Verifying Resource Bounds of Programs with Lazy Evaluation and Memoization

EPFL-REPORT-215783

Ravichandhran Madhavan
EPFL, Switzerland
ravi.kandhadai@epfl.ch

Sumith Kulal
IIT Bombay, India
sumith@cse.iitb.ac.in

Viktor Kuncak
EPFL, Switzerland
viktor.kuncak@epfl.ch

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 $\mathsf{steps} \leq ? * \mathsf{size}(\mathsf{l}) + ?$ 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 data-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.

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 between the number of steps performed at runtime and the actual wall-clock execution time for our benchmarks. In particular, for a lazy, bottom-up merge sort implementation [4] one step of evaluation at runtime corresponded to 2.35 nanoseconds (ns) on average with an absolute deviation of 0.01 ns, and for a real-time queue data structure implementation [59] it corresponded to 12.25 ns with an absolute deviation of 0.03 ns. These results further add to the importance of proving bounds even if they are with respect to the abstract resource metrics.

In this paper, we propose a system for specifying and verifying abstract resource bounds, such as steps and alloc, of programs written in a pure subset of Scala [57] with added support for memoization and new specification constructs. In our approach, users can specify the desired resource bound as templates with numerical holes e.g. as $\mathsf{steps} \leq ? * \mathsf{size}(\mathsf{l}) + ?$ in the contracts of functions along with other invariants necessary for proving the bounds. Our system proves the bound by automatically inferring values for the holes that will make the bound hold for all executions

```
1  private case class SCons(x: (BigInt,Bool), tfun:() ⇒ SCons) {
2    lazy val tail = tfun()
3  }
4  private val primes = SCons((1, true), () ⇒ nextElem(2))
5
6  def nextElem(i: BigInt): SCons = {
7    require(i ≥ 2)
8    val x = (i, isPrimeNum(i))
9    val y = i + 1
10   SCons(x, () ⇒ nextElem(y))
11 } ensuring(r ⇒ steps ≤ ? ∗ i + ?)
12
13 def isPrimeNum(n: BigInt): Bool = {
14   def rec(i: BigInt): Bool = {
15     require(i ≥ 1 && i < n)
16     if (i == 1) true else (n % i != 0) && rec(i − 1)
17   } ensuring (r ⇒ steps <= ? ∗ i + ?)
18   rec(n − 1)
19 } ensuring(r ⇒ steps ≤ ? ∗ n + ?)
20
21 def isPrimeStream(s: SCons, i: BigInt): Bool = {
22  require(i ≥ 2)
23  s.tfun ≈ (() ⇒ nextElem(i)) }
24
25 def takePrimes(i: BigInt, n: BigInt, s: SCons): List = {
26  require(0 ≤ i && i ≤ n && isPrimeStream(s, i+2))
27  if(i < n) {
28    val t = takePrimes(i+1, n, s.tail)
29    if(s.x._2) Cons(s.x._1, t) else t
30  } else Nil()
31 } ensuring(r ⇒ steps ≤ ? ∗ (n(n−i)) + ?)
32
33 def primesUntil(n: BigInt): List = {
34   require(n ≥ 2)
35   takePrimes(0, n−2, primes)
36 } ensuring(r ⇒ steps ≤ ? ∗ n^2 + ?)
```

Figure 1. Prime numbers until n using an infinite stream.

```
1  def concrUntil(s: SCons, i: BigInt): Bool =
2    if(i > 0) cached(s.tail) && concrUntil(s.tail, i−1)
3    else true
4
5  def primesUntil(n: BigInt): List = {
6    // see Fig. 1 for the code of the body
7  } ensuring{r ⇒ concrUntil(primes, n−2) &&
8    (if(concrUntil(primes, n−2) in inSt)
9      steps ≤ ? ∗ n + ?
10    else steps ≤ ? ∗ n^2 + ?) }
```

Figure 2. Specifying properties dependent on memoization state.

of the function. For instance, our system was able to infer that the number of steps spent in accessing the k^{th} element of an unsorted list l using a lazy, bottom-up merge sort algorithm [4] is bounded by $36(k \cdot \lfloor \log{(l.size)} \rfloor) + 53l.size + 22$. We empirically compared the number of steps used by this program at runtime against the bound inferred by our tool by varying the size of the list l from 10 to 10K and k from 1 to 100. Our results showed that the inferred values were 90% accurate for this example (section 5 presents more results). We now present an overview of how programs can be specified and verified in our system using the pedagogical example shown in Fig. 1 that creates an infinite stream of prime numbers.

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 $\approx$, 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 iff 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 $\leq 16n^2 + 28$.

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 takePrimes will take constant time as it has been cached during the previous call to 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 properties we support a built-in operation 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 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.tail is similar to calling a memoized function tail(s).) This property holds for primes stream at the end of a call to primesUntil(n), and hence is stated in the postcondition of 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 "inSt" that can be used in the postcondition to refer to the state at the beginning of the function, and an "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 transformations, 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 [78], but other termination algorithms for higher-order programs [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 termination checks from resource verification enables checking termination 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 enables 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 verification (sections 3 and 4).
- We use our system to prove asymptotically precise resource bounds of 17 benchmarks, expressed in an functional subset of Scala [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 runtime 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 language extended with memoization, contracts and specification constructs, that we will use to formalize our approach. Every expression has a static label belonging to $Labels$ (omitted in Fig. 3). We use e^l to denote an expression with its label. To reduce clutter, we omit the label if it is not relevant to the context. $Tdef$ shows the syntax of user-defined algebraic datatypes and $Fdef$ shows the syntax of function definitions. A program P is a set of functions defini-

$$
\begin{array}{lll}
x, y \in \mathit{Vars}, \bar{x} \in \mathit{Vars}^*, c \in \mathit{Cst} & & \textit{(Variables \& Constants)} \\
a \in \mathit{TVars} & & \textit{(Template Variables)} \\
f \in \mathit{Fids}, C_i \in \mathit{Cids}, i \in \mathbb{N} & & \textit{(Function \& Constructor ids)} \\
\mathit{Tdef} & ::= & \mathsf{type}\ d := (C_1\ \bar{\tau}, \cdots, C_n\ \bar{\tau}) \\
\tau \in \mathit{Type} & ::= & \mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \tau \Rightarrow \tau \mid d \\
\mathit{Blk}_\alpha & ::= & \mathsf{let}\ x := e_\alpha\ \mathsf{in}\ e_\alpha \mid x\ \mathsf{match}\{(C\ \bar{x} \Rightarrow e_\alpha;)^+\} \\
pr \in \mathit{Prim} & ::= & + \mid - \mid * \mid \cdots \mid \wedge \mid \neg \\
e_s \in E_{src} & ::= & x \mid c \mid pr\ x \mid x\ \mathsf{eq}\ y \mid f\ x \mid C\ \bar{x} \mid e_\lambda \mid x\ y \mid \mathit{Blk}_s \\
e_\lambda \in \mathit{Lam} & ::= & \lambda\ x.f\ (x, y) \\
e_p \in E_{spec} & ::= & e_s \mid \mathit{Blk}_p \mid \mathsf{cached}(f\ x) \mid \mathsf{inSt} \mid \mathsf{in}(e_p, x) \mid \mathsf{res} \\
& \mid & \mathsf{steps} \le ub \mid \mathsf{alloc} \le ub \\
ub \in \mathit{Bnd} & ::= & e_p \mid e_t \\
e_t \in E_{tmp} & ::= & a \cdot x + e_t \mid a \\
\mathit{Fdef} & ::= & (\texttt{@memoize})?\ \mathsf{def}\ f\ x := \{e_p\}\ e_s\ \{e_p\}
\end{array}
$$

Figure 3. Syntax of types, expressions, functions, and programs. The rule Blk_α is parametrized by the subscript α.

tions in which every function identifier is unique, every direct call invokes a function defined in the program, and the labels of expressions are unique. As a syntactic sugar, we consider tuples as a special datatype, and denote tuple construction using $(x_1, \cdots, x_n)$, and selecting the i^{th} element of a tuple using $x._i$.

In particular, our language supports a structural equality operator eq, direct calls to named functions: $f\ x$, and indirect calls or lambda applications: $x\ y$. We also define an if-else operation if $cond\ e_1$ else e_2 that is similar to a match construct with two cases. The annotation @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 implemented using lambdas with unit parameter and memoized functions. Expressions that are bodies of functions can have contracts (or specifications). Such expressions have the form $\{e_1\}\ e\ \{e_2\}$ 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 using the variable res, and to the resource usage of e using steps and alloc. Users can specify upper bounds on resources as templates $e_t \in E_{tmp}$ with holes. The holes always appear as coefficients of variables defined in the program, which could be bound to more complex expressions through let binders.

For ease of formalization we enforce the following syntactic restrictions without reducing generality. All expressions except lambda terms are in *A-normal form* i.e, the arguments of all operations/functions are variables. All lambdas are of the form: $\lambda x.f\ (x, y)$ 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 $\bar{a} \in A^*$ 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 $A \mapsto B$ to denote a partial function from A to B. Given a partial function h, $\hat{h}(\bar{x})$ 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 $h[\bar{a} \mapsto \bar{b}]$ to denote $h[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. We omit h in the above notation if h is an empty function. We define a partial function $h_1 \uplus h_2$ as $(h_1 \uplus h_2)(x) =$ if $(x \in dom(h_2))\ h_2(x)$ else $h_1(x)$. Let $labels_P$ denote the set of labels of all expressions in a program

For steps: $c_{miss} = 2$, $c_{match(i)} = i + 1$, $c_{var} = c_{let} = 0$, for every other operation op: $c_{op} = 1$. $\oplus = +$
For alloc: $c_{cons} = c_{\lambda} = c_{miss} = 1$, for every other operation op: $c_{op} = 0$. $\oplus = +$

$$\textsc{Cst}\ \frac{c \in Cst}{\Gamma \vdash c \Downarrow_{c_{cst}} c, \Gamma} \qquad \textsc{Var}\ \frac{x \in Vars}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash x \Downarrow_{c_{var}} \sigma(x), \Gamma} \qquad \textsc{Prim}\ \frac{pr \in Prim}{\Gamma \vdash pr\ x \Downarrow_{c_{pr}} pr(\sigma(x)), \Gamma}$$

$$\textsc{Equal}\ \frac{v = \sigma(x) \underset{\mathcal{H}}{\approx} \sigma(y)}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash x \text{ eq } y \Downarrow_{c_{eq}} v, \Gamma}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e_1 \Downarrow_p v_1, (\mathcal{C}', \mathcal{H}', \sigma') \qquad (\mathcal{C}', \mathcal{H}', \sigma[x \mapsto v_1]) \vdash e_2 \Downarrow_q v_2, (\mathcal{C}'', \mathcal{H}'', \sigma'')}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash \text{let } x := e_1 \text{ in } e_2 \Downarrow_{c_{let} \oplus p \oplus q} v_2, (\mathcal{C}'', \mathcal{H}'', \sigma)}$$

$$\textsc{Cons}\ \frac{a = fresh(\mathcal{H}) \qquad \mathcal{H}' = \mathcal{H}[a \mapsto (cons\ \hat{\sigma}(\bar{x}))]}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash cons\ \bar{x} \Downarrow_{c_{cons}} a, (\mathcal{C}, \mathcal{H}', \sigma)}$$

$$\textsc{Match}\ \frac{\mathcal{H}(\sigma(x)) = C_i\ \bar{v} \qquad (\mathcal{C}, \mathcal{H}, \sigma[\bar{x_i} \mapsto \bar{v}]) \vdash e_i \Downarrow_q v, (\mathcal{C}', \mathcal{H}', \sigma')}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash x \text{ match } \{C_i\ \bar{x_i} \Rightarrow e_i)\}_{i=1}^{n} \Downarrow_{c_{match(i)} \oplus q} v, (\mathcal{C}', \mathcal{H}', \sigma)}$$

$$\textsc{ConcreteCall}\ \frac{(\mathcal{C}, \mathcal{H}, \sigma[param_{\Gamma}(f) \mapsto u] \vdash body_{\Gamma}(f) \Downarrow_p v, (\mathcal{C}', \mathcal{H}', \sigma')}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash f\ u \Downarrow_p v, (\mathcal{C}', \mathcal{H}', \sigma)}$$

$$\textsc{Lambda}\ \frac{a = fresh(\mathcal{H}) \qquad clo = (\lambda x.f\ (x, y), [y \mapsto \sigma(y)])}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash \lambda x.f\ (x, y) \Downarrow_{C_{\lambda}} a, (\mathcal{C}, \mathcal{H}[a \mapsto clo], \sigma)}$$

$$\textsc{NonMemoizedCall}\ \frac{f \in Fids \qquad f \notin Mem_{\Gamma} \qquad \Gamma \vdash (f\ \sigma(x)) \Downarrow_p v, \Gamma'}{\Gamma \vdash f\ x \Downarrow_{c_{call} \oplus p} v, \Gamma'}$$

$$\textsc{IndirectCall}\ \frac{\mathcal{H}(\sigma(x)) = (\lambda z.e, \sigma') \qquad (\mathcal{C}, \mathcal{H}, (\sigma \uplus \sigma')[z \mapsto \sigma(y)]) \vdash e \Downarrow_p v, (\mathcal{C}', \mathcal{H}', \sigma')}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash x\ y \Downarrow_{c_{app} \oplus p} v, (\mathcal{C}', \mathcal{H}', \sigma)}$$

$$\textsc{MemoCallHit}\ \frac{f \in Mem_{\Gamma} \qquad ((f\ \sigma(x)), v) \in_{\mathcal{H}} \mathcal{C}}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash f\ x \Downarrow_{c_{hit}} v, \Gamma}$$

$$\textsc{MemoCallMiss}\ \frac{f \in Mem_{\Gamma} \qquad u = \sigma(x) \qquad \neg\big((f\ u) \in_{\mathcal{H}} dom(\mathcal{C})\big) \qquad \Gamma \vdash (f\ u) \Downarrow_p v, (\mathcal{C}', \mathcal{H}', \sigma')}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash f\ x \Downarrow_{c_{miss} \oplus c_{call} \oplus p} v, \mathcal{C}'[(f\ u) \mapsto v], \mathcal{H}', \sigma}$$

$$\textsc{Cached}\ \frac{v \Leftrightarrow \big((f\ \sigma(x)) \in_{\mathcal{H}} dom(\mathcal{C})\big)}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash \text{cached}(f\ x) \Downarrow_0 v, \Gamma}$$

$$\textsc{Contract}\ \frac{\Gamma \vdash pre \Downarrow_p true, \Gamma_1 \qquad \Gamma \vdash e \Downarrow_q v, \Gamma_2 : (\mathcal{C}_2, \mathcal{H}_2, \sigma_2) \qquad (\mathcal{C}_2, \mathcal{H}_2, \sigma_2[R \mapsto q, \text{res} \mapsto v]) \vdash post \Downarrow_r true, \Gamma_3}{\Gamma \vdash \{pre\}\ e\ \{post\} \Downarrow_q v, \Gamma_2}\ R \in \{\text{steps}, \text{alloc}\}$$

Figure 4. Resource annotated operational semantics for the concrete expressions of the language defined in Fig. 3.

P. Let $type_P(e)$ denote the type of an expression e in a program P. Given a lambda e_λ, we use $FV(e_\lambda)$ to denote the free variable captured by e_λ and $target(e_\lambda)$ to denote the function called in the body of the lambda. The operation $e[e'/x]$ 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 $\iota : TVars \mapsto \mathbb{Z}$, we use $e\ \iota$ 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 Lauchbury'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 Adr denote the addresses of heap-allocated structures namely closures and datatypes. The state of an interpreter evaluating expressions of our language is a quadruple consisting of a cache $\mathcal{C}$, a heap $\mathcal{H}$, an assignment of variables to values σ, and a set of function definitions, defined as follows:

$u, v \in Val = \mathbb{Z} \cup Bool \cup Adr$
$FVal = Fids \times Val \qquad DVal = Cids \times Val^*$
$Clo = Lam \times Store \qquad \mathcal{H} \in Heap = Adr \mapsto (DVal \cup Clo)$
$\sigma \in Store = Vars \mapsto Val \qquad \mathcal{C} \in Cache = FVal \mapsto Val$
$\Gamma \in Env \subseteq Cache \times Heap \times Store \times 2^{Fdef}$

We define a few helper functions on the semantic domains. Let $fresh(\mathcal{H})$ denote an element $a \in (Adr \setminus dom(\mathcal{H}))$. Let $body_\Gamma(f)$ and $param_\Gamma(f)$ denote the body and parameter of a function f defined in the environment Γ, and $Mem_\Gamma \subseteq Fids$ denote the set of memoized functions in the function definitions in Γ.

Structural equivalence. We define a structural equivalence relation $\underset{\mathcal{H}}{\approx}$ on the values Val with respect to a $\mathcal{H} \in Heap$, 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 $\lambda w.f\ (w, z)$ and the captured variables y and z are bound to structurally equivalent values. Formally, (subscript omitted below for clarity)

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

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 $\Gamma \in Env$, an expression e evaluates to a value $v \in Val$ and results in a new environment $\Gamma' \in Env$, while consuming $p \in \mathbb{Z}$ units of a resource. When necessary we expand Γ 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 parametrization. We parametrize the operational semantics in a way that it can be instantiated on multiple resources using the following parametrization functions: (a) A cost function c_{op} that returns the resource requirement of an operation op such as $cons$ or app. c_{op} may possibly have parameters. In particular, we use $c_{match(i)}$ 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 $\oplus : \mathbb{Z}^* \to \mathbb{Z}$ that computes the resource usage of an expression by combining the resource usages 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_{let} and c_{var} are zero as the operations are normally optimized away or subsumed by a machine instruction. c_{op} is 1 for every other operation except c_{miss} and $c_{match(i)}$. We consider datatype construction and primitive operations on big integers as unitary steps. We define $c_{match(i)}$ proportional to i as we need to include the cost of failing all the $i-1$ match cases. In the case of alloc, c_{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: MEMOCALLHIT and MEMOCALLMISS. 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* $\in_{\mathcal{H}}$ that uses structural equivalence to lookup the cache as follows: $(f\ u) \in_{\mathcal{H}} dom(\mathcal{C}) = \exists u' \in Val.(f\ u') \in dom(\mathcal{C}) \wedge u' \underset{\mathcal{H}}{\approx} u$. We parametrize the cost of searching and updating the cache using the parameters c_{hit} and c_{miss}. To calculate the steps resource, we consider lookup and update as unitary steps, and hence define $c_{miss} = 2$ (as it involves a lookup and an update operation) and $c_{hit} = 1$. In general, c_{miss} and c_{hit} may depend on the values of the arguments.

Specifications. The construct cached$(f\ x)$ evaluates to true in an environment Γ iff the call f is cached for the value of x in Γ. 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 $\{pre\}\ e\ \{post\}$. The expression evaluates to a value v only if pre holds in the input environment and $post$ 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 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 fmatch 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 $A_1 \cdots A_n$ 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 $1 \leq i \leq n$.

$$\frac{A_1 \cdots A_m \qquad B_1 \cdots B_{i-1}}{\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_i, e_i\rangle}$$

Let $\rightsquigarrow^*$ represent the reflexive, transitive closure of $\rightsquigarrow$. We say that an environment Γ' reaches e' during the evaluation of e from Γ iff $\langle \Gamma, e\rangle \rightsquigarrow^* \langle \Gamma', e'\rangle$. We say that the evaluation of e under Γ *diverges* iff there exists an infinite sequence $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_1, e_1\rangle \rightsquigarrow \cdots$. We say an expression e (or a function f) *terminates* iff there does not exist a $\Gamma \in Env$ under which e (or $body_\Gamma(f)$) 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 $P_c = P' \| P$ denote a closed program obtained by composing a client P' with an open program P. The evaluation of a closed program P_c starts from a distinguished entry expression e_{entry} (such as a call to the main function) under an initial environment $\Gamma_{P_c} : (\emptyset, \emptyset, \emptyset, F)$ where F is the set of function definitions in the program P_c. We define the valid environments of an expression e belonging to an open program P, denoted $Env_{e,P}$, as $\{\Gamma \mid \exists P'. \langle \Gamma_{P' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, e\rangle\}$.

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 def $f\ x := \tilde{e}$, where $\tilde{e} = \{pre\}\ e\ \{post\}$, whether in every valid environment that reaches $\tilde{e}$ in which pre does not evaluate to $false$, e evaluates to a value. Formally, $\forall \Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env_{\tilde{e},P}$. $\exists v.\ (\Gamma \vdash pre \Downarrow false) \vee \Gamma \vdash \tilde{e} \Downarrow v$. (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 ι 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 $e\ \iota$ denotes substituting the holes in an expressions e with the assignment given by ι. The resource bound inference problem is to find an assignment ι such that for every function def $f\ x := \{pre\}\ e\ \{post\}$ where $post$ may contain holes, $\forall \Gamma \in Env.\ (\Gamma \vdash pre \Downarrow false) \vee \Gamma \vdash \{pre\}\ e\ \{post\ \iota\} \Downarrow v$.

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 $c = x\ y$ belonging to a program P is an *encapsulated call* iff in every environment $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env_{c,P}$, if $\mathcal{H}(\sigma(x))$ is a closure $({e_\lambda}^l, \sigma')$, $l \in labels_P$. 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 () ⇒ 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 tfun() 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, $E_{spec} = E_{src}$). 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 Val and also sets of values of the source language ($Set = 2^{Val}$). The environments of the model do not have the cache component, i.e, $\Gamma \in Env^\sharp = Heap \times Store \times 2^{Fdef}$.

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 concrUntil, 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.

