, \mathcal{H}, \sigma, F) \in \text{Env}^f \mid x \in \text{dom}(\sigma) \}, \]
\[\left( \exists k > m + 1, h \in \text{Fids}, y \in \text{Vars}, (\Gamma, \vec{e}) \rightsquigarrow^{k+1} (\Gamma', (h y)) \right) \]
\[\vee (\exists \vec{e}, \Gamma \vdash p \not\vdash \forall \Gamma \vdash \vec{e} \downarrow v) \]

As before, let us consider a \( \Gamma \) such that \( \neg (\exists k > m + 1, h, y, (\Gamma', \vec{e}) \rightsquigarrow^{k+1} (\Gamma', (h y))) \). Otherwise the claim trivially holds. That is, all direct calls made by \( \vec{e} \) under \( \Gamma \) have depth at most \( m+1 \). Let \( S \) denote the top-level calls made by \( \vec{e} \). These are all calls that appear in the syntax tree of \( \vec{e} \). Formally,
\[S = \{ (\Gamma', (g x)) \mid \exists i \in \mathbb{N}, (\Gamma', \vec{e}) \rightsquigarrow^i (\Gamma', (g x)) \}
\[\wedge \neg \exists j < i, h.((\Gamma', \vec{e}) \rightsquigarrow^j (\Gamma', (h x))) \]

Note that by the definition of \( \rightsquigarrow^i \), every call transitively made during the evaluation of \( \vec{e} \) should be reachable (w.r.t. \( \rightsquigarrow^i \)) from the body of a calle in \( S \) in \( \leq m \) depth (otherwise \( \vec{e} \) would invoke a call at a depth \( > m + 1 \) violating the assumption). That is,
\[\forall (\Gamma', (g y)) \in S \text{ s.t. } \text{ def } f := \{ \text{ pre}_g \} e_g \{ \text{ post}_g \} \in P, \]
\[\neg (\exists i > m, (\Gamma'[x \mapsto \sigma'(y)], \{ \text{ pre}_g \} e_g \{ \text{ post}_g \}) \rightsquigarrow^i (\Gamma'', (g x))) \]

By inductive hypothesis the above implies that
\[\forall (\Gamma'', (g y)) \in S \text{ s.t. } \text{ def } f := \{ \text{ pre}_g \} e_g \{ \text{ post}_g \} \in P, \]
\[\exists \vec{e}, \Gamma'[x \mapsto \sigma'(y)] \vdash \text{ pre}_g \not\vdash \forall \Gamma'[x \mapsto \sigma'(y)] \vdash \{ \text{ pre}_g \} e_g \{ \text{ post}_g \} \not\vdash v \]

Based on the operational semantics and the definition of \( \text{ pre } \), the above can be rewritten as
\[\forall (\Gamma', (g y)) \in S, \exists \vec{e}, \Gamma' \vdash \{ \text{ pre}_g \} e_g \{ \text{ post}_g \} \not\vdash v \]

As a consequence of the above fact we also know that every call invoked inside \( \text{ pre}_g (y) \) terminates and results in a value. That is,
\[\forall (\Gamma', (g y)) \in S, \text{ A}_p(\Gamma', \text{ pre}_g(y)) \]

Now consider the definition of the path condition \( \text{ path } \) of a call \( (g y) \) with label \( l \) contained in the body \( \vec{e} \) of a function \( f \). By definition,
\[\forall \Gamma \in \text{ Env}^f, (\Gamma, \vec{e}) \rightsquigarrow^* (\Gamma', (g y)) \Rightarrow \Gamma' \vdash \text{ path}(g y) \not\vdash \text{ true} \]
\[\Rightarrow \forall (\Gamma', (g y)) \in S, \text{ A}_p(\Gamma', \text{ path}(g y)) \]