Expression Translation

$$\llbracket x \rrbracket_P\ st = (x, st, c_{var})$$
$$\llbracket pr\ x \rrbracket_P\ st = (pr\ x, st, c_{pr}) \quad \text{if } pr \in Prim$$
$$\llbracket x \text{ eq } y \rrbracket_P\ st = (x \text{ eq } y, st, c_{eq})$$
$$\llbracket C\ \bar{x} \rrbracket_P\ st = (C\ \bar{x}, st, c_{cons}) \quad \text{if } C \in Cids$$
$$\llbracket \text{let } x := e_1 \text{ in } e_2 \rrbracket_P\ st =$$
$$\quad \text{let } u := \llbracket e_1 \rrbracket_P\ st \text{ in}$$
$$\quad \text{let } w := \llbracket e_2[u._1/x] \rrbracket_P\ u._2 \text{ in } (w._1, w._2, c_{let} \oplus u._3 \oplus w._3)$$
$$\llbracket x \text{ match}\{C_i\ \bar{x}_i \Rightarrow e_i\}_{i=1}^n \rrbracket_P\ st = x \text{ match } \{$$
$$\big(C_i\ \bar{x}_i \Rightarrow \text{let } u := \llbracket e_i \rrbracket_P\ st \text{ in } (u._1, u._2, c_{match(i)} \oplus u._3)\big)_{i=1}^n\}$$

Call and Lambda Translation

$$\llbracket f\ x \rrbracket_P\ st = \qquad \text{if } f \text{ does not have @memoize annotation}$$
$$\quad \text{let } w := f^\sharp\ (x, st) \text{ in } (w._1, w._2, c_{call} \oplus w._3)$$
$$\llbracket f\ x \rrbracket_P\ st = \qquad \text{if } f \text{ has @memoize annotation}$$
$$\text{let } w := f^\sharp\ (x, st) \text{ in}$$
$$\text{let } x_{cost} = \text{if } (C_f\ x) \in st\ c_{hit} \text{ else } c_{miss} \oplus c_{call} \oplus w._3$$
$$\quad \text{in } (w._1, w._2 \cup \{(C_f\ x)\}, x_{cost})$$
$$\llbracket e_\lambda \rrbracket_P\ st = (C_l\ FV(e_\lambda), st, c_\lambda) \quad \text{if } e_\lambda/_{\cong,P} \text{ has label } l$$
$$\llbracket (x\ y)^l \rrbracket_P\ st =$$
$$\quad \text{let } w := App_l\ (x, y, st) \text{ in } (w._1, w._2, c_{app} \oplus w._3)$$

Specification Construct Translation

$$\llbracket \text{cached}(f\ x) \rrbracket_P\ st = ((C_f\ x) \in st, st, 0)$$
$$\llbracket \text{in}(e, x) \rrbracket_P\ st = \llbracket e \rrbracket_P\ x$$

Contract Translation

$$\llbracket \{pre\}\ e\ \{post\} \rrbracket_P\ st = \qquad \text{if } R \in \{\text{steps}, \text{alloc}\}$$
$$\quad \{(\llbracket pre \rrbracket_P\ st)._1\}$$
$$\qquad \llbracket e \rrbracket_P\ st$$
$$\quad \{\text{let } y = \llbracket post[\text{res}._1/\text{res}][st/\text{inSt}][\text{res}._3/R] \rrbracket_P\ \text{res}._2 \text{ in } y._1\}$$

Function Definition Translation

$$\llbracket \text{def } f\ x := e \rrbracket_P = \text{def } f^\sharp\ (x, st) := \llbracket e \rrbracket_P\ st$$

Dispatch Functions

For every indirect call $(x\ y)^l$ in P where $type_P(x) = \tau$,

$$\text{def } App_l\ (cl, w, st) :=$$
$$\quad cl \text{ match}\{\ C_{l_1}\ y_1 \Rightarrow \llbracket e'_1 \rrbracket_P\ st; \cdots; C_{l_n}\ y_n \Rightarrow \llbracket e'_n \rrbracket_P\ st$$
$$\qquad C_\tau\ y \Rightarrow \text{ error }\}$$

where, $\forall i \in [1, n]$. C_{l_i} are constructors of d_τ,
$(\lambda a_i.e_i)^{l_i}$ is a lambda in P and $e'_i = e_i[y_i/z_i][w/a_i]$

Figure 5. Resource and cache-state instrumentation.

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 ${e_\lambda}' = \lambda x.f'\ (x, z)$ are compatible, and denote it as $e_\lambda \cong {e_\lambda}'$, iff they invoke the same targets i.e, $f = f'$. This relation is interesting because during any evaluation two closures could be structurally equivalent iff their lambdas are compatible i.e, $e_\lambda \cong {e_\lambda}'$ iff $\exists \mathcal{H}, \sigma, \sigma'$ s.t. $(e_\lambda, \sigma) \underset{\mathcal{H}}{\approx} ({e_\lambda}', \sigma')$. In the generated model we ensure that the closures of lambdas that are compatible are represented using the same datatype. For each lambda e_λ, we define a representative denoted $e_\lambda/_{\cong,P}$ of the equivalence class with respect to $\cong$ that belongs to a program P. (It is undefined if P does not have a compatible lambda.) For each function type $\tau = A \Rightarrow B$ used in P, we add a datatype d_τ to the model (defined shortly), and replace every use of τ in the input program by the datatype d_τ.

Let $\{e_{\lambda_i} \mid i \in [1, n]\}$ be the representatives (with respect to $\cong$) of the lambda terms in the program P that are of type τ, and let

```
1  type Stream := (SCons (BigInt, Unit ⇒ Stream), SNil)
2  @memoize
3  def tail s = s match { SNil ⇒ SNil;
4    SCons (x, tfun) ⇒ (tfun Unit);
5  }
6  def take (n, s) =
7    { concrUntil(s, n) }
8    if (n ≤ 0) SNil else (s match {
9      SNil ⇒ SNil;
10     SCons (x, tfun) ⇒
11       let t := tail s in
12        let n1 := n − 1 in SCons(x, λa.take (n1, t));
13  }){ steps ≤ ? }
```

(a) A constant-time, lazy take operation

```
14 type tStream := (Take (BigInt, Stream), Other BigInt)
15 type Stream := (SCons (BigInt, tStream), SNil)
16 type Dcache := (Tail Stream)
17
18 def tail♯ (s, st) = s match { SNil ⇒ SNil;
19   SCons (x, tfun) ⇒ app (tfun, Unit, st); }
20 def app (cl,x,st) = cl match{ Take (n1,s1) ⇒ take♯ (n1,s1,st);}
21 def take♯ (n, s, st) =
22   { concrUntil♯ (s, n, st) }
23   if (n ≤ 0) (SNil, st, 3) else (s match {
24     SNil ⇒ (SNil, st, 5);
25     SCons (x, tfun) ⇒
26       let u := tail♯ (s, st) in
27        let nst := u.2 ∪ { (Tail s) } in
28         let u_cost := if ((Tail s) ∈ st) 1 else u.3 + 3 in
29          let ns := (SCons (x, Take (n − 1, u.1)) in
30           (ns, nst, u_cost + 10);
31   }){ res.3 ≤ ? }
```

Figure 6. Illustration of the translation shown in Fig. 5.

$\{l_i \mid i \in [1, n]\}$ be their labels. The datatype d_τ has $n + 1$ constructors denoted C_{l_i}, $i \in [1, n]$ and C_τ. That is, d_τ is of the form: type $d_\tau := (C_{l_1}\ \tau_1, \cdots, C_{l_n}\ \tau_n, C_\tau\ \mathsf{Int})$. The i^{th} constructor C_{l_i} represents the closure of the i^{th} lambda term e_{λ_i}. The parameter of the constructor represents $FV(e_{\lambda_i})$. The type τ_i is obtained by recursively replacing the function types by their closure datatypes in $type_P(FV(e_{\lambda_i}))$. The $(n+1)^{th}$ constructor C_τ of d_τ 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 tStream defined at line 14 represents the closures of lambdas of type Unit ⇒ Stream. The constructor Take of tStream represents the closure of λa.take (n1, t) created at line 12. As shown at line 29, the lambda is replaced by an instance of Take in the model. The constructor Other represents the stub closure c_τ.

Cache encoding. We instrument the expressions of the input program to explicitly track the changes to the cache as the program undergoes evaluation. Our instrumentation tracks only the keys of the cache, which are elements of $FVal$, as it fully specifies the state of the cache at every instance. We introduce a datatype Dcache to represent elements of $FVal$ defined as follows: type Dcache $:= (C_{f_1}\ \tau_1, \cdots, C_{f_n}\ \tau_n)$, where f_i's are functions in the program annotated with @memoize, and τ_i is the type of the parameter of f_i. In Fig. 6, the datatype Dcache with one constructor: (Tail Stream) corresponds to this datatype.

Translation of expressions. Fig. 5 formally defines the transformation $[\![\cdot]\!]_P$ that maps expressions of a input program P to a model program $P^\sharp$. For every expression e, $[\![e]\!]_P$ takes a state expression st representing the keys of the cache before the evaluation of e and returns the translated expression denoted $e^\sharp$. $e^\sharp$ is a triple where the first element $e^\sharp.{}_1$ corresponds to the value of e, the second element $e^\sharp.{}_2$ corresponds to the keys of the cache after evaluation of e, and the last element $e^\sharp.{}_3$ corresponds to the resource usage of e, which are explained in the sequel.

Cache-state propagation. The propagation of cache state proceeds top down in a store-passing style following the control flow of the program. To every function definition in the model, we add a fresh parameter st (of type Set[Dcache]) that represents the state of the cache at the beginning of the function (see translation of function definitions). This parameter is propagated through the bodies of the function recording all the calls that are memoized along the way. (E.g. see the translation of let expression.) The state parameter is used at two places: (a) by calls to memoized functions, and (b) by the cached construct to check whether the call given as argument is memoized. Consider the translation of a call to a memoized function shown in Fig. 5. It uses the input state parameter st to check whether the call would be a cache hit by testing if st contains $(C_f\ x)$ which represents the $FVal$: $(f\ x)$. The resource usage in the cache hit case is given by c_{hit}, whereas in the miss case it is a combination of c_{miss}, the cost of the call c_{call} and the resource usage of the callee $w.{}_3$. Finally, $(C_f\ x)$ is added to the output state to record that the call is memoized. Observe that the call always happens in the model regardless of whether or not it was memoized before. This encodes the referential transparency of memoized functions i.e, the value of the call is always equal to the result of the invoked function, and avoids having to specify an invariant on the cache. (Recall that we are not interested in the resource usage of the model.) During the translation of contracts, the precondition is translated using the initial state st and the postcondition using the state resulting after the translation of the body res.$_2$, as in the operational semantics. Any changes to the state caused by the contracts are discarded at the end of the contracts. The uses of res in the postcondition is replaced by res.$_1$, the uses of a resource R by res.$_3$, and the uses of inSt, representing the input cache state, by st.

Fig. 6(b) illustrates the result of propagating the state through the body of take function as outputted by our tool. Observe that after the call tail♯ (s, st) at line 26, an instance of (Tail s) is added to the output state to record that the call is memoized, and that the computation of steps at line 28 depends on whether or not (Tail s) belongs to the input state st.

Resource Instrumentation. The instrumentation for resources closely mimics the computation of resources in the operational semantics. It proceeds bottom-up, first instrumenting the sub-expressions of an expression e, and then using the resource usages of the sub-expressions to instrument e, with the exception of a call to a memoized function, which uses the state (propagated top down) reaching the call expression to handle the cache hit case. The model shown in Fig. 6(b) is obtained after a few straightforward static simplifications performed by our tool. For instance, the constants such as 10 and 5 that appear in the resource expressions 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_τ where $\tau = type_P(x)$, (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_i}, where l_i is the label of the lambda $\lambda a_i.e_i$ represented by the constructor, invokes the expression $[\![e'_i]\!]_P\ st$ 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_i}. If the closure matches C_τ, 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 tfun inside the function tail is translated to a call to the dispatch function app. (The case Other is omitted in app as we assume that the call is encapsulated.) Even though the set of possible cases in the function 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 $\{\mathcal{H}, \mathcal{H}^\sharp\} \subseteq Heap$. Define a relation $\underset{\mathcal{H},\mathcal{H}^\sharp,P}{\sim}$ on the semantic domains as follows: (subscripts omitted below for clarity)

1.$\forall a \in \mathbb{Z} \cup Bool.\ a \sim a$

2.$\forall c \in Cids, \{\bar{a}, \bar{b}\} \subseteq Val^n.\ c\ \bar{a} \sim c\ \bar{b}\ \mathit{iff}\ \forall i \in [1,n].a_i \sim b_i$

3.$\forall (e_\lambda, \sigma) \in Closure, v \in Val, l \in labels_P.(e_\lambda, \sigma) \sim C_l\ v\ \mathit{iff}$
$\sigma(FV(e_\lambda)) \sim v \wedge (e_\lambda/_{\cong,P}$ is defined and has label $l)$

4.$\forall f \in Fids$ defined in $P, \{a, b\} \subseteq Val.\ f\ a \sim C_f\ b\ \mathit{iff}\ a \sim b$

5.$\forall \mathcal{C} \in Cache, S \in Set.\ \mathcal{C} \sim S\ \mathit{iff}$
$|dom_P(\mathcal{C})| = |dom(S)| \wedge (\forall x \in dom_P(\mathcal{C}).\exists y \in S.x \sim y)$

6.$\forall \{a, b\} \subseteq Adr.\ a \sim b\ \mathit{iff}\ \mathcal{H}(a) \sim \mathcal{H}^\sharp(a)$

7.$\forall \{\sigma, \sigma^\sharp\} \subseteq Store.\ \sigma \sim \sigma^\sharp\ \mathit{iff}$
$dom(\sigma) \subseteq dom(\sigma^\sharp) \wedge \forall x \in dom(\sigma).\ \sigma(x) \sim \sigma^\sharp(x)$

where, $dom_P(\mathcal{C}) = \{(f\ u) \in dom(\mathcal{C}) \mid f \text{ is defined in } P\}$.

The relation formally captures that a cache is simulated by a set of instances of Dcache (rule 4 and 5), and that a closure is simulated by an instance of the datatype d_τ if the lambda of the closure has a representative in the program P with respect to $\cong$ (rule 3). We now define a simulation relation $\sim_P$ that relates an environment $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \in Env$ with an environment $\Gamma^\sharp : (\mathcal{H}^\sharp, \sigma^\sharp) \in Env^\sharp$, like a bisimulation relation between transition systems. But, somewhat unique to our setting, Γ is simulated by a pair $(\Gamma^\sharp, S)$ where $S \in Set$. We say $\Gamma \sim_P (\Gamma^\sharp, S)\ \mathit{iff}\ \mathcal{C} \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\sim} S$ and $\sigma \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\sim} \sigma^\sharp$.

Theorem 1 (***Bisimulation.***)**.** *Let P be a program. Let e, st and e' be expressions such that Let $e' = [\![e]\!]_P\ st$. Let $\Gamma \in Env$ and $\Gamma^\sharp \in Env^\sharp$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \sim_P (\Gamma^\sharp, S)$.*

(a) If $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ then $\exists \Gamma^\sharp_o \in Env^\sharp, u \in DVal$ such that $\Gamma^\sharp \vdash e' \Downarrow u, \Gamma^\sharp_o$ and

- $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o,P}{\sim} u._1$ • $p = u._3$

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

- $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o,P}{\sim} u._1$ • $p = u._3$

Using the above theorem, we now establish that for every function f in the program P, verifying the contracts of its translation $f^\sharp$ will imply that the contracts of f hold and vice-versa. A tricky aspect here is that there exist valid environments $\Gamma \in Env$ 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 $Env^\sharp$. The following theorem holds despite this because if such lambdas are invoked by P, the contracts of f and $f^\sharp$ do not hold for all environments as there exists an environment each in Env and $Env^\sharp$ that results in a contract violation in f and enforces the error condition in $f^\sharp$ respectively.

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

$$\forall \Gamma^\sharp \in {Env^\sharp}_{\tilde{e}',P^\sharp}.\exists u.\ \Gamma^\sharp \vdash p' \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash \tilde{e}' \Downarrow u\ \ \mathit{iff}$$
$$\forall \Gamma \in Env_{\tilde{e},P}.\exists v.\ \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \tilde{e} \Downarrow v$$

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 holes ι, $\forall \Gamma^\sharp \in {Env^\sharp}_{\tilde{e}',P^\sharp}.\ \Gamma^\sharp \vdash p' \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash (\tilde{e}'\ \iota) \Downarrow u\ \ \mathit{iff}$ $\forall \Gamma \in Env_{\tilde{e},P}.\Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash (\tilde{e}\ \iota) \Downarrow v$.

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 assume/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 $e_1 \to e_2$ denote that whenever e_1 does not evaluate to *false*, e_2 evaluates to *true* i.e, $\forall \Gamma \in Env^\sharp.\ \Gamma \vdash e_1 \Downarrow \mathit{false} \vee \Gamma \vdash e_2 \Downarrow \mathit{true}$. (The operation $\to$ can be considered as an implication with respect to the operational semantics of the model language.) We use $\models_P e_1 \to e_2$ 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, $e_1 \to e_2$ is *guaranteed*. The modular reasoning described above corresponds to the following two rules:

Function-level modular reasoning:

- For each def $f\ x = \{pre\}\ e\ \{post\}$, $\models_P pre \to post[e/\mathsf{res}]$
- For each call site $c = f\ x$ in P, $\models_P path(c) \to pre(c)$

Recall that the variable res refers to the result of e in the postcondition of e. For a call $c = f\ x$, we use $pre(c)$ 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 c from the entry of the function containing c. For instance, the path condition of the call tail$^\sharp$ (s,st) at line 26 of the program shown in Fig. 6(b) is: concrUntil$^\sharp$ (s, n, st) $\wedge$ n $>$ 0 $\wedge$ s = SCons (x,tfun). For programs with templates, the assume/guarantee assertions generated as above would have holes ($TVars$). The goal is then to find an assignment ι 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 $\Gamma \in Env^\sharp$ (by the definition of $\to$), even though for contract verification

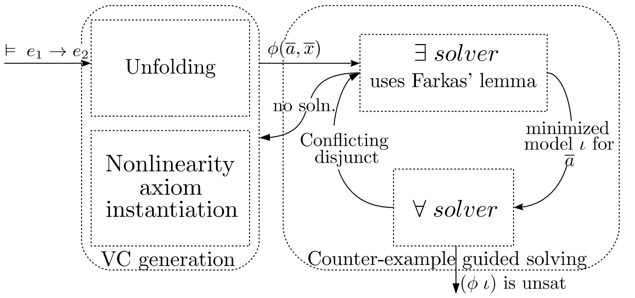

Figure 7. Counter-example guided inference for numerical holes.

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 $\mathsf{take}^\sharp$ 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 $\mathsf{concrUntil}^\sharp$ (s1,n1,st). To make this example verify, it would in fact require $\mathsf{concrUntil}^\sharp$ to hold on the arguments of every instance of Take reachable from the recursive datatype Stream, due to the mutual recursion between app, $\mathsf{take}^\sharp$ and $\mathsf{tail}^\sharp$. That is, the precondition of app would need a function pre (cl,st) defined as follows:

```
def pre (cl, st) = cl match{
  Take (n1,s1) ⇒ concrUntil♯ (s1,n1,st) ∧
      (s1 match { SCons(x, t) ⇒ pre (t, st); SNil ⇒ 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 $\mathsf{concrUntil}^\sharp$ 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 $p \in E_{spec}$ is cache monotonic iff whenever it holds in an environment with a cache $\mathcal{C}_1$, it also holds in all environments where the cache has more entries than $\mathcal{C}_1$. 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 $\Gamma_1 : (\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F)$ and $\Gamma_2 : (\mathcal{C}_2, \mathcal{H}_2, \sigma_2, F)$. We say $\Gamma_1 \sqsubseteq \Gamma_2$ iff every component of Γ_2 has more entries than the corresponding component of Γ_1 i.e, $(k, v) \in C_1 \Rightarrow (k, v) \in C_2$, where C could be $\mathcal{H}$, $\mathcal{C}$ or σ. A property pr is cache monotonic iff $\forall\{\Gamma_1, \Gamma_2\} \subseteq Env.(\Gamma_1 \sqsubseteq \Gamma_2 \wedge \Gamma_1 \vdash pr \Downarrow true) \Rightarrow \Gamma_2 \vdash pr \Downarrow true$. To check if a property pr is cache monotonic it suffices to check the following property on the translation of pr with respect to $[\![\cdot]\!]_P$ defined in Fig. 5: $(st_1 \subseteq st_2 \wedge [\![pr]\!]_P\ st_1) \to [\![pr]\!]_P\ st_2$.

Creation-dispatch rule for encapsulated calls. Recall that each indirect call $x\ y$ has a set of target lambdas that are estimated at the time of model construction based on $type_P(x)$. Let $\Lambda = \{e_i \mid i \in [1, n]\}$, 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 $Clo^\sharp = \{C_i\ w_i \mid i \in [1, n]\}$ be the closure constructions in the model of P representing the lambdas Λ. In the model program, the dispatch functions App_l corresponding to the encapsulated calls invoke the function $f_i^\sharp$ (the translation of f_i) in each case $C_i\ w_i$ (see Fig. 5 and the illustration Fig 6(b)). Let $DispCalls = \{f_i^\sharp\ (x, z_i, st) \mid i \in [1, n]\}$ be the calls invoked by such App_l functions. Let $Props = \{\rho_i \mid i \in [1, n]\}$ be a set of boolean-valued expressions (properties) in E_{spec} defined on the captured argument y_i of the lambda $e_i \in \Lambda$ (i.e, ρ_i has only y_i as free variable). We augment the function-level assume/guarantee rules with the following condition: if each property ρ_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 $\mathsf{def}\, f\ x := \{pre\}\ e\ \{post\}$, $\models_P pre \to post[e/\mathsf{res}]$

II. For each call site $c \notin DispCalls$, $\models_P path(c) \to pre(c)$

III. *(Cache monotonicity)* For each $\rho_i \in Props$
$\models_P (st_1 \subseteq st_2 \wedge [\![\rho_i]\!]_P\ st_1) \to [\![\rho_i]\!]_P\ st_2$

IV. For each closure construction site $c = C_i\ w_i$ in $Clo^\sharp$
$\models_P path(c) \to ([\![\rho_i]\!]_P\ st(c))$

V. For each call site $c = f_i^\sharp\ (x, z_i, st)$ in $DispCalls$
$\models_P (path(c) \wedge [\![\rho_i[z_i/y_i]]\!]_P\ st) \to pre(c)$

In the above rules, $st(c)$ denotes the cache-state expression propagated by the translation function $[\![\cdot]\!]_P$ to an expression c 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, whereas the state expression reaching the line 29 is nst.

While the above reasoning holds irrespective of the how the properties ρ_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 ρ_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) concrUntil is a cache monotonic property: $\models_P (st_1 \subseteq st_2 \wedge \mathsf{concrUntil}(s, i, st_1)) \to \mathsf{concrUntil}(s, i, st_2)$, and (b) that the property concrUntil(u.$_1$, n$-$1,nst) holds at the point of creation of the closure Take(n$-$1,u.$_1$) at line 29. The property concrUntil(s1,n1,st) is assumed to hold while checking the precondition of call to $\mathsf{take}^\sharp$ 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 $P^\sharp$ the model program. Let $\mathsf{def}\, f^\sharp\ x := \tilde{e}$ where $\tilde{e} = \{p\}\ e\ \{s\}$ be a function definition in $P^\sharp$. If every function defined in P terminate and the assume/guarantee assertions (I) to (V) defined above hold, the contracts of $f^\sharp$ holds i.e, $\forall \Gamma^\sharp \in Env^\sharp{}_{\tilde{e}, P^\sharp}.\exists u.\ \Gamma^\sharp \vdash p \Downarrow false \vee \Gamma^\sharp \vdash \tilde{e} \Downarrow u$.*

Solving parametric verification conditions. To solve the assertions generated by assume/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 assume/guarantee assertion

$\models_P e_1 \rightarrow e_2$ the VC generation phase converts it to a quantifier-free formula (VC) of the form $\phi(\bar{x}, \bar{a})$, where the variables $\bar{a}$ corresponds to the numerical holes, such that the assume/guarantee assertion holds if there exists a assignment ι for $\bar{a}$ such that $\phi\ \iota$ is unsatisfiable. (The VC could be thought of as a $\exists\forall$ formula where the holes are existentially quantified, and the rest including uninterpreted function symbols are universally quantified.)

Converting an assume/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 $\bar{x}$ are axiomatized in the VC. The VCs thus generated belong to the theory $\mathcal{T}$ 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 $\bar{a}$ and x belonging to $\bar{x}$. 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: $\phi(\bar{x}, \bar{a})$, the solution for $\bar{a}$ that will make ϕ 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 ι for the free variables $\bar{a}$. 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 ι makes ϕ unsatisfiable. That is, if $\phi\ \iota$ is unsatisfiable. If not, it chooses a disjunct $d(\bar{x}, \bar{a})$ satisfiable under ι 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 $C(\bar{a})$, based on Farkas' Lemma [20], to obtain the next candidate assignment for $\bar{a}$ that will make $d(\bar{x}, \bar{a})$ 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 $\phi(\bar{x}, \bar{a})$ 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 ι such that $dom(\iota) = |\bar{a}|$ and $(\phi\ \iota)$ is $\mathcal{T}$-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 $\forall x, st_1, st_2.(f^\sharp\ (x, st_1)).1 = (f^\sharp\ (x, st_2)).1$ for every function $f^\sharp$ in the model. We encode this axiom efficiently by adding the predicate $f^\sharp\ (x, st_1) = UF_f(x)$ for every application of $f^\sharp$ in the VC, where UF_f is a unique uninterpreted function for $f^\sharp$. 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: $st \subseteq f^\sharp\ (x, st).2$ for every application of $f^\sharp$ in the VC. (c) Also, whenever the counter-example guided solving fails, we unfold only calls along the disjuncts $d(\bar{x}, \bar{a})$ 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.

B	***I***	***(dynamic/static) * 100***		***(optimal/static) * 100***	
		steps	***alloc***	***steps***	***alloc***
sel	10k	99	99	100	100
prims	1k	60	89	82	100
fibs	10k	99	99	100	100
hams	10k	86	83	98	100
slib	10k	65	75	85	88
rtq	2^{20}	93	83	97	87
msort	10k	90	91	96	97
deq	2^{20}	48	48	59	62
num	2^{20}	94	97	96	100
conq	2^{20}	72	54	82	72
lcs	1k	88	100	95	100
levd	1k	90	100	96	100
hmem	10k	79	100	92	100
ws	10k	99	100	100	100
ks	1k	94	100	99	100
pp	10k	77	70	88	84
vit	100	42	100	86	100
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.

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 byte code 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

Benchmark	*LOC*	*BC*	*T*	*S*	*AT*	steps $\leq$ **Resource bounds**	alloc $\leq$
Lazy data-structures							
Lazy Selection Sort (*sel*)	70	36kb	4	1	1m	$15k \cdot l.size + 8k + 13$	$2k \cdot l.size + 2k + 2$
Prime Stream (*prims*)	95	51kb	7	2	1m	$16n^2 + 28$	$6n - 11$
Fibonacci Stream (*fibs*) [12]	199	59kb	5	5	2m	$45n + 4$	$4n$
Hamming Stream (*hams*) [12]	223	78kb	8	6	1m	$129n + 4$	$16n$
Stream library (*slib*) [72]	408	0.1mb	22	5	1m	$25l.size + 6$	$3l.size$
Lazy Mergesort (*msort*) [4]	290	0.1mb	6	8	1m	$36k\lfloor \log l.size \rfloor + 53l.size + 22$	$6k\lfloor \log l.size \rfloor + 6l.size + 3$
Real time queue (*rtq*) [58, 59]	207	69kb	5	6	1m	40	7
Deque (*deq*) [58, 59]	426	0.1mb	16	7	5m	893	78
Lazy Numerical Rep.(*num*)[59]	546	0.1mb	6	25	1m	106	15
Conqueue (*conq*) [61, 62]	880	0.2mb	12	33	5m	$29\lvert xs.lvl - ys.lvl\rvert + 8$	$2\lvert xs.lvl - ys.lvl\rvert + 1$
Dynamic Programming							
LCS (*lcs*) [21]	121	37kb	4	4	1m	$30mn + 30m + 30n + 28$	$2mn + 2m + 2n + 3$
Levenshtein Distance(*levd*) [24]	110	37kb	4	4	1m	$36mn + 36m + 36n + 34$	$2mn + 2m + 2 + 3$
Hamming Numbers (*hm*) [12]	105	44kb	3	3	3m	$66n + 65$	$3n + 4$
Weight Scheduling (*ws*) [21]	133	44kb	3	5	1m	$20jobi + 19$	$2jobi + 3$
Knapsack (*ks*) [21]	122	48kb	5	4	1m	$17(w \cdot i.size) + 18w + 17i.size + 18$	$2w + 3$
Packrat Parsing (*pp*) [30]	249	73kb	7	5	1m	$61n + 58$	$10n + 10$
Viterbi (*vit*) [76]	191	63kb	6	7	1m	$34k^2t + 34k^2 - 6kt + 14k + 47t + 26$	$2kt + 2k + 4t + 5$

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

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 bounds* 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 $\lfloor \log \rfloor$ are expressed as a recursive function that uses integer division: log(x) = 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 *conq*). 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 upto 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: $Mean\left(\frac{\text{resource consumed by the } i^{th} \text{ input}}{\text{static estimate for } i^{th} \text{ input}} \times 100\right)$. 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 function 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 $O(k \cdot \lfloor \log(l.size) \rfloor)$ 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, $\lfloor \log(l.size) \rfloor$ 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 *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 nonlinearity 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 *Viterbi* and *prims*). In the sequel, we discuss the benchmarks and the results of their evaluation in more detail.

Cyclic streams. The benchmarks *fibs* and *hams* implement infinite fibonacci and hamming sequences as cyclic streams using lazy *zipWith* and *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 *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 *rtq*, *deq*, *num*, and *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 *rtq* [58] benchmark takes a few nanoseconds to persistently enqueue an element into a queue of size 2^{30}. The *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 *spines* that track content, and a list of references to closures nested deep within the spines: *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 rotateDrop function shown in Fig 8.4 of [59], which was unraveled by the system.

As the results in Fig. 9 show, the inferred bounds were at least 83% accurate for *rtq* and *num* benchmarks, but have low accuracy for *deq* and *conq* benchmarks. On further analysis of *deq* we found that the bounds inferred by our system for the inner functions of *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 *slib* is a collection of operations over streams such as *map*, *scan*, *cycle* etc. The operations were chosen from the Haskell stream library [72]. We excluded functions such as *filter* that can potentially diverge on infinite streams. The bounds presented are for a specific client of the library. The benchmarks *msort* and *sel* implement lazy sorted streams that allows accessing the k^{th} minimum without performing the entire sorting. In particular, *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 *pp* is a memoized implementation of a packrat parser presented by Ford [30] for the *parsing expression grammar* used in that work. As shown in Fig. 9, the inferred bounds for steps are on average 90% accurate for the dynamic programming algorithms except *pp* and *vit*, and is 100% accurate in the case of alloc for all benchmarks except *pp*. In the case of *vit*, the main reason for inaccuracy stems from the *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 *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: pAdd, pMul, and 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 *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, 32, 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, 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 *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

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[2] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis Symposium, SAS*, pages 117–133, 2010.

[3] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *Static Analysis Symposium, SAS*, pages 405–421, 2012.

[4] H. Apfelmus. Quicksort and k-th smallest elements. 2009.

[5] A. W. Appel. Intensional equality ;=) for continuations. *SIGPLAN Not.*, 31(2), Feb. 1996.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, Oct. 1989.

[7] M. Avanzini, U. D. Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *International Conference on Functional Programming, ICFP*, pages 152–164, 2015.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification, CAV*, pages 171–177, 2011.

[9] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1):79 – 103, 2004.

[10] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[11] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *Computer Aided Verification, CAV*, 2013.

[12] R. Bird and P. Wadler. *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd., 1988.

[13] R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An overview of the Leon verification system. In *Scala Workshop*, 2013.

[14] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: a proof assistant perspective. In *International Conference on Functional Programming, ICFP*, pages 192–204, 2015.

[15] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In *European Symposium on Programming, ESOP*, pages 359–382, 2015.

[16] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: making parametric shape analysis competitive. In *Computer Aided Verification, CAV*, pages 221–225, 2007.

[17] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, pages 13:1–13:50, 2016.

[18] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation, PLDI*, 2014.

[19] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation, PLDI*, pages 234–245, 2011.

[20] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification, CAV*, 2003.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.

[22] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of Programming Languages, POPL*, pages 133–144, 2008.

[23] N. Danner, J. Paykin, and J. S. Royer. A static cost analysis for a higher-order language. In *Workshop on Programming languages meets program verification, PLPV*, pages 25–34, 2013.

[24] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[25] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, pages 337–340, 2008.

[26] D. Distefano and M. J. Parkinson J. jStar: Towards practical verification for java. In *Object-oriented Programming Systems Languages and Applications, OOPSLA*, pages 213–226, 2008.

[27] M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming, IWACO*, page 58, 2003.

[28] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming, ICFP*, pages 48–59, 2002.

[29] A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Programming Languages and Systems - 12th Asian Symposium, APLAS*, pages 275–295, 2014.

[30] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *International Conference on Functional Programming ICFP*, pages 36–47, 2002.

[31] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, Feb. 2011.

[32] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages, POPL*, 2009.

[33] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Compuer Aided Verification, CAV*, 2015.

[34] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[35] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource Aware ML. In *Computer Aided Verification, CAV*, pages 781–786, 2012.

[36] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of NASA Formal Methods, NFM*, pages 41–55, 2011.

[37] N. D. Jones and N. Bohr. Termination analysis of the untyped lamba-calculus. In *Rewriting Techniques and Applications, RTA*, pages 1–23, 2004.

[38] S. Jost, K. Hammond, H. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Principles of Programming Languages, POPL*, pages 223–236, 2010.

[39] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *Foundations of Software Engineering, FSE*, pages 105–116, 2006.

[40] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[41] G. Klein, P. Derrin, and K. Elphinstone. Experience report: Sel4: Formally verifying a high-performance microkernel. In *International Conference on Functional Programming, ICFP*, pages 91–96, 2009.

[42] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, Feb. 2010.

[43] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Principles of Programming Languages, POPL*, pages 416–428, 2009.

[44] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Programming Language Design and Implementation, PLDI*, pages 222–233, 2011.

[45] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 36(3), 2006.

[46] J. Launchbury. A natural semantics for lazy evaluation. In *Principles of Programming Languages, POPL*, 1993.

[47] D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, Apr. 1988.

[48] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

[49] K. R. M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *Formal Methods, FM*, pages 382–398, 2014.

[50] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[51] R. Madhavan and V. Kuncak. Symbolic resource bound inference, EPFL-REPORT-190578. Technical report, EPFL, 2014.

[52] R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In *Computer Aided Verification, CAV*, pages 762–778, 2014.

[53] J. A. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *International Conference on Logic Programming, ICLP*, pages 348–363, 2007.

[54] P. C. Nguyen and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *Programming Language Design and Implementation, PLDI*, pages 446–456, 2015.

[55] P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Soft contract verification. In *international conference on Functional programming, ICFP*, pages 139–152, 2014.

[56] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[57] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[58] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5:583–592, 10 1995.

[59] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[60] R. Piskac, T. Wies, and D. Zufferey. Grasshopper - complete heap verification with mixed specifications. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 124–139, 2014.

[61] A. Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, EPFL, 2014.

[62] A. Prokopec and M. Odersky. Conc-trees for functional and parallel programming. In *Languages and Compilers for Parallel Computing, LCPC*, pages 254–268, 2015.

[63] A. Reynolds and V. Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 80–98, 2015.

[64] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[65] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles of Programming Languages, POPL*, pages 105–118, 1999.

[66] D. Sands. *Calculi for Time Anlaysis of Functional Programs*. PhD thesis, Imperial College, University of London, 1990.

[67] D. Sands. Complexity analysis for a lazy higher-order language. In *European Symposium on Programming, ESOP*, pages 361–376, 1990.

[68] D. Sereni. *Termination analysis of higher-order functional programs*. PhD thesis, University of Oxford, UK, 2006.

[69] H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *International Conference on Functional Programming, ICFP*, pages 165–176, 2012.

[70] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Computer Aided Verification CAV*, pages 745–761, 2014.

[71] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Symposium on Static Analysis SAS*, 2011.

[72] W. Swierstra. Stream: A library for manipulating infinite lists. `https://hackage.haskell.org/package/Stream-0.4.7.2/docs/Data-Stream.html`. 2015.

[73] S. Tobin-Hochstadt and D. V. Horn. Higher-order symbolic execution via contracts. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 537–554, 2012.

[74] P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In *European Symposium on Programming, ESOP*, 2015.

[75] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *International Conference on Functional Programming, ICFP*, pages 269–282, 2014.

[76] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967.

[77] N. Voirol, E. Kneuss, and V. Kuncak. Counter-example complete verification for higher-order functions. In *Symposium on Scala*, pages 18–29, 2015.