That is, every environment that reaches \( (g y) \) will satisfy the path condition of \( (g y) \). We are given that the following assertion holds:
\[\forall \text{ call-site } c \in P \models_p \text{ path}(c) \rightarrow \text{ pre}(c) \]
\[\Rightarrow \forall (\Gamma', (g y)) \in S, \neg \text{ A}_p(\Gamma', \text{ path}(g y)) \vee \neg \text{ A}_p(\Gamma', \text{ pre}(g y)) \]
\[\Rightarrow \forall (\Gamma', (g y)) \models_p \text{ path}(g y) \not\vdash \forall (\Gamma', (g y)) \not\vdash \text{ true} \]
\[\Rightarrow \forall (\Gamma', (g y)) \in S, \exists \vec{e}, \Gamma' \vdash (g y) \not\vdash v, \text{ by 10} \]
\[\Rightarrow \forall (\Gamma', (g y)) \in \text{ Calls}(\Gamma, \vec{e}), \exists \vec{e}, \Gamma' \vdash (g y) \not\vdash v, \text{ by the def. of } \text{ Calls} \]
\[\Rightarrow \text{ A}_p(\Gamma, \vec{e}) \wedge \text{ A}_p(\Gamma, p) \]

Also, 19 implies that evaluations of \( p \) and \( \vec{e} \) terminates.

As in the base case, the above fact, 20 and \( \models_p p \rightarrow s[e/\text{res}] \) imply that
\[\exists \vec{e}, \Gamma \vdash p \not\vdash \forall \Gamma \vdash \{ p \} e \{ s \} \vdash v \]

Hence, the claim.

\[\square\]

Lemma 31 (Partial correctness of function-level, assume/guarantee reasoning). Let \( \text{ def } f^z := \vec{e} \in \{ \vec{e} \} e \{ s \} \in \text{ P}^z \). \forall \Gamma \in \text{ Env}^z_{\vec{e}, p, s} \text{ such that there exists no infinite sequence } (\Gamma, \vec{e}) \rightsquigarrow^* (\Gamma', \vec{e}') \rightsquigarrow^* \cdots, \exists \vec{e}, \Gamma' \vdash p \not\vdash \forall \Gamma' \vdash \vec{e} \not\vdash u.

Proof. Let \( \Gamma \in \text{ Env}^z_{\vec{e}, p, s} \). If there exists no infinite sequence \( (\Gamma, \vec{e}) \rightsquigarrow^* (\Gamma', \vec{e}') \rightsquigarrow^* \cdots \), then there exists a \( n \in \mathbb{N} \) such that \( \neg (\exists k > n, (\Gamma', \vec{e}) \rightsquigarrow^k (\Gamma', \vec{e})) \). We know that \( \Gamma \in \text{ Env}^z_{\vec{e}, p} \) implies that \( x \in \text{ dom}(\sigma) \). Hence, by Lemma 30, \exists \vec{e}, \Gamma' \vdash p \not\vdash \forall \Gamma' \vdash \vec{e} \not\vdash u.

\[\square\]

Lemma 32 (Soundness of function-level, assume/guarantee reasoning). Let \( \text{ def } f^z := \vec{e} \in \{ \vec{e} \} e \{ s \} \in \text{ P}^z \). If every function defined in \( \text{ P}^z \) terminates and satisfies the rules of function-level assume/guarantee reasoning, the contract of \( f^z \) holds i.e. \( \forall \Gamma \in \text{ Env}^z_{\vec{e}, p, s, u} \). \( \Gamma \vdash p \not\vdash \forall \Gamma' \vdash \vec{e} \not\vdash u. \)

Proof. The proof follows from Lemma 31 and the definition of termination of a function, which requires that the body of the function does not diverge.

\[\square\]
I. For each def f x := {pre} e {post}, |=p pre ⇒ post[e/res]
II. For each call site c ∈ DispCalls, |=p path(c) ⇒ pre(c)
III. (Cache monotonicity) For each π ∈ Props
    |=p(π(stz ≤ stz) & [π]p stz1) ⇒ [π]p stz2
IV. For each closure construction site c ∈ C, w1 in C in
    |=p path(c) ⇒ ([π]p st(c))
V. For each call site c = f ′(x, z, s, t) in DispCalls
    |=p(path(c) & [π]p[z0, y0] p st) ⇒ pre(c)

Similar to Lemma 30, now we prove a lemma that establishes that the above assume-guarantee rules are essentially a part of an induction reasoning. For the simplicity of the proof, we assume that ([π]p st(c)) is invoked just before the construction site c ∈ (C, w1), and that the result of π is ignored (including the state). That is, we replace (C, w1) by let := ([π]p st(c)) in (C, w1). It is obvious that this transformation is semantics preserving. But the benefit of this is that it simplifies the statement of the following Lemma, which now only talks about the named functions defined in the program.

Lemma 33. Let P be a program and P♯ the model program. If every function defined in P♯ terminate and the assume/guarantee assertions (I) to (V) defined above hold, the following property holds for all n ∈ \mathbb{N}

∀ program P', ∀Γ ∈ Env
(∃(def f x := e ∈ P, Γ♯ ∈ Env♯ s.t.
(T♯ e, path(Γ)) ⇒ Γ♯ \sim (Γ♯, σ♯(st))) ∧
\exists k > n, e', Γ', [Γ]p st(Γ') ⇒ (Γ', e') \∨
∀ def f x := e ∈ P, e = (p e {s}), Γ♯ ∈ Env♯
if (Γ♯ p, path(Γ') ⇒ (Γ', e') ∧ Γ♯ \sim (Γ♯', σ♯(st))) then
(∃v, Γ'♯ ⇒ [p]ρ st ∣v false ∧ Γ'♯ ⇒ [e]ρ st ∣v)

Proof. We prove this by induction on n. Intuitively, this limits the depth of evaluation of any expression in the model program P♯, during a run starting from the entry expression eentry. Let P′♯ be a client program. The base case is when n = 1. Consider a function definition def f x := e. Let e′ = [p]ρ st and let e′ = [p']ρ s {s}. Let (Γ♯ p, eentry) ⇒ (Γ', e′) and Γ'♯ \sim (Γ'♯, σ♯(st))

Now, if the evaluation of e′ under Γ'♯ has depth more than 1 then the claim trivially holds. Therefore, say the evaluation of e′ under Γ'♯ has depth at most 1. Hence, it cannot make any function calls. (Note that there are only direct calls in the model program.)

\textbf{Calls}(Γ'♯, e′) \cup \textbf{Calls}(Γ'♯, e′) = \emptyset

By 2 – 8, \exists v, Γ'♯ ⇒ [p]ρ st ∣v false ∧ Γ'♯ ⇒ [e]ρ st ∣v

Hence the claim holds in the base case.

Now, consider the inductive case and say the claim holds up to some number m. Now, if the evaluation of e′ under Γ'♯ has depth more than m then the claim trivially holds. Therefore, say the evaluation of e′ under Γ'♯ has depth at most m + 1.

(a) Say \neg e ∈ \textbf{Calls}(Γ', e′, Γ') ⇒ (c, Γ′) in this case, for all (c, ρ) ∈ Calls(Γ', e′, Γ'), p = path(c) ⇒ pre(c) holds. Hence, by 9 – 21, \exists Γ'♯ ⇒ [p]ρ st ∣v false ∧ Γ'♯ ⇒ [e]ρ st ∣v.

(b) Therefore, there exists an e = g♯(x, z, s, t) in DispCalls and Γ♯ ∈ Env♯ such that (e′, Γ') ⇒ (x, Γ′) and \neg (e′, Γ′) ⇒ (Appy, a, s′, t′) and Γ♯ \sim (Γ′, s′, t′) where σ♯(H′(y2)) = σ♯(H′(2)) = (c0 w). By the definition of \textbf{DispCalls} and the model translation, \langle e′, Γ'⟩ ⇒ (Appy, y, a, s′, t′) and Γ♯ \sim (Γ′, s′, t′) where σ♯(H′(y2)) = σ♯(H′(2)) = (c0 w). By Lemma 26, \langle e′, Γ'⟩ ⇒ (x, Γ′) and Γ’♯ \sim (Γ’♯, σ♯(st')).