[78] N. Voirol and V. Kuncak. Automating verification of functional programs with quantified invariants, EPFL-REPORT-222712. Technical report, EPFL, 2016.

[79] D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. In *Principles of Programming Languages, POPL*, pages 431–442, 2013.

[80] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[81] D. N. Xu. Hybrid contract checking via symbolic simplification. In *Workshop on Partial Evaluation and Program Manipulation, PEPM*, pages 107–116, 2012.

[82] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for haskell. In *Principles of Programming Languages, POPL*, pages 41–52, 2009.

[83] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *Programming Language Design and Implementation, PLDI*, 2008.

[84] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis Symposium, SAS*, pages 280–297, 2011.

A. Formal Definitions, Semantics and Proofs

A.1 Semantics and Properties of Input Language

In all the formalism that follow we adopt the following convention. If Γ_i (or Γ^i) is an environment then we refer to its individual components C of the environment, namely $(\mathcal{C}, \mathcal{H}, \sigma, F)$, using C_i (or C^i), respectively.

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

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

$$\textsc{Let1}\quad \langle \Gamma, \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \rangle \rightsquigarrow \langle \Gamma, e_1 \rangle$$

$$\textsc{Let2}\quad \frac{\Gamma \vdash e_1 \Downarrow v_1, (\mathcal{C}', \mathcal{H}', \sigma')}{\langle \Gamma, \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \rangle \rightsquigarrow \langle (\mathcal{C}', \mathcal{H}', \sigma'[x \mapsto v_1]), e_2 \rangle}$$

$$\textsc{Match}\quad \frac{\mathcal{H}(\sigma(x)) = C_i\ \bar{v}}{\langle \Gamma : (\mathcal{C}, \mathcal{H}, \sigma), x\ \mathsf{match}\ \{C_i\ \bar{x_i} \Rightarrow e_i)\}_{i=1}^{n} \rangle \rightsquigarrow \langle (\mathcal{C}, \mathcal{H}, \sigma[\bar{x_i} \mapsto \bar{v}]), e_i \rangle}$$

$$\textsc{ConcreteCall}\quad \langle \Gamma : (\mathcal{C}, \mathcal{H}, \sigma), f\ u \rangle \rightsquigarrow \langle (\mathcal{C}, \mathcal{H}, \sigma[param_\Gamma(f) \mapsto u]), body_\Gamma(f) \rangle$$

$$\textsc{NonMemoizedCall}\quad \frac{f \in Fids \qquad f \notin Mem_\Gamma}{\langle \Gamma, f\ x \rangle \rightsquigarrow \langle \Gamma, f\ \sigma(x) \rangle}$$

$$\textsc{IndirectCall}\quad \frac{\mathcal{H}(\sigma(x)) = (\lambda z.e, \sigma')}{\langle \Gamma : (\mathcal{C}, \mathcal{H}, \sigma), x\ y \rangle \rightsquigarrow \langle (\mathcal{C}, \mathcal{H}, (\sigma \uplus \sigma')[z \mapsto \sigma(y)]), e \rangle}$$

$$\textsc{MemoCallMiss}\quad \frac{f \in Mem_\Gamma \qquad \neg\big((f\ \sigma(x)) \in_{\mathcal{H}} dom(\mathcal{C})\big)}{\langle \Gamma, f\ x \rangle \rightsquigarrow \langle \Gamma, f\ \sigma(x) \rangle}$$

$$\textsc{Pre}\quad \langle \Gamma, \{pre\}\ e\ \{post\} \rangle \rightsquigarrow \langle \Gamma, pre \rangle$$

$$\textsc{Body}\quad \frac{\Gamma \vdash pre \Downarrow true}{\langle \Gamma, \{pre\}\ e\ \{post\} \rangle \rightsquigarrow \langle \Gamma, e \rangle}$$

$$\textsc{Post}\quad \frac{\Gamma \vdash pre \Downarrow true \qquad \Gamma \vdash e \Downarrow_q v, \Gamma_2 : (\mathcal{C}_2, \mathcal{H}_2, \sigma_2)}{\langle \Gamma, \{pre\}\ e\ \{post\} \rangle \rightsquigarrow \langle (\mathcal{C}_2, \mathcal{H}_2, \sigma_2[R \mapsto q, \mathsf{res} \mapsto v]), post \rangle}$$

Figure 10. Definition of the reachability relation.

of the evaluation. That is, we inductively establish that $\forall n \in \mathbb{N}.\ \neg\big(\exists k > n, e, \Gamma''.\langle \Gamma, e \rangle \rightsquigarrow^k \langle \Gamma'', e \rangle\big) \Rightarrow \rho(\Gamma, e, v, \Gamma', p)$. This boils down to the following strategy. For every semantics rule Rule, we assume that the property holds for the big-step reductions in the antecedent and establish that it holds in the consequent. The base cases of the induction are the rules: Cst, Var, Prim, Equal, Cons, Lambda, MemoCallHit and Cached, which do not have any big-step reductions in the antecedents. Every other rule is an inductive step. We refer to this as structural induction over the big-step semantic rules. Many of the theorems that follow are established using this form of structural induction.

Structural induction over $\approx$ and $\sim$. Recall that the relations $\approx$, and $\sim$ are defined recursively. As is 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 $\approx$, $\hat{\sim}$ 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 $\bigcup_{i=0}^{n} R^i$. 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 $\sim$, 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.

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

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

Lemma 5. *Let $\mathcal{H}$ be an acyclic heap. The structural equivalence relation $\underset{\mathcal{H}}{\approx}$ is reflexive, transitive and symmetric. That is,*
(a) $x \underset{\mathcal{H}}{\approx} y \wedge y \underset{\mathcal{H}}{\approx} z \Rightarrow x \underset{\mathcal{H}}{\approx} z$
(b) $x \underset{\mathcal{H}}{\approx} y \Rightarrow y \underset{\mathcal{H}}{\approx} x$
(c) $x \underset{\mathcal{H}}{\approx} x$

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 $<$. 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 $\{h_1, h_2\} \subseteq A \to B$, we say $h_1 \sqsubseteq h_2$ iff h_2 has more entries than h_1. That is, $a \in dom(h_1)$ 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

$$(\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F) \sqsubseteq (\mathcal{C}_2, \mathcal{H}_2, \sigma_2, F) \triangleq$$
$$\mathcal{C}_1 \sqsubseteq \mathcal{C}_2 \wedge \mathcal{H}_1 \sqsubseteq \mathcal{H}_2 \wedge \sigma_1 \sqsubseteq \sigma_2$$

Structural Simulation Relation. Similar to structural equivalence, we define a structural simulation $\underset{\mathcal{H}_1, \mathcal{H}_2}{\approx}$, 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.)

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

Notice that the only change compared to $\underset{\mathcal{H}}{\approx}$ 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 $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$, $\underset{\mathcal{H}_1,\mathcal{H}_2}{\approx}$ reduces to $\underset{\mathcal{H}_2}{\approx}$
- *(Symmetry)* $x \underset{\mathcal{H}_1,\mathcal{H}_2}{\approx} y$ implies $y \underset{\mathcal{H}_2,\mathcal{H}_1}{\approx} x$
- *(Transitivity)* $x \underset{\mathcal{H}_1,\mathcal{H}_2}{\approx} y$and $y \underset{\mathcal{H}_2,\mathcal{H}_3}{\approx} z$ implies $x \underset{\mathcal{H}_1,\mathcal{H}_3}{\approx} y$
- If $u \underset{\mathcal{H}_1,\mathcal{H}_2}{\approx} v$ then $(u \underset{\mathcal{H}_1,\mathcal{H}_2}{\approx} v' \Leftrightarrow v \underset{\mathcal{H}_2}{\approx} v')$ and $(u' \underset{\mathcal{H},\mathcal{H}_2}{\sim} v \Leftrightarrow u \underset{\mathcal{H}_1}{\approx} u')$.

Structural Abstraction Relation. Using the structural simulation relation, we now define a structural abstraction relation $\lesssim$ between two environments.

$$(\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F) \lesssim (\mathcal{C}_2, \mathcal{H}_2, \sigma_2, F) \triangleq$$
$$\mathcal{C}_1 \underset{\mathcal{H}_1,\mathcal{H}_2}{\lesssim} \mathcal{C}_2 \wedge \sigma_1 \underset{\mathcal{H}_1,\mathcal{H}_2}{\lesssim} \sigma_2, \text{ where,}$$
$$\sigma_1 \underset{\mathcal{H}_1,\mathcal{H}_2}{\lesssim} \sigma_2\ \mathit{iff}\ \forall x \in dom(\sigma_1).\sigma_1(x) \underset{\mathcal{H}_1,\mathcal{H}_2}{\approx} \sigma_2(x), \text{ and}$$
$$\mathcal{C}_1 \lesssim \mathcal{C}_2\ \mathit{iff}\ \forall k \in dom(\mathcal{C}_1).$$
$$\exists k' \in dom(\mathcal{C}_2).k \underset{\mathcal{H}_1,\mathcal{H}_2}{\approx} k' \wedge \mathcal{C}_1(k) \underset{\mathcal{H}_1,\mathcal{H}_2}{\approx} \mathcal{C}_2(k')$$

Note that $\sqsubseteq$ is a stronger relation than $\lesssim$.

Structural Equivalence of Environments. We say two environments are structurally equivalent iff $\Gamma_1 \lesssim \Gamma_2$ and vice-versa. That is,

$$\Gamma_1 \approx \Gamma_2\ \mathit{iff}\ (\Gamma_1 \lesssim \Gamma_2 \wedge \Gamma_2 \lesssim \Gamma_1)$$

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.

Lemma 6. *For all $\{\Gamma_1, \Gamma_2\} \subseteq Env$ such that $\Gamma_1 \approx \Gamma_2$, for all expression e,*

$$\Gamma_1 \vdash e \Downarrow_p u, \Gamma_1' \Rightarrow \quad \exists v, q, \Gamma_2'.\ \Gamma_2 \vdash e \Downarrow_q v, \Gamma_2' \wedge \Gamma_1' \approx \Gamma_2' \wedge u \underset{\mathcal{H}_1',\mathcal{H}_2'}{\approx} v \wedge p = q$$

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.

Lemma 7. *Let $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F)$, $\Gamma_1 : (\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F)$ and e be an expression. If $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_1, e_1\rangle$ or $\Gamma \vdash e \Downarrow v, \Gamma_1$ then $\mathcal{H} \sqsubseteq \mathcal{H}_1$ and $\mathcal{C} \sqsubseteq \mathcal{C}_1$. That is, the evaluation can only add more entries to the heap and cache, and cannot update existing entries.*

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. □

Lemma 8. *Let $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env$ and e be an expression. Let $\Gamma_1 = (\mathcal{C}, \mathcal{H}_1, \sigma, F)$ where $\mathcal{H} \subseteq \mathcal{H}_1$. If $\Gamma \vdash e \Downarrow_p u, \Gamma'$ then $\Gamma_1 \vdash e \Downarrow_p v, \Gamma_1'$ and $u \underset{\mathcal{H}_1'}{\approx} v$. That is, adding more entries to the heap preserves the result of the evaluation with respect to the structural equivalence relation $\approx$.*

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 $\mathcal{H}$ via the store σ e.g. as $\mathcal{H}(\sigma(x))$, 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 $Env_P \subseteq Env$ be the set of environments such that for every $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env_P$ the following properties hold.

(a) $(dom(\sigma) \cap Adr) \subseteq dom(\mathcal{H})$
(b) For all variable x in P,
$x \in dom(\sigma)$ implies that $\sigma(x)$ 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 $\lambda x.\ f\ (x, y) \in range(\mathcal{H})$, f is defined in F.
(f) $\neg\exists\{k, k'\} \subseteq dom(\mathcal{C}).\ k \neq k' \wedge k \underset{\mathcal{H}}{\approx} k'$
(g) $\forall (k, v) \in \mathcal{C}.\exists \mathcal{C}', \mathcal{C}'', \mathcal{H}'$ s.t. $\mathcal{C}' \sqsubseteq \mathcal{C}'' \sqsubseteq \mathcal{C} \wedge \mathcal{H}' \sqsubseteq \mathcal{H}'' \sqsubseteq \mathcal{H}$
$\wedge (\mathcal{C}', \mathcal{H}', \{\}, F) \vdash k \Downarrow v, (\mathcal{C}'', \mathcal{H}'', \{\}, F)$

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 $\Gamma \in Env_P$ and $\langle e, \Gamma\rangle \rightsquigarrow \langle e', \Gamma'\rangle$ then $\Gamma' \in Env_P$. Moreover, if Γ is defined on all the free variables of e, denoted $fv(e)$, then Γ' will be defined on all the free variables of e'.

Lemma 9. *Let e and e' be expressions in a program P. Let $\Gamma \in Env_P$ and $\Gamma' \in Env$. If $\Gamma \vdash e \Downarrow v, \Gamma'$ then $\Gamma' \in Env_P$.*

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 $\mathcal{C}_1$ and $\mathcal{H}_1$ and the output heap and cache are $\mathcal{C}_1'$ and $\mathcal{H}_1'$ In this case, the invariant (g) holds for the following assignment: $\mathcal{C} = \mathcal{C}'' = \mathcal{C}_1'$ (see Fig. 4), $\mathcal{C}' = \mathcal{C}_1$. Similarly, $\mathcal{H} = \mathcal{H}'' = \mathcal{H}_1'$ and $\mathcal{H}' = \mathcal{H}_1$. 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 $\Gamma \in Env_P$ and $\Gamma' \in Env$. (a) If $\langle e, \Gamma\rangle \rightsquigarrow^* \langle e', \Gamma'\rangle$ then (a) $\Gamma' \in Env_P$ and (b) $fv(e) \subseteq dom(\Gamma) \Rightarrow fv(e') \subseteq dom(\Gamma')$.*

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

In the rest of the paper whenever we say $\Gamma \in Env$ we assume that $\Gamma \in Env_P$ if the program under consideration is clear from the context.

Valid Environments. Recall that in section 2 we define the valid environments $Env_{e,P}$ that reach an expression e in a program P as

$$\{\Gamma \mid \exists P'.\langle \Gamma_{P' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, e\rangle\}$$

We also impose a constraint that the evaluation $\langle \Gamma_{P' \| P}, e_{entry}\rangle$ is terminating, unless the functions in program P are non-terminating. This is because for every $\Gamma \in \Gamma_{\tilde{e},P}$ there always exist a program P'' such that $\langle \Gamma_{P'' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle$, 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 $\Gamma_{P' \| P}$. Moreover, $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env_{e,P}$ implies that $fv(e) \subseteq dom(\sigma)$.

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 $\lesssim$ 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 $\sqsubseteq$ by Lemma 7, and hence are also related by the weaker relation $\lesssim$.)

Lemma 11. *Let* $\Gamma_1 : (\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F)$ *in* Env_P. *For all expression* $e_s \in (E_{src} \cup FVal)$, *if* $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ *then* $\forall \Gamma_2 : (\mathcal{C}_2, \mathcal{H}_2, \sigma_2, F) \in Env_P$ *s.t.* ${\Gamma_1}' \lesssim \Gamma_2$,

$$\Gamma_2 \vdash e_s \Downarrow v, {\Gamma_2}' \Rightarrow u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v$$

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 $\Gamma_1 \sqsubseteq {\Gamma_1}'$. (The store components of Γ_1 and ${\Gamma_1}'$ are identical.) Therefore, $\Gamma_1 \lesssim \Gamma_2$. Every case other than MEMOCALLHIT uses only the heap and the store (and not the cache). Since $\Gamma_1 \lesssim \Gamma_2$, the free variables in the expressions are bound to structurally similar values in Γ_1 and Γ_2. It is easy to see that in each of the cases the resulting values are also structurally similar. Now say the evaluation $\Gamma_1 \vdash e_s \Downarrow u$ uses MEMOCALLHIT. Therefore, e is of the form $(f\ x)$ and $\sigma_1(x) \underset{\mathcal{H}_1}{\approx} k$ where k is a key in the cache $\mathcal{C}_1$. Since $\Gamma_1 \lesssim \Gamma_2$, $\sigma_1(x) \underset{\mathcal{H}_1, \mathcal{H}_2}{\approx} \sigma_2(x)$ and there exists a $k' \in dom(\mathcal{C}_2)$ such that $k \underset{\mathcal{H}_1, \mathcal{H}_2}{\approx} k'$. By the properties of $\approx$, $\sigma_2(x) \underset{\mathcal{H}_2}{\approx} k'$ 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 $\lesssim$).

Now say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ uses one of the inductive cases: LET, MATCH, CONCRETECALL, NONMEMOIZEDCALL, MEMOCALLMISS and CONTRACT. If $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ uses any rule RULE other than MEMOCALLMISS, then $\Gamma_2 \vdash e_s \Downarrow v, {\Gamma_2}'$ will also use the same rule, which is determined by the syntax of the expression (see Fig. 4). Say now $\langle \Gamma_1, e_s\rangle \rightsquigarrow \langle \Gamma_3, e'\rangle$ and $\langle \Gamma_2, e_s\rangle \rightsquigarrow \langle \Gamma_4, e'\rangle$. Firstly, the environment Γ_3 and Γ_4 are obtained from a prior big-step evaluation given by an antecedent of RULE, after possible updations to the store component. Let $\Gamma_3 \vdash e' \Downarrow \_, {\Gamma_3}'$. By Lemma 7 and the given facts, $\mathcal{C}_1 \sqsubseteq \mathcal{C}_3 \sqsubseteq \mathcal{C}_3' \sqsubseteq \mathcal{C}_1' \underset{\mathcal{H}_1', \mathcal{H}_2}{\lesssim} \mathcal{C}_2 \sqsubseteq \mathcal{C}_4$. Consider the store component of Γ_3, which is identical to ${\Gamma_3}'$, and Γ_4 namely σ_3 and σ_4. Any new mappings added to the store components depend on the prior big-step reductions in the antecedent of RULE, which satisfies the induction hypothesis. Thus, the new entries added are structurally similar. Hence, $\sigma_3' = \sigma_3 \underset{\mathcal{H}_3', \mathcal{H}_4}{\lesssim} \sigma_4$. Therefore, ${\Gamma_3}' \lesssim \Gamma_4$. By induction hypothesis, e' evaluates to structurally similar values in Γ_3 and Γ_4. Since this holds for every antecedent of RULE and since 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 $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ and $\Gamma_2 \vdash e_s \Downarrow v, {\Gamma_2}'$ produce structurally similar results. That is, $u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v$.

Now say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ uses the MEMOCALLMISS rule. In this case, since $\mathcal{C}_2$ has more entries than $\mathcal{C}_1$, $\Gamma_2 \vdash e_s \Downarrow v, {\Gamma_2}'$ will use the rule MEMOCALLHIT, as explained below. In this case, we know that $e_s = (f\ y)$ and $((f\ \sigma_1'(y)), u) \in \mathcal{C}_1'$. (Recall that the rule MEMOCALLMISS records the function value and the result of the evaluation in the cache.) Since $\mathcal{C}_1' \underset{\mathcal{H}_1', \mathcal{H}_1}{\lesssim} \mathcal{C}_2$, there exists an entry $(k, v) \in \mathcal{C}_2$ such that $((f\ \sigma_1'(y)) \underset{\mathcal{H}_1', \mathcal{H}_2}{\approx} k$ and $u \underset{\mathcal{H}_1', \mathcal{H}_2}{\approx} v$. Since $\sigma_1' \lesssim \sigma_2$, $\sigma_1'(y) \underset{\mathcal{H}_1', \mathcal{H}_2}{\approx} \sigma_2(y)$. Thus $(f\ \sigma_1'(y)) \underset{\mathcal{H}_1', \mathcal{H}_2}{\approx} (f\ \sigma_2(y))$. By the property of $\approx$, $(f\ \sigma_2(y)) \underset{\mathcal{H}_2}{\approx} k$. By the definition of MEMOCALLHIT, the result of the evaluation under Γ_2 is v. Since $\mathcal{H}_2 \sqsubseteq \mathcal{H}_2'$, $u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v$ which implies the claim. $\square$

Weak Cache Correctness. Consider the following property on an environment $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F)$ that states that every key in the cache is mapped to a value that it will evaluate to under Γ (if the key evaluates to any value at all).

$$\begin{aligned} &WeakCacheCorr(\Gamma) \triangleq \\ &\qquad \forall k \in dom(\mathcal{C}).\, (\Gamma \vdash k \Downarrow_p v, (\mathcal{C}', \mathcal{H}', \sigma', F')) \Rightarrow v \underset{\mathcal{H}'}{\approx} \mathcal{C}(k) \end{aligned}$$

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

Lemma 12. *For all expression e, For all* $\Gamma_1 : (\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F)$ *in* Env_P,

$$WeakCacheCorr(\Gamma_1) \wedge \Gamma_1 \vdash e \Downarrow u, {\Gamma_1}' \Rightarrow WeakCacheCorr({\Gamma_1}')$$

Proof. We prove the lemma using structural induction over the evaluation $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$. 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 $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ uses one of the inductive cases: LET, MATCH, CONCRETECALL, NONMEMOIZEDCALL, MEMOCALLMISS and CONTRACT. First, note that introducing new bindings to the store σ does not affect the property $WeakCacheCorr$, as the definition of $WeakCacheCorr$ does not use σ. This together with the inductive hypothesis imply that all the environments used in the antecedent of all the rules satisfy $WeakCacheCorr$. In all rules except MEMOCALLMISS the heap and cache components of the output environment are obtained directly from an antecedent. Therefore, by inductive hypothesis the environment Γ_1' satisfies the property $WeakCacheCorr$. Consider now the rule MEMOCALLMISS. Let $k \in FVal \cap dom(\mathcal{C}_1)$ be a key in the cache $\mathcal{C}_1$. Now, by the domain invariants, there exists a $\mathcal{H}_0 \sqsubseteq \mathcal{H}_1$ and $\mathcal{C}_0 \sqsubseteq \mathcal{C}_1$ such that $\Gamma_0 \vdash k \Downarrow u_0, \Gamma_0'$, where $u_0 = \mathcal{C}_1(k)$, $\Gamma_0 = (\mathcal{C}_0, \mathcal{H}_0, \{\})$, $\Gamma_0' = (\mathcal{C}_0', \mathcal{H}_0', \{\})$ and $\mathcal{C}_0' \sqsubseteq \mathcal{C}_1$. Since $\mathcal{C}_0' \sqsubseteq \mathcal{C}_1 \sqsubseteq \mathcal{C}_1'$ and $\{\} \sqsubseteq \sigma_1'$, $\Gamma_0' \lesssim \Gamma_1'$. Therefore, by Lemma 11,

$$\Gamma_1' \vdash k \Downarrow w, \Gamma_4 \Rightarrow u_0 \underset{\mathcal{H}_0', \mathcal{H}_4}{\approx} w$$

$$\Rightarrow \mathcal{C}_1(k) \underset{\mathcal{H}_1', \mathcal{H}_4}{\approx} w, \quad \text{since } \mathcal{C}_1(k) = u_0 \text{ and } \mathcal{H}_0' \sqsubseteq \mathcal{H}_1 \sqsubseteq \mathcal{H}_1'$$

$$\Rightarrow \mathcal{C}_1(k) = \mathcal{C}_1'(k) \underset{\mathcal{H}_4}{\approx} w, \text{ since } \mathcal{C}_1 \sqsubseteq \mathcal{C}_1' \text{ and } \mathcal{H}_1' \sqsubseteq \mathcal{H}_4$$

Therefore, $WeakCacheCorr(\Gamma_1')$

□

Cache Monotonicity. We now present the definition of *cache monotonicity*, which was also discussed in section 4. A boolean-valued expression e is cache monotonic iff $\forall \{\Gamma_1, \Gamma_2\} \subseteq Env$.

$$(\Gamma_1 \sqsubseteq \Gamma_2 \wedge \Gamma_1 \vdash e \Downarrow true) \Rightarrow \Gamma_2 \vdash e \Downarrow true$$

In the above definition, we can also substitute $\sqsubseteq$ by $\lesssim$. That is, a cache monotonic property satisfying the above definition is also monotonic with respect to the relation $\lesssim$ as state below.

Lemma 13. *Let e be a cache monotonic property. That is, $\forall \{\Gamma_1, \Gamma_2\} \subseteq Env$. $(\Gamma_1 \sqsubseteq \Gamma_2 \wedge \Gamma_1 \vdash e \Downarrow true) \Rightarrow \Gamma_2 \vdash e \Downarrow true$.*

$$(\Gamma_1 \lesssim \Gamma_2 \wedge \Gamma_1 \vdash e \Downarrow true) \Rightarrow \Gamma_2 \vdash e \Downarrow true$$

Proof. Let $\Gamma_1 = (\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F)$. Say $\Gamma_1 \vdash e \Downarrow true$ and $\Gamma_1 \lesssim \Gamma_2$. We now show that there exists a Γ_3 such that $\Gamma_2 \approx \Gamma_3$ and $\Gamma_1 \sqsubseteq \Gamma_3$. First define as mapping M from every address in $dom(\mathcal{H}_2)$ to a unique address not in $dom(\mathcal{H}_1)$. Let Γ_4 be the environment obtained by applying the mapping M to every address in every component of Γ_2. In other words, Γ_4 is obtained from Γ_2 by renaming addresses to not overlap with Γ_1. Clearly, $\Gamma_4 \approx \Gamma_2$ and hence $\Gamma_1 \lesssim \Gamma_4$. Define Γ_3 as $(\mathcal{C}_3, \mathcal{H}_3, \sigma_3, F)$, where

$$\mathcal{H}_3 = \mathcal{H}_1 \cup \mathcal{H}_4$$

$$\mathcal{C}_3 = \mathcal{C}_4 \setminus \{(k, v) \mid (k, v) \in \mathcal{C}_4 \wedge \exists k'. k' \underset{\mathcal{H}_1, \mathcal{H}_4}{\approx} k \wedge k' \in dom(\mathcal{C}_1)\} \cup \mathcal{C}_1$$

$$\sigma_3 = \sigma_4 \setminus \{(x, v) \mid x \in dom(\sigma_1)\} \cup \sigma_1$$

Clearly, by construction $\Gamma_1 \sqsubseteq \Gamma_3$. Now $\Gamma_4 \approx \Gamma_3$ because for every cache or store entry removed from Γ_4 we add a structurally similar entry to Γ_4 taken from Γ_1 which is $\lesssim \Gamma_4$. By cache monotonicity, $\Gamma_3 \vdash e \Downarrow true$. By Lemma 6, $\Gamma_2 \vdash e \Downarrow true$. Hence the claim. □

Referential Transparency and Cache-monotonicity requirement of Contracts. As mentioned earlier, the loss of referential transparency in our language is due to the presence of contracts that may use the specification construct cached. In order to guarantee full referential transparency of the source expressions, we impose the restriction that the contracts of memoized functions in the input programs should be *cache monotonic*. This property is soundly enforced by the model program defined in section 3 (as is proven later). The following theorem states that a stronger form of referential transparency holds with this restriction. The property guarantees that if a source expression evaluates to a value u at a point in the evaluation, then it *will* evaluate to a value v at a later point in the evaluation such that u and v are structurally similar. That is, memoization has absolutely no effect on the result of the function calls (which are source expressions).

Lemma 14. *Let $F \subseteq 2^{Fdef}$ be a set of function definitions such that for all* def f $x := \{p\}$ b $\{s\}$ *in F, p and s are cache monotonic properties. Let $\Gamma_1 : (\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F)$ in Env_P. For all expression $e_s \in (E_{src} \cup FVal)$, if $\Gamma_1 \vdash e_s \Downarrow u, \Gamma_1'$ then $\forall \Gamma_2 : (\mathcal{C}_2, \mathcal{H}_2, \sigma_2, F) \in Env_P$ s.t. $\Gamma_1' \lesssim \Gamma_2$,*

$$\Gamma_2 \vdash e_s \Downarrow v, \Gamma_2' \wedge u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v$$

Proof. The proof for this lemma is very similar to the proof of Lemma 11 except for the case of CONTRACT. Thus we only show the proof for this rule below. Say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, \Gamma_1'$ uses the rule CONTRACT. In this case, e_s is of the form $\{p\}$ e_b $\{s\}$. As per the language syntax, this means that e_s is the body of a function definition in F. It is given that p is cache monotonic. By the definition of the rule CONTRACT, $\Gamma_1 \vdash p \Downarrow true$. Since $\Gamma_1 \sqsubseteq \Gamma_1' \lesssim \Gamma_2$, by Lemma 13, $\Gamma_2 \vdash p \Downarrow true$. Since $e_b \in E_{src}$, by inductive hypothesis, $\Gamma_2 \vdash e_b \Downarrow v, \Gamma_2'$ and $u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v$. Now $\Gamma_1' \lesssim \Gamma_2 \sqsubseteq \Gamma_2'$. Since s is also cache monotonic, $\Gamma_1' \vdash s \Downarrow true$, which holds by the definition of CONTRACT, implies that $\Gamma_2' \vdash s \Downarrow true$. Hence, all antecedents of the rule CONTRACT are satisfied under Γ_2 for expression e_s. Hence $\Gamma_2 \vdash e_s \Downarrow v, \Gamma_2'$ and $u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v$. Hence the claim. □

Strong Cache Correctness. Analogous to the weak cache correctness property that was induced by weak referential transparency, we now establish that given a valid program P where function definitions have cache monotonic contracts, a stronger cache correctness property, defined below, follows from the strong referential transparency (Lemma 14).

$$CacheCorr(\Gamma) \triangleq \forall k \in dom(\mathcal{C}). (\Gamma \vdash k \Downarrow_p v, (\mathcal{C}', \mathcal{H}', \sigma', F)) \wedge v \underset{\mathcal{H}'}{\approx} \mathcal{C}(k)$$

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* def f $x := \{p\}$ b $\{s\}$ *in F, p and s are cache monotonic properties. For all expression e, for all $\Gamma_1 : (\mathcal{C}_1, \mathcal{H}_1, \sigma_1, F)$ in Env_P,*

$$CacheCorr(\Gamma_1) \wedge \Gamma_1 \vdash e \Downarrow u, \Gamma_1' \Rightarrow CacheCorr(\Gamma_1')$$

Proof. The proof of this lemma is very similar to the proof of Lemma 12, except for the rule MEMOCALLMISS. 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 MEMOCALLMISS.

Let $k \in FVal \cap dom(\mathcal{C}_1)$ be a key in the cache $\mathcal{C}_1$. By the domain invariants, there exists a $\mathcal{H}_0 \sqsubseteq \mathcal{H}_1$ and $\mathcal{C}_0 \sqsubseteq \mathcal{C}_1$ such that $\Gamma_0 \vdash k \Downarrow u_0, \Gamma_0'$, where $u_0 = \mathcal{C}_1(k)$, $\Gamma_0 = (\mathcal{C}_0, \mathcal{H}_0, \{\})$, $\Gamma_0' = (\mathcal{C}_0', \mathcal{H}_0', \{\})$ and $\mathcal{C}_0' \sqsubseteq \mathcal{C}_1$. Since $\mathcal{C}_0' \sqsubseteq \mathcal{C}_1 \sqsubseteq \mathcal{C}_1'$ and

$\{\} \sqsubseteq \sigma_1', {\Gamma_0}' \lesssim {\Gamma_1}'$. Therefore, by Lemma 14,

$${\Gamma_1}' \vdash k \Downarrow w, \Gamma_4 \wedge u_0 \underset{\mathcal{H}_0', \mathcal{H}_4}{\approx} w$$

$$\Rightarrow \mathcal{C}_1(k) \underset{\mathcal{H}_1', \mathcal{H}_4}{\approx} w, \quad \text{since } \mathcal{C}_1(k) = u_0 \text{ and } \mathcal{H}_0' \sqsubseteq \mathcal{H}_1 \sqsubseteq \mathcal{H}_1'$$

$$\Rightarrow \mathcal{C}_1(k) = \mathcal{C}_1'(k) \underset{\mathcal{H}_4}{\approx} w, \text{ since } \mathcal{C}_1 \sqsubseteq \mathcal{C}_1' \text{ and } \mathcal{H}_1' \sqsubseteq \mathcal{H}_4$$

Therefore, $CacheCorr({\Gamma_1}')$

□

In the rest of the section, we assume that every environment in Env_P also satisfy $WeakCacheCorr$ and $CacheCorr$ properties (if the contracts are cache monotonic), which are also like domain invariants that preserved by the semantics.

Contract Violation. Given an expression $\tilde{e}$ with contract: $\tilde{e} = \{p\}\ e\ \{s\}$. We say the contract of $\tilde{e}$ is violated under Γ iff $\Gamma \vdash p \Downarrow \mathit{false} \vee \exists \Gamma'.(\langle \Gamma, \tilde{e}\rangle \rightsquigarrow \langle \Gamma', s\rangle \wedge \Gamma' \vdash s \Downarrow \mathit{false})$.

Lemma 16. *Let e be an expression in a* type-correct *program P. Let $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env_P$ be such that $fv(e) \subseteq dom(\sigma)$. If $\neg \exists v. \Gamma \vdash e \Downarrow v$ then either e has a contract and is violated under Γ, or $\exists \Gamma', e'. \langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma', e'\rangle$ and $\neg \exists v. \Gamma' \vdash e' \Downarrow v$.*

Proof. Since the semantic rules shown in Fig. 4 cover every possible expression in the language, The expression e should match the expression in the consequent of one or more semantic rules. Say e matches a rule RULE.

Case (a): RULE is not CONTRACT. We now prove that every value accessed by RULE, except for those that are defined by big-step reductions, is defined.

Case (a.1): If RULE is not one of CONTRACT, PRIM, INDIRECTCALL, LET, or MATCH. Every other value required by RULE are either the output of a total function like *fresh* which is always defined, or $\sigma(x)$ or $\sigma(\mathcal{H}(x))$, where x is a free variable in e', σ and $\mathcal{H}$ are the store and heap components of Γ' (see Fig. 4). By definition, $fv(e) \subseteq dom(\sigma)$, and Γ satisfies all the domain invariants. Thus, both $\sigma(x)$ and $\sigma(\mathcal{H}(x))$ are defined.

Case (a.2): Say RULE is one of INDIRECTCALL, MATCH,PRIM or LET. In addition to requiring that $\sigma(x)$ or $\sigma(\mathcal{H}(x))$ are defined, these rule require more properties on shape (or type) of $\sigma(x)$ to be applicable. In the case of INDIRECTCALL, $\sigma(\mathcal{H}(x))$ is required to be a closure. In the case of MATCH, $\sigma(\mathcal{H}(x))$ is required to be a datatype with the constructors that are pattern matched in the MATCH construct. In the case of PRIM, $\sigma(x)$ should have the type of the argument of the primitive operation pr. (Recall that every primitive operation is total.) By definition, Γ satisfies all the domain invariants. Therefore, $\sigma(x)$ should inhabit the $type_P(x)$. Since we are given that the program P type checks, $type_P(x)$ will satisfy the above requirements in each of the rules. Hence, every value required by RULE that are not defined by big-step reductions will be defined.

If every big-step reduction $\Gamma' \vdash e' \Downarrow v$ in the antecedent of RULE produces *any* value, then clearly $\exists v. \Gamma \vdash e \Downarrow v$ (see Fig. 4). Since this is not the case, $\exists \Gamma', e'. \langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma', e'\rangle$ and $\neg \exists v. \Gamma' \vdash e' \Downarrow v$.

Case (b): Say RULE is CONTRACT. That is, e is an expression with contract i.e, $\{p\}\ e'\ \{s\}$. First, if the pre-or post-condition of e evaluates to *false*, then the contract of e is violated and hence the claim holds trivially. If e' evaluates to any value and p or s evaluate to $true$ then e evaluates to a value, contradicting the claim. Therefore, e' or p or s does not evaluate to a value. Hence the claim. □

Lemma 17. *Let e be an expression in a* type-correct *program P. Let $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env_P$ be such that $fv(e) \subseteq dom(\sigma)$. If $\neg \exists v. \Gamma \vdash e \Downarrow v$ then (a) there exists an infinite sequence $\langle \Gamma, e\rangle \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \cdots$,or (b) $\exists \Gamma' \in Env_P$ and an expression with contract $\tilde{e}$ such that $\langle \Gamma, e\rangle \rightsquigarrow^* \langle \Gamma', \tilde{e}\rangle$ and the contract of $\tilde{e}$ is violated under Γ'.*

Proof. It is easy to show by induction and Lemma 16 that $\forall n \in \mathbb{N}$, (a) $\exists \Gamma', e'. \langle \Gamma, e\rangle \rightsquigarrow^n \langle \Gamma', e'\rangle \wedge \neg \exists v. \Gamma' \vdash e \Downarrow v$ or (b) $\exists \tilde{e}. \langle \Gamma, e\rangle \rightsquigarrow^* \langle \Gamma', \tilde{e}\rangle$ and there is contract violation in $\tilde{e}$.

Say there is no infinite sequence starting from $\langle \Gamma, e\rangle$, otherwise the claim trivially holds. Therefore, $\exists n \in \mathbb{N}. \neg\left(\exists k > n, e, \Gamma'. \langle \Gamma, e\rangle \rightsquigarrow^k \langle \Gamma', e\rangle\right)$. This means that there does not exist a sequence of length $n+1$ such that $\langle \Gamma, e\rangle \rightsquigarrow^{n+1} \langle \Gamma', e'\rangle$. By the lemma 16, this implies that $\exists \tilde{e}. \langle \Gamma, e\rangle \rightsquigarrow^* \langle \Gamma', \tilde{e}\rangle$ and there is contract violation in $\tilde{e}$ under Γ'. □

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 MEMOCALLHIT 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.

MEMOCALLHIT2

$$\frac{f \in Mem_\Gamma \quad ((f\ \sigma(x)), v) \in_{\mathcal{H}} \mathcal{C} \qquad \Gamma \vdash (f\ \sigma(x)) \Downarrow^I_q u, (\mathcal{C}', \mathcal{H}', \sigma')}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash f\ x \Downarrow^I_{c_{hit}} u, (\mathcal{C}, \mathcal{H}', \sigma)}$$

The semantics I is an over-approximation of the operation semantics. That is, if the semantics I evaluates an expression e to a value u under and environment Γ, then the operation semantics also produces an equivalent value for the expression under the environment Γ, 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 $\{\Gamma_1, \Gamma_2\} \subseteq Env_P$ such that $\Gamma_1 \approx \Gamma_2$.*

$$\Gamma_2 \vdash e \Downarrow^I_p v, {\Gamma_2}' \Rightarrow \left(\Gamma_1 \vdash e \Downarrow_p u, {\Gamma_1}' \wedge u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v \wedge {\Gamma_1}' \approx {\Gamma_2}'\right)$$

Proof. We prove this using structural induction on the evaluation $\Gamma_2 \vdash e \Downarrow^I_p v, {\Gamma_2}'$. The base cases are the rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MEMOCALLHIT2 and CACHED. In all cases except MEMOCALLHIT2 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 $\Gamma_2 \vdash e \Downarrow^I_p v, {\Gamma_2}'$ uses 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 $\mathcal{C}_1 \underset{\mathcal{H}_1, \mathcal{H}_2}{\approx} \mathcal{C}_2$ and $((f\ \ \sigma_2(x)), v) \in_{\mathcal{H}_2} \mathcal{C}_2$ implies $((f\ \ \sigma_1(x)), v) \in_{\mathcal{H}_1} \mathcal{C}_2$. By the definition of the rules MEMOCALLHIT2 and MEMOCALLHIT, the cache and sigma components of ${\Gamma_1}'$ and ${\Gamma_2}'$ are identical to Γ_1 and Γ_2, respectively. Moreover, $\mathcal{H}_2 \sqsubseteq \mathcal{H}_2'$ and $\mathcal{H}_1 \sqsubseteq \mathcal{H}_1'$. Therefore, ${\Gamma_1}' \approx {\Gamma_2}'$. The resource usage $p = c_{hit}$ in both cases. We now show that $u \underset{\mathcal{H}_1', \mathcal{H}_2'}{\approx} v$.

By the definition of MEMOCALLHIT2, we know that $\Gamma_2 \vdash (f\ \sigma_2(x)) \Downarrow^I v, (\mathcal{C}', \mathcal{H}_2', \sigma')$ for some $\mathcal{C}'$ and σ'. By the inductive hypothesis, $\Gamma_1 \vdash (f\ \sigma_1(x)) \Downarrow u', (\mathcal{C}'', \mathcal{H}'', \sigma'')$, for some $\mathcal{C}''$, $\mathcal{H}''$ and σ', and $u' \underset{\mathcal{H}'', \mathcal{H}_2'}{\approx} v$. Since $WeakCacheCorr(\Gamma_1)$

holds, $u' \underset{\mathcal{H}''}{\approx} u$. Hence, by the properties of $\approx$, $u \underset{\mathcal{H}'',\mathcal{H}_2'}{\approx} v$. But since $u \in dom(\mathcal{H}_1)$ and $\mathcal{H}_1 = \mathcal{H}_1' \sqsubseteq \mathcal{H}''$, $u \underset{\mathcal{H}_1',\mathcal{H}_2'}{\approx} v$. 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 $\Gamma_1 \approx \Gamma_2$. $\square$

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 $\mathsf{def} f\ x := \tilde{e}$ be a function definition in P, If $\forall \Gamma \in Env_{\tilde{e},P}.\exists v.\ \Gamma \vdash p \Downarrow^I false \vee \Gamma \vdash \tilde{e} \Downarrow^I v$ then*
$\forall \Gamma \in Env_{\tilde{e},P}.\exists u.\ \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow u$

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

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 $\{\Gamma_1, \Gamma_2\} \subseteq Env_P$ such that $\Gamma_1 \approx \Gamma_2$.*

$$\Gamma_1 \vdash e \underset{p}{\Downarrow} v, {\Gamma_1}' \Rightarrow \left(\Gamma_2 \vdash e \underset{p}{\Downarrow^I} u, {\Gamma_2}' \wedge u \underset{\mathcal{H}_1',\mathcal{H}_2'}{\approx} v \wedge {\Gamma_1}' \approx {\Gamma_2}' \right)$$

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 $\forall k \in dom(\mathcal{C}).\ (\Gamma_1 \vdash k \Downarrow v, (\mathcal{C}', \mathcal{H}', \sigma', F)) \wedge v \underset{\mathcal{H}'}{\approx} \mathcal{C}(k)$. Therefore, there exists an evaluation among these, and $\Gamma_1 \vdash e \underset{p}{\Downarrow} v, {\Gamma_1}'$, 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 $\Gamma_1 \vdash e \underset{p}{\Downarrow} v, {\Gamma_1}'$ uses MEMOCALLHIT. The evaluation $\Gamma_2 \vdash e \underset{p}{\Downarrow} u, {\Gamma_2}'$ will use the rule MEMOCALLHIT2, since $\Gamma_1 \approx \Gamma_2$. Since $CacheCorr(\Gamma_1)$, $\Gamma_1 \vdash (f\ \sigma_1(x)) \Downarrow u', \Gamma'$ and $u \underset{\mathcal{H}'}{\approx} u'$. By inductive hypothesis, $\Gamma_2 \vdash (f\ \sigma_2(x)) \Downarrow^I v, \Gamma''$, and $u' \underset{\mathcal{H}',\mathcal{H}''}{\approx} v$. 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 Γ_1 and Γ_2, respectively. Moreover, $\mathcal{H}_2 \sqsubseteq \mathcal{H}_2'$ and $\mathcal{H}_1 \sqsubseteq \mathcal{H}_1'$. Therefore, ${\Gamma_1}' \approx {\Gamma_2}'$. The resource usage $p = c_{hit}$ in both cases.

From the above facts, $u' \underset{\mathcal{H}',\mathcal{H}''}{\approx} v$ and $u \underset{\mathcal{H}'}{\approx} u'$ and $\mathcal{H}'' = \mathcal{H}_2'$ (by the definition of MEMOCALLHIT2). Hence, $u \underset{\mathcal{H}',\mathcal{H}_2'}{\approx} v$ Since $u = \mathcal{C}_1(k) \in dom(\mathcal{H}_1)$, for some $k \underset{\mathcal{H}_1}{\approx} (f\ \sigma_2(x))$, and $\mathcal{H}_1 = \mathcal{H}_1' \sqsubseteq \mathcal{H}'$, $u \underset{\mathcal{H}_1',\mathcal{H}_2'}{\approx} v$. Hence the claim. $\square$

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 $\mathsf{def} f\ x := \tilde{e}$ be a function definition in P, If $\forall \Gamma \in Env_{\tilde{e},P}.\exists u.\ \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow v$ then*
$\forall \Gamma \in Env_{\tilde{e},P}.\exists v.\ \Gamma \vdash p \Downarrow^I false \vee \Gamma \vdash \tilde{e} \Downarrow^I v$

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

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 ${Env^\sharp}_{e,P^\sharp}$ for an expression e belonging to a model program $P^\sharp$. Recall that to define $Env_{e,P}$, 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 $P^\sharp$ is possible. However, since the cache in the model program is an expression of the model program, considering all possible clients of $P^\sharp$ is a 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 $P^\sharp$ using the valid environments of program P. 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 $\underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}}$ very similar to $\underset{\mathcal{H},\mathcal{H}^\sharp,P}{\sim}$ defined in section 3. Let $hash_\Gamma : Lam \mapsto \mathbb{N}$ be a function that maps structurally equal lambdas in Γ to the same natural number. That is,

$$\forall \{e_\lambda, {e_\lambda}'\} \subseteq range(\mathcal{H}).\ e_\lambda \underset{\mathcal{H}}{\approx} {e_\lambda}' \Rightarrow hash(e_\lambda) = hash({e_\lambda}')$$

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

$$dom_P(\mathcal{C}) = \{(f\ u) \in dom(\mathcal{C}) \mid f \text{ is defined in } P\}$$

Define a relation $\hat{\sim}$ between the semantic domains of the input and the model language as follows:

1.$\forall a \in \mathbb{Z} \cup Bool.\ a \mathrel{\hat{\sim}} a$
2.$\forall c \in Cids, \{\bar{a}, \bar{b}\} \subseteq Val^n.\ c\ \bar{a} \mathrel{\hat{\sim}} c\ \bar{b}$ *iff* $\forall i \in [1,n].a_i \mathrel{\hat{\sim}} b_i$
3.$\forall (e_\lambda, \sigma) \in Closure, v \in Val, l \in labels_P.(e_\lambda, \sigma) \mathrel{\hat{\sim}} C_l\ v$ *iff*
$\sigma(FV(e_\lambda)) \mathrel{\hat{\sim}} v \wedge (e_\lambda/_{\cong,P}$ is defined and has label $l)$
4.$\forall (e_\lambda, \sigma) \in Closure, v \in Val, l \in labels_P.$
$(e_\lambda, \sigma) \mathrel{\hat{\sim}} (C_{type_P(e_\lambda)}\ hash(e_\lambda))$ *iff* $e_\lambda/_{\cong,P}$ is undefined
5.$\forall f \in Fids$ defined in $P, \{a, b\} \subseteq Val.\ f\ a \mathrel{\hat{\sim}} C_f\ b$ *iff* $a \mathrel{\hat{\sim}} b$
6.$\forall \mathcal{C} \in Cache, S \in Set.\ \mathcal{C} \mathrel{\hat{\sim}} S$ *iff*
$|dom_P(\mathcal{C})| = |dom(S)| \wedge (\forall x \in dom_P(\mathcal{C}).\exists y \in S.x \mathrel{\hat{\sim}} y)$
7.$\forall \{a, b\} \subseteq Adr.\ a \mathrel{\hat{\sim}} b$ *iff* $\mathcal{H}(a) \mathrel{\hat{\sim}} \mathcal{H}^\sharp(a)$
8.$\forall \{\sigma, \sigma^\sharp\} \subseteq Store.\ \sigma \mathrel{\hat{\sim}} \sigma^\sharp$ *iff*
$dom(\sigma) \cup \{st\} = dom(\sigma^\sharp) \wedge \forall x \in dom(\sigma).\ \sigma(x) \mathrel{\hat{\sim}} \sigma^\sharp(x)$

Note that the only difference between $\sim$ and $\hat{\sim}$ is the rule 4, which is relates a closure not created within P to a constructor representing an error scenario. Let $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F) \in Env$ and $\Gamma^\sharp : (\mathcal{H}^\sharp, \sigma^\sharp, F') \in Env^\sharp$ and $S \in 2^{Val}$. As before,

$$\Gamma \mathrel{\hat{\sim}}_P (\Gamma^\sharp, S) \text{ iff } \mathcal{C} \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}} S, \sigma \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}} \sigma^\sharp \wedge F' = \{[\![d]\!]_P \mid d \in F \cap P\}$$

$$\frac{v \Leftrightarrow \left(\forall u \in \sigma(x).\exists u' \in \sigma(y).u \underset{\mathcal{H}}{\approx} u'\right)}{\Gamma^\sharp : (\mathcal{H}, \sigma) \vdash x \subseteq y \Downarrow v, \Gamma^\sharp}\ \textsc{Subset} \qquad \frac{v \Leftrightarrow \left(\exists u' \in \sigma(y).\sigma(x) \underset{\mathcal{H}}{\approx} u'\right)}{\Gamma^\sharp : (\mathcal{H}, \sigma) \vdash x \in y \Downarrow v, \Gamma^\sharp}\ \textsc{Contains}$$