Therefore, there exists an address a such that σ(y) = a, H♯(a) = ([λx.g(x, p), p → v]) and v ∼ w.

By definition of \textbf{DispCalls} the call (g y) is an encapsulated call. Therefore, (λx.g(x, p))p belongs to the program P i.e., l ∈ labelsP. Let "def cr x := {e0}" be the function in P that contains the lambda with label l. The closure ((λx.g(x, p))p)[p → v]) should have been created at some point during the run starting from eentry. Therefore, there exists a sequence \langle Γ♯ p, eentry⟩ ⇒ (\langle x, cr x⟩ ⇒ (Γ0, λx.g(x, p))⟩, Γ0, I, λx.g(x, p) ⊥ a, C0, H♯0[⟨⇒ (λx.g(x, p), p → v))⟩], σ0, H♯0 ⊆ H1 and C0 ⊆ C1.

Let Γ0 be such that Γ0 ∼ (Γ♯0, e♯0(st)) and let e♯0 = [e0]ρ st.

Subclaim: forall k ∈ N, ∀v ∈ Env♯ k, Γ♯ \sim (Γ'♯0, σ♯0(st)) if (Γ♯0, e0) ⇒k (Γ', e′) then (a) there exists an \langle Γ'♯0, e0⟩ ⇒k (Γ'♯, e′) and r > m + 1, or (b) 33s. (Γ'♯0, e0) ⇒k (Γ'♯, e′) and Γ'♯ ∼ (Γ'♯, S) and Γ'♯ ⇒ s ∼ S.

We now prove one of the claims of the lemma holds. In the ⇒ relations (shown in Fig. 10) introduced by all rules except LET and CONTRACT, the environments Γ and Γ♯ differ only by the store component. By the definition of the translation and the operational semantics it is easy to see that there exists an Γ0 such that (Γ♯0, e0) ⇒ (Γ♯, e′) and Γ ∼ (Γ♯0, S) and Γ'♯ ∼ (Γ'♯, S) ⇒ s ∼ S. Say (Γ', e′) ⇒ (Γ', e0).

Now consider the rule LET. Let e = let x := e1 in e2. By the definition of the translation: e′ = let x := e1ρ p st in e2ρ x 2. By definition, there are two ⇒ relations introduced by the rule. Consider the relations: (Γ', e′) ⇒ (Γ, e1) and (Γ', e′) ⇒ (Γ, e1) p st. These clearly satisfy the claim.

Consider the other relation defined as follows: If Γ ⇒ e1 ρ st u1, Γ1 then (Γ, e1) ⇒ (Γ1, e2) ρ st u1. We also have similar rule for the translated expression. If Γ ⇒ e1ρ st u1, Γ1 then (Γ', e′) ⇒ (Γ', e2) ρ st u2, Γ1 and (Γ', e′) ⇒ (Γ', e2) ρ st u2, Γ1. Now there are two cases to consider

(a) There exists a chain (Γ', e1ρ st) ⇒ (Γ', e2) ρ st and r > m. In this case, there exists an (Γ', e1ρ st) ⇒ (Γ', e′) and r > m + 1, since given (Γ', e1ρ st) is reachable from (Γ', e1ρ st). Hence the claim holds.

(b) There does not exist a chain (Γ', e1ρ st) ⇒ (Γ', e′) and r > m. Now, we claim that there exists a chain (Γ', e1ρ st) ⇒ (Γ', e′) and r > m. This is because, by Lemma 17, the evaluation could be undefined only if either the evaluation does not terminate or because there is an output violation during the evaluation. Thus, the former case is not possible since there does not exist a chain (Γ', e1ρ st) ⇒ (Γ', e′) and r > m. The latter case is not possible because every call (h g) encountered during the evaluation (under an environment Γ♯′) cannot have a chain (Γ♯′, (h g)) ⇒ (Γ', e′) and r > m (otherwise (Γ', e1ρ st) ⇒ (Γ', e2) and r > m, which contradicts the given fact). Therefore, by the (outer) induction hypothesis the call should produce a value. That is, there can be no contract violation.