$$\frac{v = \sigma(x) \cup \sigma(y)}{\Gamma^\sharp : (\mathcal{H}, \sigma) \vdash x \cup y \Downarrow v, \Gamma^\sharp}\ \textsc{Union} \qquad \frac{v = \{\sigma(x)\}}{\Gamma^\sharp : (\mathcal{H}, \sigma) \vdash \{x\} \Downarrow v, \Gamma^\sharp}\ \textsc{SetCons}$$

Figure 11. Semantics of set operations used by the model.

Let $\mathsf{def}\ f^\sharp\ (x, st) := \tilde{e}'$ be a function definition in $P^\sharp$ that is a translation of the definition $\mathsf{def}\ f\ x := \tilde{e}$ in P.

$$\begin{aligned} Env_{\tilde{e}', P^\sharp} = \{&\Gamma^\sharp : (\mathcal{H}^\sharp, \sigma^\sharp, F') \in Env^\sharp \mid \\ &\exists \Gamma \in Env_{\tilde{e}, P}.\Gamma \hat{\sim}_P (\Gamma^\sharp, \sigma^\sharp(st))\} \end{aligned} \tag{1}$$

We define ${Env^\sharp}_{e,P^\sharp}$ only if e is a body of a function in the model as the theorems that follow would need only these.

B.1 Soundness and Completeness Proofs.

In this section we detail the proofs of theorems stated in section 3 by establishing and utilizing several intermediate Lemmas. As before in all the formalism that follow if Γ_i (or Γ^i) is an environment then we refer to its individual components C of the environment, namely $(\mathcal{C}, \mathcal{H}, \sigma, F)$, using C_i (or C^i), respectively.

Lemma 22. *Let $\mathcal{H}_1, \mathcal{H}_2$ be two heaps and let P be a program. The relation $\underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}}$ is monotonic with respect to $\sqsubseteq$ on the heaps. That is, if $x \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}} y$, $\mathcal{H} \sqsubseteq \mathcal{H}_o$, and $\mathcal{H}^\sharp \sqsubseteq \mathcal{H}^\sharp_o$ then, $x \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o,P}{\hat{\sim}} y$.*

Proof. This can be established using straightforward structural induction over the definition of $\sim$. ☐

Lemma 23. *Let P be a program. Let u, v be two values and $\mathcal{H}, \mathcal{H}^\sharp$ be two heaps. The simulation relation $\underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}}$ is preserved by the structural equality relations $\underset{\mathcal{H}}{\approx}$ and $\underset{\mathcal{H}^\sharp}{\approx}$ and vice versa. That is, if $u \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}} v$ then $(u \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}} v' \Leftrightarrow v \underset{\mathcal{H}^\sharp}{\approx} v')$ and $(u' \underset{\mathcal{H},\mathcal{H}^\sharp,P}{\hat{\sim}} v \Leftrightarrow u \underset{\mathcal{H}}{\approx} u')$.*

Proof. We omit the subscripts of $\sim$ and $\approx$ in the rest of the proof. We show the proof for one part: if $u \sim v$ then $(u' \sim v \Leftrightarrow u \approx u')$. The proof of the other part is symmetric.

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 $(C\ \bar{w})$ i.e, $\mathcal{H}(u) = (C\ \bar{w})$. By the definition of $\approx$ and $\sim$, u' and v are also addresses of $(C\ \bar{w'})$ and $(C\ \bar{z})$ such that for all $i \in [1, |u|]$, $w_i \approx w'_i$ and $w_i \sim z_i$, respectively. By inductive hypothesis, $w'_i \sim z_i$. Hence, the claim.

Now say u is an address of a closure (e_λ, σ'). If $e_\lambda/_{\cong,P}$ is defined and has label l, $v = (C_l\ t)$ and $\sigma'(FV(e_\lambda)) \hat{\sim} t$. Since $u \approx u'$, $u' = ({e_\lambda}', \sigma'')$, ${e_\lambda}'/_{\cong,P} = e_\lambda/_{\cong,P}$ (by the definition of the $\cong$ relation) and $\sigma''(FV(e_\lambda)) \approx \sigma'(FV(e_\lambda))$. By induction hypothesis, $\sigma''(FV(e_\lambda)) \hat{\sim} t$. Hence, $u' \hat{\sim} v$.

If $e_\lambda/_{\cong,P}$ is not defined, $v = (C_{type_P(e_\lambda)}\ hash(e_\lambda))$. Since $u \approx u'$, $u' = ({e_\lambda}', \_)$ and ${e_\lambda}'/_{\cong,P}$ is not defined. By the definition of $hash$, $hash(e_\lambda) = hash({e_\lambda}')$. Hence, $u' \hat{\sim} v$.

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 $u = v = v' \in \mathbb{N} \cup Bool$. Say u is an address of $(C\ \bar{w})$ i.e, $\mathcal{H}(u) = (C\ \bar{w})$. By the definition of $\approx$ and $\sim$, u' and v are also addresses of $(C\ \bar{w'})$ and $(C\ \bar{z})$ such that for all $i \in [1, |u|]$, $w_i \sim z_i$ and $w'_i \sim z_i$, respectively. By inductive hypothesis, $w_i \approx w'_i$. Hence, the claim. The case where u is an address of a closure can be similarly proven. ☐

Lemma 24. *Let P be a program. Let $\Gamma \in Env_P$ and $\Gamma^\sharp \in {Env^\sharp}_P$ be such that $\Gamma \hat{\sim} (\Gamma^\sharp, S)$. $(\forall x \in dom_P(\mathcal{C}).\exists y \in S.\ x \hat{\sim} y)$ and $(\forall y \in S.\exists x \in dom_P(\mathcal{C}).\ x \hat{\sim} y)$.*

Proof. The first part of the claim follows by the definition of $\hat{\sim}$. That is, $(\forall x \in dom_P(\mathcal{C}).\exists y \in S.\ x \hat{\sim} y)$. By skolemization, the above implies that there exists a function $g : dom_P(\mathcal{C}) \to S$. We know that $|dom_P(\mathcal{C})| = |dom(S)|$ by the definition of $\hat{\sim}$. If g is injective, it should also be bijective and hence the claim holds. If g is non-injective, there exists an element $s \in S$ such that $x_1 \hat{\sim} s$ and $x_2 \hat{\sim} s$ for some $\{x_1, x_2\} \subseteq dom_P(\mathcal{C}) \subseteq dom(\mathcal{C})$ and $x_1 \neq x_2$. By Lemma 23, $x_1 \underset{\mathcal{H}}{\approx} x_2$. But by the domain invariants, every key in the cache is unique with respect to structural equality. Therefore, this case is not possible. ☐

Lemma 25. *Let P be a program. Let $\Gamma \in Env_P$ and $\Gamma^\sharp \in {Env^\sharp}_P$ be such that $\Gamma \hat{\sim} (\Gamma^\sharp, S)$. Let $x \in dom(\sigma)$ and $f \in Fids$ be a function defined in P. $\left(\exists u.(f\ u) \in dom_P(\mathcal{C}) \wedge u \underset{\mathcal{H}}{\approx} \sigma(x)\right)$ iff $\left(\exists u'.(C_f\ u') \in S \wedge u' \underset{\mathcal{H}^\sharp}{\approx} \sigma^\sharp(x)\right)$*

Proof. Consider the only if direction. Say $(f\ u) \in dom_P(\mathcal{C})$ and $u \underset{\mathcal{H}}{\approx} \sigma(x)$. By the definition of $\hat{\sim}$, $\exists y \in S.(f\ u) \sim y$. In other words, $\exists u'.(C_f\ u') \in S \wedge u \sim u'$. We are given that $\sigma(x) \sim \sigma^\sharp(x)$ and $\sigma(x) \underset{\mathcal{H}}{\approx} u$. By Lemma 23, $u \sim \sigma^\sharp(x)$. This together with the fact that $u \sim u'$ imply that $u' \underset{\mathcal{H}}{\approx} \sigma^\sharp(x)$. Hence, the claim. The other direction is symmetric (by Lemma 24). ☐

Correctness of the model programs for contract verification. Below we establish that if $\Gamma \hat{\sim} (\Gamma^\sharp, S)$, evaluating an expression e under Γ results in fewer crashes than evaluating the translation of e under Γ. That is, Γ progresses as long as $\Gamma^\sharp$ progresses on the translation of e.

Lemma 26. *Let P be a program. Let st be an expression of the model language. Let $\Gamma \in Env_P$ and $\Gamma^\sharp \in {Env^\sharp}_P$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \hat{\sim} (\Gamma^\sharp, S)$. Let e be any expression. If $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$ then $\exists \Gamma_o \in Env, v \in Val, p \in \mathbb{N}$ such that $\Gamma \vdash e \Downarrow^I_p v, \Gamma_o$ and*

- $\Gamma_o \hat{\sim} (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o,P}{\hat{\sim}} u._1$ • $p = u._3$

Proof. We prove this using structural induction over the evalution $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$.

Base cases. Say the evaluation $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$ uses one of the rules: Cst, Var, Prim, Equal, Cons, Lambda and Cached. Let $e' = \llbracket e \rrbracket_P\ st$. The free variables of e' and e are identical, and by the definition of $\hat{\sim}$, $fv(e) \subseteq dom(\sigma^\sharp) = dom(\sigma) \cup \{st\}$. 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, $\Gamma \vdash e \Downarrow^I_p v, \Gamma_o$ 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 c_{op} is a constant as per the semantics I, and is exactly same as $u._3$ as per the translation $\llbracket\cdot\rrbracket_P$. Therefore, $p = u._3$ holds trivially.

Consider now the claim: $\Gamma_o \hat{\sim} (\Gamma^\sharp_o, u._2)$. Recall that the relation $\sim$ is monotonic with respect to the ordering $\sqsubseteq$ between the heaps. In all the base cases, the cache and store components of the input and the output environments Γ and Γ_o are identical. The heaps of Γ and $\Gamma^\sharp$ are contained in the heaps of Γ_o and $\Gamma^\sharp_o$. Moreover, as per the translation, st and $u._2$ are also identical. Therefore, by Lemmas 7 and 22, $\Gamma \hat{\sim} (\Gamma^\sharp, S)$ directly implies $\Gamma_o \hat{\sim} (\Gamma^\sharp_o, u._2)$.

Consider now the claim: $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o}{\hat{\sim}} u._1$. In the case of CST it is easy to see that the values returned by e are identical primitive values (in $\mathbb{N} \cup Bool$) in both evaluations under Γ and $\Gamma^\sharp$. In the case of PRIM, the arguments of the operations are integer or boolean. By the definition of $\hat{\sim}$, the arguments are equal in both σ and $\sigma^\sharp$. Hence the output of PRIM is also equal under both environments. (We allow only deterministic primitive operations.) Therefore, $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o}{\hat{\sim}} u._1$ in both cases.

Consider the case of VAR. Say $\sigma(x) = a$ and $\sigma^\sharp(x) = a'$. It is given that $a \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} a'$. By definition, $v = a$ and $u._1 = a'$. Hence, the claim holds by Lemmas 7 and 22. In the case of CONS, $(a \mapsto cons\ \hat{\sigma}(\bar{x}))$ is added to $\mathcal{H}$ and $(a' \mapsto cons\ \hat{\sigma^\sharp}(\bar{x}))$ is added to $\mathcal{H}^\sharp$, for some fresh a and a' that are not bounded in $\mathcal{H}$ and $\mathcal{H}^\sharp$, respectively. It is given that $\sigma \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} \sigma^\sharp$. Therefore, $a \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} a'$ by the definition of $\hat{\sim}$, which by Lemma 22 implies $a \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o}{\hat{\sim}} a'$. Therefore, $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o}{\hat{\sim}} u._1$. The LAMBDA case can be similarly proved.

Consider now the CACHED case, i.e, e is $\mathsf{cached}(f\ x)$ (for some f and x). We are given that $\sigma(x) \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} \sigma^\sharp(x)$. By the definition of $\underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}}$, $(f\ \sigma(x)) \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} (C_f\ \sigma^\sharp(x))$, provided f is define 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, $\exists u'.(C_f\ u') \in S \wedge u' \underset{\mathcal{H}^\sharp}{\approx} \sigma^\sharp(x)$, where $\Gamma^\sharp \vdash st \Downarrow S$, if and only if $\exists u.(f\ u) \in dom(\mathcal{C}) \wedge u \underset{\mathcal{H}}{\approx} \sigma(x)$. By the semantics of set inclusion shown in Fig. 11 and $\in_{\mathcal{H}}$, $(C_f\ x) \in st$ evaluates to true under $\Gamma^\sharp$ 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 iff $\sigma(x) \underset{\mathcal{H}}{\approx} \sigma(y)$. It is given that $\sigma(x) \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} \sigma^\sharp(x)$ and $\sigma(y) \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} \sigma^\sharp(y)$. By Lemma 23, if $\sigma(x) \underset{\mathcal{H}}{\approx} \sigma(y)$ is true then $\sigma(y) \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} \sigma^\sharp(x)$, which in turn by the same lemma implies that $\sigma^\sharp(x) \underset{\mathcal{H}^\sharp}{\approx} \sigma^\sharp(y)$. Similarly, if $\sigma(x) \underset{\mathcal{H}}{\approx} \sigma(y)$ is false then by Lemma 23, $\neg(\sigma(y) \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} \sigma^\sharp(x))$ which in turn implies that $\neg(\sigma^\sharp(x) \underset{\mathcal{H}^\sharp}{\approx} \sigma^\sharp(y))$. Hence, the claim.

Proof of Inductive Step. Say the evaluation $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$ 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 listed above are the most interesting ones. The rest follow by inductive hypothesis.

Consider now the MEMOCALLHIT rule. In this case $p = u._3 = c_{hit}$. Also, $\Gamma_o \hat{\sim} (\Gamma^\sharp_o, u._2)$, since the output caches, state expressions and stores are identical to the input in both evaluations Γ and $\Gamma^\sharp$, and the output heaps are only larger. Consider now the claim: $v \underset{\mathcal{H},\mathcal{H}^\sharp}{\hat{\sim}} u._1$. Here, v and $u._1$ are the result of the evaluation $(f\ \sigma(x))$ (as per semantics I), and $(f\ \sigma^\sharp(x))$ under Γ and $\Gamma^\sharp$, respectively. Thus, by inductive hypothesis, $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o}{\hat{\sim}} u._1$. The proof for MEMOCALLMISS case is very similar, except that in this case $\mathcal{C}^\sharp_o$ is added a new entry $C_f\ \sigma^\sharp(x)$ (by the semantics of set union). However, $\mathcal{C}_o$ is also added a new entry $(f\ \sigma(x)) \mapsto v$. Since $(f\ \sigma(x)) \hat{\sim} C_f\ \sigma^\sharp(x)$ by definition, the claim that $\Gamma_o \hat{\sim} (\Gamma^\sharp_o, u._2)$ holds in this case as well.

Consider now the case INDIRECTCALL i.e, $e = (x\ y)^l$. The translated expression $\llbracket e \rrbracket_P\ st$ invokes the function App_l defined in Fig. 6. Let $\sigma(x) = (e_\lambda, \sigma')$.

Now say $e_\lambda/_{\cong,P}$ is not defined. By definition of $\hat{\sim}$, $\sigma^\sharp(x) = (C_{type_P(e_\lambda)}\ hash(e_\lambda))$. Therefore App_l with execute the error expression, and thus will crash. That is, $\neg\exists\Gamma^\sharp_o, u.\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$. Hence the claim trivially holds.

Now say $e_\lambda/_{\cong,P} = (\lambda x.f\ (x,z), \sigma')^l$, where $dom(\sigma') = \{z\}$. By definition of $\cong$, $target(e_\lambda) = f$. By the definition of $\hat{\sim}$, $\sigma^\sharp(x) = (C_l\ t)$ where $\sigma'(z) \hat{\sim} t$. By the definition of App_l (Fig. 6) and the match construct, $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$ reduces to $\Gamma^{\sharp\prime} \vdash (\llbracket f\ (y, y_i) \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$, where $\Gamma^{\sharp\prime} = (\mathcal{H}^\sharp, \sigma^\sharp \uplus (y_i \mapsto t))$. Now consider $\Gamma' = (\mathcal{C}, \mathcal{H}, \sigma \uplus \sigma')$. Clearly, $\Gamma' \hat{\sim} (\Gamma^{\sharp\prime}, \sigma^\sharp(st))$. Therefore, by induction hypothesis, $\Gamma' \vdash f\ (y,z) \Downarrow_p v, \Gamma_o$, $\Gamma_o \hat{\sim} (\Gamma^\sharp_o, S)$, $p = u._3$ and $u \underset{v}{\hat{\sim}}$. (Note that the variables y_i and z can be renamed to a variable say $r \notin dom(\sigma)$ 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 $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ and hence the claim holds.

□

Corollary 27. *Let P be a program. Let st be an expression of the model language. Let $\Gamma \in Env_P$ and $\Gamma^\sharp \in Env^\sharp{}_P$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \sim_P (\Gamma^\sharp, S)$. Let e be any expression. If $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$ then $\exists \Gamma_o \in Env, v \in Val, p \in \mathbb{N}$ such that $\Gamma \vdash e \Downarrow^I_p v, \Gamma_o$ and*

- $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o,P}{\sim} u._1$ • $p = u._3$

Proof. Notice that here the environments (and the states) are related by $\sim_P$ which is stronger than $\hat{\sim}$. Thus the only fact that is not implied by Lemma 26 is that $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$. 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 Γ with respect to the semantics I, and the evaluation of $\llbracket e \rrbracket_P\ st$ under $\Gamma^\sharp$ such that $\Gamma \hat{\sim} (\Gamma^\sharp, \sigma^\sharp(st))$ bisimulate each other, provided every indirect call invoked by the expression e during its evaluation under Γ 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 $\Gamma^\sharp \in Env^\sharp{}_P$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \hat{\sim} (\Gamma^\sharp, S)$. Let e be any expression such that if $\langle \Gamma, e \rangle \rightsquigarrow^* \langle \Gamma', x\ y \rangle$, $\mathcal{H}'(\sigma'(x)) = ({e_\lambda}^l, \sigma'')$ and $l \in labels_P$. If $\Gamma \vdash e \Downarrow^I_p v, \Gamma_o$ then $\exists \Gamma^\sharp_o \in Env^\sharp, u \in DVal$ such that $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$ and*

- $\Gamma_o \hat{\sim} (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o,\mathcal{H}^\sharp_o,P}{\hat{\sim}} u._1$ • $p = u._3$

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 $(x\ y)^{l'}$ (the rule INDIRECTCALL), we are guaranteed that $\sigma(x) = ({e_\lambda}^l, \sigma')$ and

$e_\lambda/_{\cong,P}$ is defined, since e_λ itself belongs to the program P. Thus, the evaluation of $([\![e]\!]_P\ st)$ under $\Gamma^\sharp$ cannot go through the error case of $App_{l'}$ function, which implies that $\Gamma^\sharp \vdash ([\![e]\!]_P\ st) \Downarrow u, \Gamma^\sharp_o$ will be defined for the rule INDIRECTCALL. $\square$

Corollary 29. *Let P be a program. Let st be an expression of the model language. Let $\Gamma \in Env_P$ and $\Gamma^\sharp \in Env^\sharp{}_P$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \sim_P (\Gamma^\sharp, S)$. Let e be any expression. If $\Gamma \vdash e \Downarrow^I_p v, \Gamma_o$ then $\exists \Gamma^\sharp_o \in Env^\sharp, u \in DVal$ such that $\Gamma^\sharp \vdash ([\![e]\!]_P\ st) \Downarrow u, \Gamma^\sharp_o$ and*
• $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o, \mathcal{H}^\sharp_o, P}{\sim} u._1$ • $p = u._3$

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 $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$. It is easy to see that this property holds from the proof argument of the above lemma. $\square$

Theorem 1.(*Bisimulation.*) *Let P be a program. Let st be an expression of the model language. Let $e' = [\![e]\!]_P\ st$. Let $\Gamma \in Env_P$ and $\Gamma^\sharp \in Env^\sharp{}_P$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \sim_P (\Gamma^\sharp, S)$.*

(a) If $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ then $\exists \Gamma^\sharp_o \in Env^\sharp, u \in DVal$ such that $\Gamma^\sharp \vdash e' \Downarrow u, \Gamma^\sharp_o$ and
• $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o, \mathcal{H}^\sharp_o, P}{\sim} u._1$ • $p = u._3$

(b) If $\Gamma^\sharp \vdash e' \Downarrow u, \Gamma^\sharp_o$ then $\exists \Gamma_o \in Env, v \in Val, p \in \mathbb{N}$ such that $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ and
• $\Gamma_o \sim_P (\Gamma^\sharp_o, u._2)$ • $v \underset{\mathcal{H}_o, \mathcal{H}^\sharp_o, P}{\sim} u._1$ • $p = u._3$

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. $\square$

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

$$\forall \Gamma^\sharp \in Env^\sharp{}_{\tilde{e}',P^\sharp}.\exists u.\ \Gamma^\sharp \vdash p' \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash \tilde{e}' \Downarrow u \ \ \mathit{iff}$$
$$\forall \Gamma \in Env_{\tilde{e},P}.\exists v.\ \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \tilde{e} \Downarrow v$$

Proof. Firstly, for every $\Gamma^\sharp \in Env^\sharp{}_{e,P}$ there exists an environment $\Gamma \in Env_{e,P}$ such that $\Gamma \mathrel{\hat{\sim}} (\Gamma^\sharp, \sigma^\sharp(st))$ and vice-versa. That is the relation $\hat{\sim}$ is total with respect to the domains $Env_{e,P}$ and $Env^\sharp{}_{e,P}$. This is because, by the definition of $Env^\sharp{}_{e,P}$, for every $\Gamma^\sharp \in Env^\sharp{}_{e,P}$ there exists an environment $\Gamma \in \Gamma e, P$ such that $\Gamma \mathrel{\hat{\sim}} (\Gamma^\sharp, \sigma^\sharp(st))$. For every $\Gamma \in Env_{e,P}$ we can construct an $\Gamma^\sharp \in Env^\sharp{}_{e,P}$ as follows:
(a) $\sigma^\sharp = \sigma \cup (st \mapsto S)$, where $S = \{(C_f\ u) \mid (f\ u) \in dom_P(\mathcal{C})\}$
(b) $\mathcal{H}^\sharp = \{(a, map(v)) \mid (a, v) \in \mathcal{H}\}$, where $map((e_\lambda, \sigma'))$ is $(C_l\ \ \sigma'(FV(e_\lambda)))$ if $e_\lambda/_{\cong,P}$ is defined and has label l, $map((e_\lambda, \sigma'))$ is $C_{type_P(e_\lambda)}\ hash(e_\lambda)$ if $e_\lambda/_{\cong,P}$ is not defined, and $map(v) = v$ otherwise.

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 $\hat{\sim}$ 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 (*soundness*) directly follows from Lemma 26 and the totality of $\hat{\sim}$ described above. Below, we prove the if direction (*completeness*). That is, if $\forall \Gamma \in Env_{\tilde{e},P}.\exists v.\ \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \tilde{e} \Downarrow v$ then $\forall \Gamma^\sharp \in Env^\sharp{}_{\tilde{e}',P^\sharp}.\exists u.\ \Gamma^\sharp \vdash p' \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash \tilde{e}' \Downarrow u$. Now, there are two cases to consider. If $\langle \Gamma, e\rangle \leadsto^* \langle \Gamma', c\ a\rangle$ implies $\mathcal{H}'(\sigma'(c)) = ({e_\lambda}^l, \sigma'') \wedge l \in labels_P$, then by Lemma 28 the claim holds.

Therefore, say $\langle \Gamma, e\rangle \leadsto^n \langle \Gamma', c\ a\rangle$, for some $n \in \mathbb{N}$, and $\mathcal{H}'(\sigma'(x)) = ({e_\lambda}^l, \sigma'') \wedge l \notin labels_P$. That is, the evaluation of e under Γ invokes a lambda created outside the program. Without loss of generality assume that $c\ a$ is the first such call. That is, every call reached before n steps is an encapsulated call. Now, it is easy to see that $\mathcal{H}'(\sigma'(c)) = \mathcal{H}(\sigma'(c))$. This is because if $\sigma'(c)$ is not bound in the input heap, it has to be bound subsequently. But we know that every expression that executes 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, $\sigma'(c)$ should be bound in the input heap. Let $\sigma'(c) = a$ and $\mathcal{H}(a) = (\lambda r.h\ (r, s), \sigma'')$. Now, consider a new environment Γ_{err} defined as follows: $\Gamma_{err} = (\mathcal{C}, \mathcal{H}[a \mapsto map(v)], \sigma, F \cup \{\mathsf{def}\ g\ t = \ \{\mathit{false}\}\ h\ t\ \{\mathit{true}\}\})$, where $map((\lambda r.h\ (r, s), \sigma'')) = (\lambda r.g\ (r, s), \sigma'')$, for some r, s and σ'', and $map(v) = v$ otherwise. That is, the new environment wraps the body of the lambdas *compatible* with $\mathcal{H}(a)$ by a contract whose precondition is *false*. By the totality of $\hat{\sim}$, $\exists \Gamma^\sharp \in Env^\sharp{}_{e,P}$ such that $\Gamma \mathrel{\hat{\sim}} (\Gamma^\sharp, \sigma^\sharp(st))$. Note that firstly (a) $\lambda r.h\ (r, s)/_{\cong,P}$ 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: $hash((\lambda r.g\ (r, s), \sigma'')) = hash((\lambda r.h\ \ (r, s), \sigma''))$. Clearly, this $hash$ function preserves structural equality, i.e, $\forall \{e_\lambda, {e_\lambda}'\} \subseteq range(\mathcal{H}_{err})$. $e_\lambda \underset{\mathcal{H}_{err}}{\approx} {e_\lambda}' \Rightarrow hash(e_\lambda) = hash({e_\lambda}')$, and hence is well-defined. Therefore, it is easy to see that $\Gamma_{err} \mathrel{\hat{\sim}} (\Gamma^\sharp, \sigma^\sharp(st))$ by our construction. Hence, $\langle \Gamma_{err}, e\rangle \leadsto^n \langle \Gamma'_{err}, c\ a\rangle$ and $\exists S.\Gamma' \mathrel{\hat{\sim}} (\Gamma'_{err}, S)$. Clearly, evaluating $(c\ a)$ under Γ'_{err} results in a contract violation as the precondition of g will not hold. Now, if $\Gamma_{err} \in Env_{e,P}$ 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 $\Gamma_{err} \in Env_{e,P}$

Since $\Gamma \in Env_{e,P}$, there exists a program P' such that $\langle \Gamma_{P' \| P}, e_{entry}\rangle \leadsto^* \langle \Gamma, e\rangle$. This implies that $\langle \Gamma_{P' \| P}, e_{entry}\rangle \leadsto^* \langle \Gamma_1, (f\ z)\rangle$, where f is the function whose contracts we are trying to verify, and $\Gamma_1 = (\mathcal{C}, \mathcal{H}, \sigma[z \mapsto \sigma(x)])$ 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 $\Gamma'_{err} = (\mathcal{C}_{err}, \mathcal{H}_{err}, \sigma_{err}[z \mapsto \sigma(x)])$, i.e, Γ_{err} before parameter translation. In our language, given a value w and a environment $\Gamma = (\mathcal{C}, \mathcal{H}, \sigma, F)$ it is possible to create an expression e such that $\Gamma \vdash e \Downarrow v, (\mathcal{C}, \mathcal{H}', \sigma)$ such that $v \underset{\mathcal{H}'}{\approx} w$. 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 $\mathcal{H}(a)$. 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_{err} that produces the value $\sigma_{err}(z)$. Since e_{err} is closed it can be inserted at any point in the program. We replace the call $(f\ z)$ by the expression let $z := e_{err}$ in $(f\ z)$. Let the new program thus obtained be called P_3. Thus, there exists a program P_3 such that $\langle \Gamma_{P_3 \| P}, e_{entry} \rangle \rightsquigarrow^* \langle \Gamma_{err}, e \rangle$. Hence, $\Gamma_{err} \in Env_{e,P}$.

$\square$

C. Correctness of Verification

Reducing* Error *construct to a precondition. Recall that the model programs use an error construct in the bodies of App_l functions corresponding to (non-encapsulated) indirect calls. Let def $App_l\ (cl, x, st)$ be one such function corresponding to an indirect call $(y\ z)$. The error construct will be encountered during the evaluation of App_l if and only if $cl = C_{type_P(y)}$. In this case, the result of the evaluation is undefined. The same effect can be achieved if we add a precondition to App_l namely $cl \neq C_{type_P(y)}$. It is obvious that the App_l with the precondition is equivalent to the App_l 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 App_l 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 $\models_P e_1 \to e_2$. As defined in section 4, let $e_1 \to e_2$ be

$$\forall \Gamma \in \{(\mathcal{C}, \mathcal{H}, \sigma, F) \in Env^\sharp \mid x \in dom(\sigma)\}$$
$$\Gamma \vdash e_1 \Downarrow false \vee \Gamma \vdash e_2 \Downarrow true$$

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 $fv(e)$ denote the set of free variables in the expression e. Let

$$Calls(\Gamma, e) = \{(\Gamma', (f\ x)) \mid \langle \Gamma, e \rangle \rightsquigarrow^* \langle \Gamma', (f\ x) \rangle\}$$

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

$$\bigwedge \{\exists v. \Gamma' \vdash (f\ x) \Downarrow v \mid (\Gamma', (f\ x)) \in Calls(\Gamma, e)\}$$

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 $\models_P e_1 \to e_2$ denotes the following:

$$\forall \Gamma \in \{(\mathcal{C}, \mathcal{H}, \sigma, F) \in Env^\sharp \mid fv(e_1) \cup fv(e_2) \subseteq dom(\sigma)\}.$$
$$\neg \mathcal{A}_P(\Gamma, e_1) \vee \neg \mathcal{A}_P(\Gamma, e_2) \vee \Gamma \vdash e_1 \Downarrow false \vee \Gamma \vdash e_2 \Downarrow true$$

Helper Functions. We define a few helper functions used by the assume/guarantee assertions. Given a function def $f\ x := \{p\}\ e\ \{s\} \in P$, let $pre_P(f\ x) = p[x/y]$. (We omit the subscript P when there is no ambiguity).

Path Condition and Reaching State. Recall that assume-guarantee rules use two expressions namely the path condition $path(c)$ and the state reaching a site $st(c)$, where c is call or construction site in the model program. Below we define the two expressions using the operational semantics. However, in our implementation they are statically computed from the program source (in the case of st this is somewhat trivial). We skip the formal details of how these expressions are satically computed, since here our focus is on the soundness of the assume-guarantee reasoning.

Let $c = (g\ x)^l$ be a call-site in a function definition def $f\ x := \tilde{e}$ in the model program. We define $path(c)$ as *any* boolean-valued expression that satisfies the following property:

$$\forall (\mathcal{H}, \sigma) \in Env^\sharp \text{ s.t. } fv(\tilde{e}) \subseteq dom(\sigma).$$
$$\langle \Gamma, \tilde{e} \rangle \rightsquigarrow^* \langle \Gamma', (g\ x)^l \rangle \Rightarrow \Gamma' \vdash path((g\ y)^l) \Downarrow true$$

That is, every environment that reaches the call-site makes the expression true.

Let e' be an expression within a function definition def $f\ (x, st) := \tilde{e}$ in the model program. If $e' = [\![e]\!]_P\ s$ for some expressions e and s, $st(e') = s$. Otherwise, if e'' is the smallest expression containing e' and of the form $[\![e]\!]_P\ s$, then $st(e') = s$. (Note that such an e'' always exist.) In other words, $st(e')$ is the state expression that reaches the expression e in the translation of $[\![\tilde{e}]\!]_P\ st$.

Lemma 30. *Consider the function-level, assume/guarantee rules shown below.*

- *For each def $g\ x = \{pre\}\ e\ \{post\}$ in P,*
  $\models_P pre \to post[e/res]$
- *For each call site $c = g\ x$ in P,*
  $\models_P path(c) \to pre(c)$

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

$$\forall (def\ f\ x := \tilde{e}) \in P \text{ s.t. } \tilde{e} = \{p\}\ e\ \{s\}.$$
$$\forall \Gamma \in \{(\mathcal{C}, \mathcal{H}, \sigma, F) \in Env^\sharp \mid x \in dom(\sigma)\}.$$
$$\left(\exists k > n, h \in Fids, y \in Vars, \Gamma'. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^k \langle \Gamma', (h\ y) \rangle\right)$$
$$\vee\ (\exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow v)$$

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 def $f\ x := \tilde{e} \in P$ where $\tilde{e} = \{p\}\ e\ \{s\}$, we need to prove that

$$\forall \Gamma \in \{(\mathcal{C}, \mathcal{H}, \sigma, F) \in Env^\sharp \mid x \in dom(\sigma)\}.$$
$$\left(\exists k > 0, h \in Fids, y \in Vars. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^k \langle \Gamma', (h\ y) \rangle\right)$$
$$\vee\ (\exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow v)$$

Consider a Γ such that $\neg\left(\exists k > 0, h, y. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^k \langle \Gamma', (h\ y) \rangle\right)$. Otherwise the claim trivially holds. This essentially means that we do not encounter a direct call either during the evaluation of p or $\tilde{e}$

under Γ. Therefore,

$$Calls(\Gamma, p) \cup Calls(\Gamma, \tilde{e}) = \emptyset \quad (2)$$

$$\Rightarrow \mathcal{A}_P(\Gamma, p) \wedge \mathcal{A}_P(\Gamma, \tilde{e}), \qquad \text{by the def. of } \mathcal{A}_P \quad (3)$$

$$\Rightarrow p \to s[e/\mathsf{res}], \quad \text{since} \models_P p \to s[e/\mathsf{res}] \quad (4)$$

$$\Rightarrow \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash s[e/\mathsf{res}] \Downarrow \mathit{true} \quad (5)$$

By the operational semantics of contract expressions Fig. 4,

$$\Rightarrow \exists v. \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \{\mathit{true}\}\ e\ \{s\} \Downarrow v \quad (6)$$

Since every call-free evaluation terminates in our language and by Lemma 17,

$$\Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash p \Downarrow \mathit{true} \quad (7)$$

By 6 and 7, $\exists v. \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \{p\}\ e\ \{s\} \Downarrow v$ (8)

Hence the claim holds in the base case.

Inductive step: Assume that the claim holds for all evaluations with m calls. We now show that the claim holds for all evaluations with $m+1$ calls. That is, we need to prove that

$$\forall \Gamma \in \{(\mathcal{C}, \mathcal{H}, \sigma, F) \in Env^\sharp \mid x \in dom(\sigma)\}.$$
$$\Big(\exists k > m+1, h \in Fids, y \in Vars, \Gamma'. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^k \langle \Gamma, (h\ y) \rangle\Big)$$
$$\vee\ (\exists v. \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \tilde{e} \Downarrow v)$$

As before, let us consider a Γ such that $\neg\big(\exists k > m+1, h, y, \Gamma'. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^k \langle \Gamma', (h\ y) \rangle\big)$. Otherwise the claim trivially holds. That is, all direct calls made by $\tilde{e}$ under Γ have depth at most $m+1$. Let S denote the top-level calls made by $\tilde{e}$. These are all calls that appear in the syntax tree of e. Formally,

$$S = \{(\Gamma', (g\ x)) \mid \exists i \in \mathbb{N}. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^i \langle \Gamma', (g\ x) \rangle$$
$$\wedge \neg \exists j < i, h. (\langle \Gamma, \tilde{e} \rangle \rightsquigarrow^j \langle \Gamma'', (h\ x) \rangle)\} \quad (9)$$

Note that by the definition of $\rightsquigarrow$, every call transitively made during the evaluation of $\tilde{e}$ should be reachable (w.r.t $\rightsquigarrow$) from the body of a callee in S in $\leq m$ depth (otherwise $\tilde{e}$ would invoke a call at a depth $> m+1$ violating the assumption). That is,

$$\forall (\Gamma', (g\ y)) \in S \text{ s.t. } \mathsf{def}\ f\ x := \{pre_g\}\ e_g\ \{post_g\} \in P.$$
$$\neg\Big(\exists i > m. \langle \Gamma'[x \mapsto \sigma'(y)], \{pre_g\}\ e_g\ \{post_g\} \rangle \rightsquigarrow^i \langle \Gamma'', (g\ x) \rangle\Big)$$

By inductive hypothesis the above implies that

$$\forall (\Gamma', (g\ y)) \in S \text{ s.t. } \mathsf{def}\ f\ x := \{pre_g\}\ e_g\ \{post_g\} \in P.$$
$$\exists v. \Gamma'[x \mapsto \sigma'(y)] \vdash pre_g \Downarrow \mathit{false}$$
$$\vee\ \Gamma'[x \mapsto \sigma'(y)] \vdash \{pre_g\}\ e_g\ \{post_g\} \Downarrow v$$

Based on the operational semantics and the definition of pre, the above can be rewritten as

$$\forall (\Gamma', (g\ y)) \in S. \exists v. \Gamma' \vdash pre(g\ y) \Downarrow \mathit{false} \vee \Gamma' \vdash (g\ y) \Downarrow v \quad (10)$$

As a consequence of the above fact we also know that every call invoked inside $pre(g\ y)$ terminates and results in a value. That is,

$$\forall (\Gamma', (g\ y)) \in S.\ \mathcal{A}_P(\Gamma', pre(g\ y)) \quad (11)$$

Now consider the definition of the path condition $path$ of a call $(g\ y)^l$ with label l contained in the body $\tilde{e}$ of a function f. By definition,

$$\forall \Gamma \in Env^\sharp. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^* \langle \Gamma', (g\ y)^l \rangle \Rightarrow \Gamma' \vdash path((g\ y)^l) \Downarrow \mathit{true} \quad (12)$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S.\ \Gamma' \vdash path(g\ y) \Downarrow \mathit{true} \quad (13)$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S.\ \mathcal{A}_P(\Gamma', path(g\ y)) \quad (14)$$

That is, every environment that reaches $(g\ y)$ will satisfy the path condition of $(g\ y)$. We are given that the following assertion holds:

$$\forall \text{ call-site c in P} . \models_P path(c) \to pre(c) \quad (15)$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S.\ \neg \mathcal{A}_P(\Gamma', path(g\ y)) \vee \neg \mathcal{A}_P(\Gamma', pre(g\ y))$$
$$\vee\ \Gamma' \vdash path(g\ y) \Downarrow \mathit{false} \vee \Gamma' \vdash pre(g\ y) \Downarrow \mathit{true} \quad (16)$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S.\ \Gamma' \vdash pre(g\ y) \Downarrow \mathit{true}, \quad \text{by 11, 13, 14} \quad (17)$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S. \exists v.\ \Gamma' \vdash (g\ y) \Downarrow v, \quad \text{by 10} \quad (18)$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in Calls(\Gamma, \tilde{e}). \exists v.\ \Gamma' \vdash (g\ y) \Downarrow v, \text{ by the def. of } Calls \quad (19)$$

$$\Rightarrow \mathcal{A}_P(\Gamma, \tilde{e}) \wedge \mathcal{A}_P(\Gamma, p) \quad (20)$$

Also, 19 implies that evaluations of p and $\tilde{e}$ terminates.

As in the base case, the above fact, 20 and $\models_P p \to s[e/\mathsf{res}]$ imply that

$$\exists v. \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \{p\}\ e\ \{s\} \Downarrow v \quad (21)$$

Hence, the claim. ☐

Lemma 31 (***Partial correctness of function-level, assume/guarantee reasoning***)**.** *Let* $\mathsf{def}\ f^\sharp\ x := \tilde{e}$ *where* $\tilde{e} = \{p\}\ e\ \{s\}$ *be a function definition in* $P^\sharp$. $\forall \Gamma \in Env^\sharp_{\tilde{e}, P^\sharp}$ *such that there exists no infinite sequence* $\langle \Gamma, \tilde{e} \rangle \rightsquigarrow \langle \Gamma', e' \rangle \rightsquigarrow \cdots$, $\exists u.\ \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \tilde{e} \Downarrow u$.

Proof. Let $\Gamma \in Env^\sharp_{\tilde{e}, P^\sharp}$. If there exists no infinite sequence $\langle \Gamma, \tilde{e} \rangle \rightsquigarrow \langle \Gamma', e' \rangle \rightsquigarrow \cdots$, then there exists a $n \in \mathbb{N}$ such that $\neg\big(\exists k > n, e, \Gamma'. \langle \Gamma, \tilde{e} \rangle \rightsquigarrow^k \langle \Gamma', e \rangle\big)$. We know that $\Gamma \in Env_{\tilde{e}, P}$ implies that $x \in dom(\sigma)$. Hence, by Lemma 30, $\exists u.\ \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \tilde{e} \Downarrow u$. ☐

Lemma 32 (***Soundness of function-level, assume/guarantee reasoning***)**.** *Let* $\mathsf{def}\ f^\sharp\ x := \tilde{e}$ *where* $\tilde{e} = \{p\}\ e\ \{s\}$ *be a function definition in* $P^\sharp$. *If every function defined in* $P^\sharp$ *terminate and satisfy the rules of function-level assume/guarantee reasoning, the contract of* $f^\sharp$ *holds i.e,* $\forall \Gamma \in Env^\sharp_{\tilde{e}, P^\sharp}. \exists u.\ \Gamma^\sharp \vdash p \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash \tilde{e} \Downarrow u$.

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. ☐

Soundness of Creation/dispatch reasoning. We now formalize and prove the soundness of the extended assume/guarantee reasoning based on creation and dispatch sites of encapsulated calls, and cache monotonic properties (shown below).

I. For each $\mathsf{def}\ f\ x := \{pre\}\ e\ \{post\}$, $\models_P pre \to post[e/\mathsf{res}]$

II. For each call site $c \notin DispCalls$, $\models_P path(c) \to pre(c)$

III. *(Cache monotonicity)* For each $\rho_i \in Props$
$\models_P (st_1 \subseteq st_2 \wedge [\![\rho_i]\!]_P\ st_1) \to [\![\rho_i]\!]_P\ st_2$

IV. For each closure construction site $c = C_i\ w_i$ in $Clo^\sharp$
$\models_P path(c) \to ([\![\rho_i]\!]_P\ st(c))$

V. For each call site $c = f_i^\sharp\ (x, z_i, st)$ in $DispCalls$
$\models_P (path(c) \wedge [\![\rho_i[z_i/y_i]]\!]_P\ st) \to pre(c)$

Similar to Lemma 30, we now 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 $([\![\rho_i]\!]_P\ st(c))$ is invoked just before the construction site $c = (C_i\ w_i)$, and that the result of ρ_i is ignored (including the state). That is, we replace $(C_i\ w_i)$ by $\mathsf{let}\ \_ := ([\![\rho_i]\!]_P\ st(c))\ \mathsf{in}\ (C_i\ w_i)$. 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 $P^\sharp$ the model program. If every function defined in $P^\sharp$ terminate and the assume/guarantee assertions (I) to (V) defined above hold, the following property holds for all $n \in \mathbb{N}$*

$$\begin{aligned}
&\forall\ \textit{program}\ P'.\forall \Gamma \in \textit{Env}.\\
&\quad \Big(\exists(\textbf{\textit{def}}\ f\ x := \tilde{e}) \in P, \Gamma^\sharp \in \textit{Env}^\sharp\ \textit{s.t.}\\
&\qquad \langle \Gamma_{P'\|P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle \wedge \Gamma \mathrel{\hat{\sim}} (\Gamma^\sharp, \sigma^\sharp(st)) \wedge\\
&\qquad \exists k > n, e', \Gamma'.\langle \Gamma^\sharp, [\![\tilde{e}]\!]_P\ st\rangle \rightsquigarrow^k \langle \Gamma', e'\rangle\Big) \vee\\
&\quad \forall \textbf{\textit{def}}\ f\ x := \tilde{e} \in P, \tilde{e} = \{p\}\ e\ \{s\}, \Gamma^\sharp \in \textit{Env}^\sharp.\\
&\qquad \textit{if}\ \big(\langle \Gamma_{P'\|P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle\big) \wedge \Gamma \mathrel{\hat{\sim}} (\Gamma^\sharp, \sigma^\sharp(st))\ \textit{then}\\
&\qquad \Big(\exists v.\Gamma^\sharp \vdash [\![p]\!]_P\ st \Downarrow \textit{false} \vee \Gamma^\sharp \vdash [\![\tilde{e}]\!]_P\ st \Downarrow v\Big)
\end{aligned}$$

Proof. We prove this by induction on n. Intuitively, n limits the depth of evaluation of any expression in the model program $P^\sharp$, during a run starting from the entry expression e_{entry}. Let P' be a client program. The base case is when $n = 1$. Consider a function definition $\mathsf{def}\, f\ x := \tilde{e}$. Let $e' = [\![\tilde{e}]\!]_P\ st$, and let $e' = \{p'\}\ b'\ \{s'\}$. Let $\langle \Gamma_{P'\|P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle$ and $\Gamma \mathrel{\hat{\sim}} (\Gamma^\sharp, \sigma^\sharp(st))$.

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

$$\begin{aligned}
&\textit{Calls}(\Gamma^\sharp, p') \cup \textit{Calls}(\Gamma^\sharp, e') = \emptyset\\
&\text{By } 2 - 8, \exists v.\Gamma \vdash p' \Downarrow \textit{false} \vee \Gamma \vdash e' \Downarrow v
\end{aligned}$$

Hence the claim holds in the base case.

Now, consider the inductive case and say the claim holds upto some number m. Now, if the evaluation of e' under $\Gamma^\sharp$ has depth more than m then the claim trivially holds. Therefore, say the evaluation of e' under $\Gamma^\sharp$ has depth at most $m + 1$.

(a) Say $\neg\exists c \in \textit{DispCalls}, \Gamma'.\langle e', \Gamma^\sharp\rangle \rightsquigarrow^* \langle c, \Gamma'\rangle$. In this case, for all $(c, \_) \in \textit{Calls}(\Gamma^\sharp, e')$, $\models_P \textit{path}(c) \rightarrow \textit{pre}(c)$ holds. Hence, by $9 - 21$, $\exists v.\Gamma \vdash p' \Downarrow \textit{false} \vee \Gamma \vdash e' \Downarrow v$.

(b) Therefore, say there exists a $c = g^\sharp\ (x, z, st) \in \textit{DispCalls}$ and $\Gamma^\sharp_3 \in \textit{Env}^\sharp$ such that $\langle e', \Gamma^\sharp\rangle \rightsquigarrow^* \langle c, \Gamma^\sharp_3\rangle$. Let $w = \sigma^\sharp_3(z)$. By the definition of *DispCalls* and the model translation, $\langle e', \Gamma^\sharp\rangle \rightsquigarrow^* \langle \textit{App}_l\ (y, a, st'), \Gamma^\sharp_1\rangle \rightsquigarrow^2 \langle \textit{App}_l\ (cl, x, st), \Gamma^\sharp_2\rangle$ $\rightsquigarrow^* \langle c, \Gamma^\sharp_3\rangle$ where $\sigma^\sharp_1(\mathcal{H}^\sharp_1(y)) = \sigma^\sharp_2(\mathcal{H}^\sharp_2(cl)) = (C_g\ w)$. By Lemma 26, $\langle \tilde{e}, \Gamma\rangle \rightsquigarrow^* \langle (y\ q), \Gamma_1\rangle$ and $\Gamma_1 \mathrel{\hat{\sim}} (\Gamma^\sharp_1, \sigma^\sharp_1(st'))$. Therefore, there exists an address a such that $\sigma_1(y) = a$, $\mathcal{H}_1(a) = ((\lambda x.g\ (x, p))^l, [p \mapsto v])$ and $v \mathrel{\hat{\sim}} w$.