We now have shown that Γ♯ ⇒ e1ρ st ∣v, Γ♯ ⇒ e1ρ st ∣v, Γ♯ ⇒ e1ρ st ∣v. Therefore, (Γ', e′) ⇒ (Γ♯0, e♯0) ρ x 2 is defined. By Lemma 26, Γ ⇒ eρ st u1, Γ1 is defined. Thus, (Γ, e) ⇒ (Γ, e2) is also defined and Γ'♯ ∼ (Γ'♯0, S) and Γ'♯ ⇒ x 2 st S hence, the claim holds. The rule contract can be similarly proven.

□
With this above established claim let us again revisit the evaluation: $(\Gamma_0, \alpha, e) \leadsto_{\text{ Env}} (\Gamma_0, \alpha, x, g(x, p))$, where $\Gamma_0 = \lambda x.g(x, p) \downarrow a, \alpha = 0 \in \epsilon$. By definition of $\text{Env}$, we have $\alpha = 0 \in \epsilon$. Due to the above subclaim, we know that one of the following cases holds: (a) either there exists $(\Gamma_0, \alpha, e) \leadsto_{\text{ Env}} (\Gamma_0, \alpha, x, g(x, p))$, and $\exists s. \Gamma_0 \vdash (\Gamma_0, S)$ and $\Gamma_0 \vdash s \vdash S$. In the former case the lemma holds as the first disjunkt of the lemma is satisfied as $\Gamma_0$ belongs to $\text{Env}_\alpha^{\text{Fr}}$. Therefore consider the latter case.

Let $cc = (\Gamma_1 p)$. By definition, $st(cc)$ is the state expression reaching the construction site $cc$. Therefore $s = st(cc)$. By the definition of $\text{path}$, for any function definition $\text{def} f \ x := e_i$ and closure construction site $cc$, we have

$$\forall i^\sigma \in \text{Env}^\sigma. (\Gamma_i, e_i) \leadsto_{\text{ Env}} (\Gamma_i, st(cc)) \Rightarrow \Gamma_i \vdash \text{path}(cc) \Downarrow \text{true}$$

Therefore, $\Gamma_i \vdash \text{path}(cc) \Downarrow \text{true}$

$$\Rightarrow \forall i^\sigma \in \text{Env}^\sigma. (\Gamma_i, e_i) \leadsto_{\text{ Env}} (\Gamma_i, st(cc)) \Rightarrow \Gamma_i \vdash \text{path}(cc) \Downarrow \text{false}$$

Now recall that we have assumed that $\rho'$ is invoked just before the closure construction $cc$. Therefore, $\exists i^\sigma \in \text{Env}^\sigma$ such that $\exists \nu. \exists \sigma \ni \nu \vdash \Gamma_i \vdash \rho' \Downarrow \nu$, since we are given that $(\Gamma_i, e_i) \leadsto_{\text{ Env}} (\Gamma_i, st(cc))$. Hence, $\forall \nu \vdash \Gamma_i \vdash \rho' \Downarrow \nu$. It is easy to see that, $\text{Calls}(\Gamma_i, \rho') = \text{Calls}(\Gamma_i, \rho)$ since $\Gamma_i \subseteq \Gamma_i'$. (Note that the model program does not have memoization and is purely functional.) Therefore, $\forall \nu \vdash \Gamma_i \vdash \rho' \Downarrow \nu$. By Lemma 26, $\Gamma_i \vdash \rho_i \Downarrow \nu$. Now, we know that $\mathcal{H}_0 \subseteq \mathcal{H}_i$ and $c_0 \subseteq C_1$. We are also given by the assume-guarantee rule (I) that

$$\Rightarrow \forall \nu \vdash \Gamma_i \vdash \rho_i \Downarrow \nu$$

By the reasoning shown in 16–18 and 37

Therefore, for every $(\Gamma', e') \in \text{DispCalls} \cap \text{Calls}(\Gamma', e')$, $\Rightarrow \Gamma_i \vdash \text{pre}(c) \Downarrow \nu$. By the reasoning shown by 18 and 37, for every $(\Gamma', e') \in \text{Calls}(\Gamma', e') \setminus \text{DispCalls}$, $\Rightarrow \Gamma_i \vdash \text{pre}(c) \Downarrow \nu$. Therefore, as shown by 19–21, $\forall \nu \vdash \Gamma_i \vdash \rho_i \Downarrow \nu$. Hence, by the assume-guarantee assertion (I), $\exists i^\sigma \in \text{Env}^\sigma. (\Gamma_i, e) \vdash \rho_i \Downarrow \nu$. "

Theorem 3.(Soundness of creation-distribution reasoning) Let $P$ be a program and $P^\sigma$ the model program. Let $\text{def} f^\sigma x := e$ where $e \in \{p \in \epsilon | s\}$ be a function definition in $P^\sigma$. If every function defined in $P$ terminates and the assume/guarantee assertions $I$ to $N$ defined above hold, the contracts of $f^\sigma$ holds i.e., $\forall i^\sigma \in \text{Env}^\sigma. (\Gamma_i, e) \vdash \rho_i \Downarrow \nu$. "

Proof. Let $\text{def} \ f^\sigma x := e'$. By the reasoning shown in $P^\sigma$, we have $e' \in \{p \in \epsilon | s\}$. By definition, there exists a $\Gamma \in \text{Env}$ and a program $P'$ such that $(\Gamma_i, e) \leadsto_{\text{ Env}} (\Gamma_i, e)$ and $\Gamma \sim (\Gamma_i, e')$. Also, by definition of $\text{Env}_\alpha^{\text{Fr}}$, $\Gamma_i \vdash (\Gamma_i, e')$. A terminating evaluation given that all functions in the program $P$ are terminating. Now say there exists an infinite chain of the form $(\Gamma_i, e) \leadsto_{\text{ Env}} (\Gamma_i, e_1) \leadsto_{\text{ Env}} \cdots$. This is a contradiction to the fact that the evaluation $(\Gamma', e) \leadsto_{\text{ Env}} (\Gamma', e_1) \leadsto_{\text{ Env}} \cdots$. Hence, the evaluation of $(\Gamma', e) \leadsto_{\text{ Env}} (\Gamma', e_1) \leadsto_{\text{ Env}} \cdots$ is terminating.
that \((\Gamma^P[x_1], e_{entry}) \Rightarrow^* (\Gamma_h, \bar{e}_h)\) and \(\Gamma_h \sim (\Gamma^P_h, e_{exit})\) and \((\Gamma^F_h, [\bar{e}_h]_P \ st)\Rightarrow^* \cdots\) is infinite. Thus, there exists a \(n \in \mathbb{N}\) such that \(\neg \exists \text{def } h x := \bar{e}_h \in P, \Gamma^P_h \in \text{Env}^\forall\text{ s.t. } (\Gamma^P[x_1], e_{entry}) \Rightarrow^* (\Gamma_h, \bar{e}) \land \Gamma \sim (\Gamma^P_h, e_{exit})\) \(\land \exists k > n, e', \Gamma', (\Gamma^F_h, [\bar{e}_h]_P \ st)\Rightarrow^k (\Gamma', e')\). Hence, by Lemma 33, \(\exists u. \, \Gamma^d \vdash p \not\exists \bar{v} \Gamma^d \vdash \bar{e} \not\exists \bar{u} \Gamma\) for every function definition in \(P^d\). Hence, the contracts of the function \(f^d\) also holds. 

\section{Decidability of Inference Algorithm}

\begin{theorem}
Given a linear parametric formula \(\phi(x, \bar{a})\) with free variables \(\bar{x}\) and \(\bar{a}\), belonging to a theory \(\mathcal{T}\) that is a combination of quantifier-free theories of uninterpreted functions, algebraic datatypes, and sets, and either integer linear arithmetic or real arithmetic, finding a assignment \(\sigma\) such that \(\text{dom}(\sigma) = |\bar{a}|\), and \((\phi \sigma)\) is \(\mathcal{T}\)-unsatisfiable is decidable.
\end{theorem}

\textbf{Proof Sketch.} We express the problem as trying to decide the validity of a formula of the form: \(\exists \bar{a}. \forall x'. (\forall f, \phi'(x', f, \bar{a})) \land (\forall s, \phi_{exit}(x', \bar{s}))\), where, \(f, \phi\) are the uninterpreted function symbols in \(\phi, \bar{s}\) are variables of set sort, \(x'\) are variables of other sorts, and \(\phi_{exit}\) is a formula in \(\mathcal{T}_{exit}\) that has only set operations. This is possible because the existentially quantified variables \(\bar{a}\) are only numerical variables. Since the theory of sets admit decidable quantifier elimination [45], the above formula could be reduced to an equivalent formula of the form \(\exists \bar{a}. \forall x', \bar{f}, \phi''(x', \bar{a})\), which can decided using the algorithm presented in [52], and depicted in Fig. 7. 

\section{Extended Specification Constructs}

Our implementation support a few other specification constructs beyond those presented in Fig. 3 to enable easier specification. As mentioned in section 2, we support a construct \(\text{in}(e, x)\) that evaluates an expression in a cache-state given by \(x\), and a construct \(\text{inSt}\) to access the state of the cache at the beginning of a function in the postcondition of the function. Analogously, we also support a construct \(\text{outSt}\) to refer to the state of the cache at the end of the function in the postcondition. In the construct \(\text{in}(e, x)\) the variable \(x\) has a cache. The constructs \(\text{inSt}\) and \(\text{outSt}\) are the only expressions that have these type. Therefore, even though the construct \(\text{in}(e, x)\) allows evaluating an expression under a cache given by \(x\) the cache can only be obtained either through \(\text{inSt}\) or \(\text{outSt}\) expressions. Essentially, \(\text{in}(e, x)\) is used to evaluate an expression under a cache encountered previously during the evaluation. Fig. 6 shows the translation of these expressions during the model program generation.

To define the semantics of these constructs, we modify the domain of values \(\text{Val}\) to also include a cache. That is, \(\text{Cache} \subseteq \text{Val}\). Fig. 12 shows their semantics with respect to the modified domain, and redefines the semantics of the contracts in the presence of these constructs. Besides these, we also introduce two constructs: \(\text{fmtch}\) and \(*\) explained below. The construct \(e\) computes the result of an expression \(e\) without caching the result of \(e\) for reuse. This is a side-effect-free operation that is to be used in places where only the result of the expression is relevant. We support a construct \(\text{fmtch}\) of the form: \(x \ \text{fmtch}\{\lambda x_1, f_1 (x_1, y_1) \Rightarrow e_1\}^{n_1}_{i=1} p \ st = \ x \ \text{match}\{C_{i_1}, y_1 \Rightarrow [e_i]_P \ st\}^{n_1}_{i=1}, \) where \(l_i\) is the label of \(\lambda x_1, f_1 (x_1, y_1)/\bar{s}, p\) \([e^*]_P \ st = ([e]_P \ st)_{i=1}\).

The soundness and completeness theorems, and other Lemmas presented in the previous section translate to programs with these additional specification constructs.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure12.png}
\caption{Semantics of the specification constructs \text{fmtch}, \text{inSt}, \text{outSt} and \(*\).}
\end{figure}