By definition of *DispCalls* the call $(y\ q)$ is an encapsulated call. Therefore, $(\lambda x.g\ \ (x, p))^l$ belongs to the program P i.e, $l \in \textit{labels}_P$. Let "$\mathsf{def}\ cr\ x\ = \{\ e_b\ \}$" be the function in P that contains the lambda with label l. The closure $((\lambda x.g\ (x, p))^l, [p \mapsto v])$ should have been created at some point during the run starting from e_{entry}. Therefore, there exists a sequence $\langle \Gamma_{P'\|P}, e_{entry}\rangle \rightsquigarrow^* \langle \_, cr\ x\rangle \rightsquigarrow \langle \Gamma_0, e_b\rangle \rightsquigarrow^* \langle \Gamma'_0, \lambda x.g\ (x, p)\rangle$, $\Gamma'_0 \vdash \lambda x.g\ (x, p) \Downarrow a, (\mathcal{C}'_0, \mathcal{H}'_0[a \mapsto (\lambda x.g\ (x, p), [p \mapsto v])], \sigma'_0)$, $\mathcal{H}'_0 \sqsubseteq \mathcal{H}_1$ and $\mathcal{C}'_0 \sqsubseteq \mathcal{C}_1$.

Let ${\Gamma^\sharp}_0$ be such that $\Gamma_0 \mathrel{\hat{\sim}} ({\Gamma^\sharp}_0, \sigma^\sharp_0(st))$ and let $e'_b = [\![e_b]\!]_P\ st$.

Subclaim: forall $k \in \mathbb{N}$. $\forall \Gamma_{in} \in {\textit{Env}^\sharp}_{e_b,P}$ $\forall \Gamma^\sharp$ such that $\Gamma_{in} \mathrel{\hat{\sim}} ({\Gamma^\sharp}_{in}, \sigma^\sharp_{in}(st))$. If $\langle \Gamma_{in}, e_b\rangle \rightsquigarrow^k \langle \Gamma', e'\rangle$ then (a) there exists an chain $\langle {\Gamma^\sharp}_{in}, e'_b\rangle \rightsquigarrow^r \_$ and $r > m + 1$, or (b) $\exists s$. $\langle {\Gamma^\sharp}_{in}, e'_b\rangle \rightsquigarrow^* \langle \Gamma^{\sharp\prime}, [\![e']\!]_P\ s\rangle$ and $\Gamma' \mathrel{\hat{\sim}} (\Gamma^{\sharp\prime}, S)$ and $\Gamma^{\sharp\prime} \vdash s \Downarrow S$.

Proof. We prove this subclaim by induction on k. The inductive and base cases are very similar and so we prove them together as shown below. Let is and e be expressions and let $e' = ([\![e]\!]_P\ is)$ be the translation of e with respect to is. Let $\Gamma^\sharp$ be some expression such that $\langle \Gamma^\sharp, e\rangle$ is reachable from $\langle {\Gamma^\sharp}_{in}, e'_b\rangle$ and $\Gamma \mathrel{\hat{\sim}} (\Gamma^\sharp, S)$ and $\Gamma^\sharp \vdash is \Downarrow S$. Say $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_o, e_o\rangle$.

We now prove that one of the claim of the lemma holds. In the $\rightsquigarrow$ relations (shown in Fig. 10) introduced by all rules except LET and CONTRACT, the environments Γ and Γ_o 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 ${\Gamma^\sharp}_o$ such that $\langle \Gamma^\sharp, e'\rangle \rightsquigarrow \langle {\Gamma^\sharp}_o, [\![e_o]\!]_P\ is\rangle$ and $\Gamma_o \mathrel{\hat{\sim}} ({\Gamma^\sharp}_o, S)$ and ${\Gamma^\sharp}_o \vdash is \Downarrow S$.

Now consider the rule LET. Let $e = \mathsf{let}\ x\ :=\ e_1\ \mathsf{in}\ e_2$. By the definition of the translation: $e' = \mathsf{let}\ x\ :=\ [\![e_1]\!]_P\ is\ \mathsf{in}\ [\![e_2]\!]_P\ x.{}_2$ By definition, there are two $\rightsquigarrow$ relations introduced by the rule. Consider the relations: $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma, e_1\rangle$ and $\langle \Gamma^\sharp, e'\rangle \rightsquigarrow \langle \Gamma^\sharp, [\![e_1]\!]_P\ is\rangle$. These clearly satisfy the claim.

Consider the other relation defined as follows: If $\Gamma \vdash e_1 \Downarrow u_1, \Gamma_1$ then $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_o, e_2\rangle$, where $\Gamma_o = (\mathcal{C}_1, \mathcal{H}_1, \sigma_1[x \mapsto u_1])$. We also have similar relation for the translated expression. If $\Gamma^\sharp \vdash [\![e_1]\!]_P\ st \Downarrow v, {\Gamma^\sharp}_1$ then $\langle \Gamma^\sharp, e'\rangle \rightsquigarrow \langle {\Gamma^\sharp}_o, [\![e_2]\!]_P\ x.{}_2\rangle$, where ${\Gamma^\sharp}_o = (\mathcal{H}^\sharp_1, \sigma^\sharp_1[x \mapsto v_1])$. Now there are two cases to consider

(a) There exists a chain $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st\rangle \rightsquigarrow^r \_$ and $r > m$. In this case, there exists an chain $\langle {\Gamma^\sharp}_{in}, e'_b\rangle \rightsquigarrow^r \_$ and $r > m+1$, since are given that $\langle \Gamma^\sharp, e'\rangle$ is reachable from $\langle {\Gamma^\sharp}_{in}, e'_b\rangle$. Hence the claim holds.

(b) There does not exist a chain $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st\rangle \rightsquigarrow^r \_$ and $r > m$. Now, we claim that $\exists v_1.\ \Gamma^\sharp \vdash [\![e_1]\!]_P\ st \Downarrow v_1, {\Gamma^\sharp}_1$. This is because, by Lemma 17, the the evaluation could be undefined only if either the evaluation does not terminate or because there is a contract violation during the evaluation The former case is not possible since there does not exist a chain $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st\rangle \rightsquigarrow^r \_$ and $r > m$. The latter case is not possible because every call $(h\ g)$ encountered during the evaluation (under an environment $\Gamma^{\sharp\prime\prime}$) cannot have a chain $\langle \Gamma^{\sharp\prime\prime}, (h\ g)\rangle \rightsquigarrow^r \_$ and $r > m$ (otherwise $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st\rangle \rightsquigarrow^r \_$ 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 have now shown that $\Gamma^\sharp \vdash [\![e_1]\!]_P\ st \Downarrow v_1, {\Gamma^\sharp}_1$ is defined. Therefore, $\langle \Gamma^\sharp, e'\rangle \rightsquigarrow \langle {\Gamma^\sharp}_o, [\![e_2]\!]_P\ x.{}_2\rangle$ is defined. By Lemma 26, $\Gamma \vdash e \Downarrow u_1, \Gamma_1$ is defined. Thus, $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_o, e_2\rangle$ is also defined and $\Gamma_o \mathrel{\hat{\sim}} ({\Gamma^\sharp}_o, S')$ and ${\Gamma^\sharp}_o \vdash x.{}_2 \Downarrow S$ Hence, the claim holds. The rule contract can be similarly proven. □

With this above established claim let us again revisit the evaluation: $\langle \Gamma_0, e_b\rangle \rightsquigarrow^* \langle \Gamma'_0, (\lambda x.g\ (x, p))^l\rangle$, $\Gamma'_0 \vdash \lambda x.g\ (x, p) \Downarrow a, (\mathcal{C}'_0, \mathcal{H}'_0[a \mapsto (\lambda x.g\ (x, p), [p \mapsto v])], \sigma'_0)$, $\mathcal{H}'_0 \sqsubseteq \mathcal{H}_1$ and $\mathcal{C}'_0 \sqsubseteq \mathcal{C}_1$. Due to the above subclaim, we know that one of the following cases hold: (a) either there exists $\langle {\Gamma^\sharp}_0, e'_b\rangle \rightsquigarrow^r \_$ and $r > m + 1$. or (b) $\langle {\Gamma^\sharp}_0, e'_b\rangle \rightsquigarrow^* \langle {\Gamma^{\sharp\prime}}_0, (C_l\ p)\rangle$ and $\exists s.\Gamma'_0 \mathrel{\hat{\sim}} ({\Gamma^{\sharp\prime}}_0, S)$ and ${\Gamma^{\sharp\prime}}_0 \vdash s \Downarrow S$. In the former case the lemma holds as the first disjunct of the lemma is satisfied as ${\Gamma^\sharp}_0$ belongs to ${\textit{Env}^\sharp}_{e',P^\sharp}$. Therefore consider the latter case.

Let $cc = (C_l\ p)$. By definition, $st(cc)$ is the state expression reaching the construction site cc. Therefore $s = st(cc)$. By the definition of *path*, for any function definition $\mathsf{def}\, f\ x := e'_b$ and closure construction site cc in f.

$$\forall \Gamma^\sharp \in \textit{Env}^\sharp.\langle \Gamma^\sharp, e'_b\rangle \rightsquigarrow^* \langle \Gamma^{\sharp\prime}, cc\rangle \Rightarrow \Gamma^{\sharp\prime} \vdash \textit{path}(cc) \Downarrow \textit{true} \tag{22}$$

$$\text{Therefore, } {\Gamma^{\sharp\prime}}_0 \vdash \textit{path}(cc) \Downarrow \textit{true} \tag{23}$$

$$\Rightarrow \mathcal{A}_P({\Gamma^{\sharp\prime}}_0, \textit{path}(cc)) \tag{24}$$

Let $\rho' = (\llbracket \rho_i \rrbracket_P\ st(c))$. We know that the $fv(\rho_i) \subseteq \{p\}$, where p is argument of the constructor (captured variable). Now, by the assume-guarantee reasoning rule (IV), we are given that

$$\models_P path(cc) \to \rho' \tag{25}$$

$$\Rightarrow \forall \Gamma^{\sharp\prime}.\neg \mathcal{A}_P(\Gamma^{\sharp\prime}, path(cc)) \vee \Gamma^{\sharp\prime} \vdash path(cc) \Downarrow false$$

$$\vee \neg \mathcal{A}_P(\Gamma^{\sharp\prime}, \rho') \vee \Gamma^{\sharp\prime} \vdash \rho' \Downarrow true \tag{26}$$

$$\Rightarrow \neg \mathcal{A}_P(\Gamma^{\sharp\prime}_0, \rho') \vee \Gamma^{\sharp\prime}_0 \vdash \rho' \Downarrow true, \quad \text{by 23, 24} \tag{27}$$

Now recall that we have assumed that ρ' is invoked just before the closure construction cc. Therefore, $\exists \Gamma^\sharp \sqsubseteq \Gamma^{\sharp\prime}_0$ such that $\exists v. \Gamma^\sharp \vdash \rho' \Downarrow v$, since we are given that $\langle \Gamma^\sharp_0, e_b' \rangle \rightsquigarrow^* \langle \Gamma^{\sharp\prime}_0, cc \rangle$. Hence, $\mathcal{A}_P(\Gamma^\sharp, \rho')$ holds. It is easy to see that, $Calls(\Gamma^{\sharp\prime}_0, \rho') = Calls(\Gamma^\sharp, \rho')$ since $\Gamma^\sharp \sqsubseteq \Gamma^{\sharp\prime}_0$. (Note that the model program does not have memoization and is purely functional.) Therefore, $\mathcal{A}_P(\Gamma^{\sharp\prime}_0, \rho')$ also holds. Substituting this in 27 we get, $\Gamma^\sharp{}_0{}' \vdash \rho' \Downarrow true$. By Lemma 26, $\Gamma_0' \vdash \rho_i \Downarrow true$.

Now, we know that $\mathcal{H}_0' \sqsubseteq \mathcal{H}_1$ and $\mathcal{C}_0' \sqsubseteq \mathcal{C}_1$. We are also given by the assume-guarantee rule (III) that

$$\models_P st_1 \subseteq st_2 \wedge \llbracket \rho_i \rrbracket_P\ st_1 \to \llbracket \rho_i \rrbracket_P\ st_2 \tag{28}$$

$$\Rightarrow \forall \Gamma^\sharp \text{ s.t. } dom(\sigma^\sharp) \subseteq fv(\rho_i) \cup \{st_1, st_2\}.$$

$$\neg \mathcal{A}_P(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_1) \vee \mathcal{A}_P(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2)$$

$$\vee \Gamma^\sharp \vdash (st_1 \subseteq st_2) \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_1 \Downarrow false$$

$$\vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_2 \Downarrow true \tag{29}$$

$$\Rightarrow \forall \Gamma^\sharp \text{ s.t. } \Gamma^{\sharp\prime}_0 \sqsubseteq \Gamma^\sharp \wedge st_2 \in dom(\sigma^\sharp).$$

$$\Gamma^\sharp \vdash (st(cc) \subseteq st_2) \Downarrow false \vee$$

$$\neg \mathcal{A}_P(\Gamma^\sharp, \rho') \vee \mathcal{A}_P(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2)$$

$$\vee \Gamma^\sharp \vdash \rho' \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_2 \Downarrow true \tag{30}$$

By the definition of the model programs, the depths of the evaluations of the expressions of the model program are independent of the state parameter. Recall that as shown by Fig. 6 the state parameter only influences the value of the last element of the tuple, namely the resource usage component. Therefore, $Calls(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2) = Calls(\Gamma^\sharp, \rho')$. We are given that $\mathcal{A}_P(\Gamma^{\sharp\prime}_0, \rho')$ holds. Therefore as $\Gamma^{\sharp\prime}_0 \sqsubseteq \Gamma^\sharp$, $\mathcal{A}_P(\Gamma^\sharp, \rho')$ and $\mathcal{A}_P(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2)$ also holds. Substituting this and the fact that $\Gamma_0' \vdash \rho_i \Downarrow true$ in 30 we get,

$$\forall \Gamma^\sharp \text{ s.t. } \Gamma^{\sharp\prime}_0 \sqsubseteq \Gamma^\sharp \wedge st_2 \in dom(\sigma^\sharp).$$

$$\Gamma^\sharp \vdash (st(cc) \subseteq st_2) \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_2 \Downarrow true \tag{31}$$

By Lemma 26 and the totality of $\hat{\sim}$ relation,

$$\forall (\mathcal{C}_1, \mathcal{H}_0', \sigma_0'). \neg(\mathcal{C}_0' \sqsubseteq \mathcal{C}_1) \vee (\mathcal{C}_1, \mathcal{H}, \sigma) \vdash \rho_i \Downarrow true \tag{32}$$

$$\Rightarrow \forall \Gamma'. \neg(\Gamma_0' \sqsubseteq \Gamma') \vee \Gamma' \vdash \rho_i \Downarrow true \tag{33}$$

Since we know $\mathcal{C}_0' \sqsubseteq \mathcal{C}_1$ and $\mathcal{H}_0' \sqsubseteq \mathcal{H}_1$. The above implies that

$$(\mathcal{C}_1, \mathcal{H}_1, \sigma_0') \vdash \rho_i \Downarrow true \tag{34}$$

$$(\mathcal{C}_1, \mathcal{H}_1, [p \mapsto \sigma_0'(p)]) \vdash \rho_i \Downarrow true, \quad \text{since } fv(\rho_i) \subseteq \{p\} \tag{35}$$

We are given that $\sigma_0'(p) = v$, $v \underset{\mathcal{H}_1, \mathcal{H}_3^\sharp}{\hat{\sim}} w$, $\sigma_3^\sharp(z) = w$, $\mathcal{C}_1 \underset{\mathcal{H}_1, \mathcal{H}_1^\sharp}{\hat{\sim}} \sigma_1^\sharp(st')$, $\sigma_1^\sharp(st') = \sigma_3^\sharp(st)$ and $\mathcal{H}_1^\sharp \sqsubseteq \mathcal{H}_3^\sharp$. The last three facts imply that $\mathcal{C}_1 \underset{\mathcal{H}_1, \mathcal{H}_3^\sharp}{\hat{\sim}} \sigma_3^\sharp(st)$. Hence,

$$(\mathcal{C}_1, \mathcal{H}_1, [p \mapsto \sigma_0'(p)]) \underset{\mathcal{H}_1, \mathcal{H}_3^\sharp}{\hat{\sim}} ((\mathcal{H}_3^\sharp, [p \mapsto \sigma_3^\sharp(z)]), \sigma_3^\sharp(st))$$

Therefore, by Lemma 26 and 35,

$$(\mathcal{H}_3^\sharp, [p \mapsto \sigma_3^\sharp(z)]) \vdash \llbracket \rho_i[z/p] \rrbracket_P\ st \Downarrow true \tag{36}$$

$$\Gamma^\sharp{}_3 \vdash \llbracket \rho_i[z/p] \rrbracket_P\ st \Downarrow true \tag{37}$$

Now, we are given that $\langle e', \Gamma^\sharp \rangle \rightsquigarrow^* \langle c, \Gamma_3^\sharp \rangle$, where $e' = \llbracket \tilde{e} \rrbracket_P\ st$ By assume-guarantee assertion (V),

$$\models_P (path(c) \wedge \llbracket \rho_i[z_i/y_i] \rrbracket_P\ st) \to pre(c) \tag{38}$$

$$\Rightarrow \Gamma^\sharp{}_3 \vdash pre(c) \Downarrow true, \tag{39}$$

by the reasoning shown in 16 – 18 and 37

Therefore, for every $(\Gamma^{\sharp\prime}, c) \in DispCalls \cap Calls(\Gamma^\sharp, e')$, $\Gamma^{\sharp\prime} \vdash pre(c) \Downarrow true$. By the reasoning shown by 18 and 37, for every $(\Gamma^{\sharp\prime}, c) \in Calls(\Gamma^\sharp, e') \setminus DispCalls$, $\Gamma^{\sharp\prime} \vdash pre(c) \Downarrow true$. Therefore, as shown by 19 – 21, $\mathcal{A}_P(\Gamma^\sharp, e')$ and $\mathcal{A}_P(\Gamma^\sharp, p')$. Hence, by the assume-guarantee assertion (I), $\exists v. \Gamma^\sharp \vdash \llbracket p \rrbracket_P\ st \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \tilde{e} \rrbracket_P\ st \Downarrow v$. ☐

Theorem 3.(***Soundness of creation-dispatch reasoning***) *Let P be a program and $P^\sharp$ the model program. Let* $\mathsf{def}\, f^\sharp\ x := \tilde{e}$ *where $\tilde{e} = \{p\}\ e\ \{s\}$ be a function definition in $P^\sharp$. If every function defined in P terminate and the assume/guarantee assertions (I) to (V) defined above hold, the contracts of $f^\sharp$ holds i.e, $\forall \Gamma^\sharp \in Env^\sharp{}_{\tilde{e}, P^\sharp}. \exists u.\ \Gamma^\sharp \vdash p \Downarrow false \vee \Gamma^\sharp \vdash \tilde{e} \Downarrow u$.*

Proof. Let $\mathsf{def}\ g^\sharp\ x := \tilde{e}'$ be a function definition in $P^\sharp$, where $\tilde{e}' = \llbracket e \rrbracket_P\ st$. Let $\Gamma^\sharp \in Env^\sharp{}_{\tilde{e}', P^\sharp}$. By definition, there exists a $\Gamma \in Env$ and a program P' such that $\big(\langle \Gamma_{\mathrm{P'} \| \mathrm{P}}, e_{entry} \rangle \rightsquigarrow^* \langle \Gamma, \tilde{e} \rangle\big) \wedge \Gamma \hat{\sim} (\Gamma^\sharp, \sigma^\sharp(st))$. Also, by definition of $Env_{e,P}$, $\langle \Gamma_{\mathrm{P'} \| \mathrm{P}}, e_{entry} \rangle$ is a terminating evaluation given that all functions in the program P are terminating. Now say there exists an infinite chain of the form $\langle \Gamma^\sharp, \llbracket e \rrbracket_P\ st \rangle \rightsquigarrow \langle \Gamma^\sharp{}_1, \llbracket e_1 \rrbracket_P\ s_1 \rangle \rightsquigarrow \cdots$. By Lemma 26 and the definition of $\rightsquigarrow$ (Fig. 10), there exists an infinite chain: $\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma_1, e_1 \rangle \rightsquigarrow \cdots$, which is a contradiction to the given fact that the evaluation $\langle \Gamma_{\mathrm{P'} \| \mathrm{P}}, e_{entry} \rangle$ is terminating. Therefore, there cannot be any infinite chains of the form: $\langle \Gamma^\sharp, \llbracket e \rrbracket_P\ st \rangle \rightsquigarrow \langle \Gamma^\sharp{}_1, \llbracket e_1 \rrbracket_P\ s_1 \rangle \rightsquigarrow \cdots$. Hence, the evaluation of $\langle \Gamma^\sharp, \llbracket e \rrbracket_P\ st \rangle$ is terminating

By the same argument, for every other function $\mathsf{def}\ h^\sharp\ x := \llbracket \tilde{e}_h \rrbracket_P\ st \in P^\sharp$ there does not exist a $\Gamma^\sharp{}_h$ such that $\langle \Gamma_{\mathrm{P'} \| \mathrm{P}}, e_{entry} \rangle \rightsquigarrow^* \langle \Gamma_h, \tilde{e}_h \rangle$ and $\Gamma_h \hat{\sim} (\Gamma^\sharp{}_h, \sigma_h^\sharp(st))$ and $\langle \Gamma^\sharp{}_h, \llbracket \tilde{e}_h \rrbracket_P\ st \rangle \rightsquigarrow \cdots$ is infinite. Thus, there exists a $n \in \mathbb{N}$ such that $\neg \exists \mathsf{def}\ h\ x := \tilde{e}_h \in P, \Gamma^\sharp{}_h \in Env^\sharp$ s.t. $\langle \Gamma_{\mathrm{P'} \| \mathrm{P}}, e_{entry} \rangle \rightsquigarrow^* \langle \Gamma_h, \tilde{e} \rangle \wedge \Gamma \hat{\sim} (\Gamma^\sharp{}_h, \sigma_h^\sharp(st)) \wedge \exists k > n, e', \Gamma'. \langle \Gamma^\sharp{}_h, \llbracket \tilde{e} \rrbracket_P\ st \rangle \rightsquigarrow^k \langle \Gamma', e \rangle$. Hence, by Lemma 33, $\exists u.\ \Gamma^\sharp \vdash p \Downarrow false \vee \Gamma^\sharp \vdash \tilde{e} \Downarrow u$ for every function definition in $P^\sharp$. Hence, the contracts of the function $f^\sharp$ also holds. ☐

C.2 Decidability of Inference Algorithm

Theorem 4. *Given a linear parametric formula $\phi(\bar{x}, \bar{a})$ 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 ι such that $dom(\iota) = |\bar{a}|$, and $(\phi\ \iota)$ is $\mathcal{T}$-unsatisfiable is decidable.*

Proof Sketch. We express the problem as trying to decide the validity of a formula of the form: $\exists \bar{a}. \forall \bar{x'}. \big(\forall \bar{f}. \phi'(\bar{x'}, \bar{f}, \bar{a})\big) \wedge \big(\forall \bar{s}. \phi_{set}(\bar{x'}, \bar{s})\big)$, where, $\bar{f}$ are the uninterpreted function symbols in ϕ, $\bar{s}$ are variables of set sort, $\bar{x'}$ are variables of other sorts, and ϕ_{set} is a formula in $\mathcal{T}_{set}$ 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

$$\frac{\mathcal{C}' = \sigma(x) \qquad (\mathcal{C}', \mathcal{H}, \sigma) \vdash e \Downarrow_p v, \Gamma'}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash \mathsf{in}(e, x) \Downarrow_0 v, \Gamma'} \;\textsc{In} \qquad \frac{\Gamma \vdash e \Downarrow_p v, \Gamma' : (\mathcal{C}', \mathcal{H}', \sigma')}{\Gamma \vdash e^\star \Downarrow_0 v, (\mathcal{C}, \mathcal{H}', \sigma')} \;\textsc{Star}$$

$$\frac{\mathcal{H}(\sigma(x)) = (\lambda x. f_i\ (x, y), \sigma_1)) \qquad (\mathcal{C}, \mathcal{H}, \sigma[y_i \mapsto \sigma_1(y)]) \vdash e_i \Downarrow_p v, (\mathcal{C}', \mathcal{H}', \sigma')}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash x\ \mathsf{fmatch}\{\lambda x_1. f_i\ (x_i, y_i) \Rightarrow e_i\}_{i=1}^n \Downarrow_0 v, (\mathcal{C}', \mathcal{H}', \sigma)} \;\textsc{Fmatch}$$

$$\frac{\Gamma \vdash pre \Downarrow_p true, \Gamma_1 \qquad \Gamma \vdash e \Downarrow_q v, \Gamma_2 : (\mathcal{C}_2, \mathcal{H}_2, \sigma_2) \qquad (\mathcal{C}_2, \mathcal{H}_2, \sigma_2[R \mapsto q, \mathsf{res} \mapsto v, \mathsf{inSt} \mapsto \mathcal{C}, \mathsf{outSt} \mapsto \mathcal{C}_2]) \vdash post \Downarrow_r true, \Gamma_3}{\Gamma : (\mathcal{C}, \mathcal{H}, \sigma) \vdash \{pre\}\ e\ \{post\} \Downarrow_q v, \Gamma_2} \;\textsc{Contract}$$

where $R \in \{\mathsf{steps}, \mathsf{alloc}\}$

Figure 12. Semantics of the specification constructs fmatch, in, inSt, outSt and $*$.

elimination [45], the above formula could be reduced to an equivalent formula of the form $\exists \bar{a}.\forall \bar{x'}, \bar{f}.\phi''(\bar{x'}, \bar{a})$, which can decided using the algorithm presented in [52], and depicted in Fig. 7. □

D. 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 in(e, x) that evaluates an expression in a cache-state given by x, and a construct 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 outSt to refer to the state of the cache at the end of the function in the postcondition. In the construct in(e, x) the variable x has a *cache*. The constructs inSt and outSt are the only expressions that have these type. Therefore, even though the construct in(e, x) allows evaluating an expression under a cache given by x. The cache can only be obtained either through inSt or outSt expressions. Essentially, 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 Val to also include a cache. That is, $Cache \subseteq Val$. 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: fmatch and $*$ explained below. The construct $e^\star$ 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 fmatch of the form: $x\ \mathsf{fmatch}\{\lambda x_i. f_i\ (x_i, y_i) \Rightarrow e_i\}_{i=1}^n$ that performs structural matching on closures, i.e, matching based on structural equality. For instance, this expression matches x to the first case if x evaluates to a closure of the form: $(\lambda x. f_1\ (x, y), [y \mapsto u])$. It binds the variable y in the match case to the value u, and evaluates e_1 using the new binding. Fig. 12 shows the semantics of these constructs. Below we show the translation of these constructs in the model program.

$$[\![x\ \mathsf{fmatch}\{\lambda x_i. f_i\ (x_i, y_i) \Rightarrow e_i\}_{i=1}^n]\!]_P\ st =$$
$$x\ \mathsf{match}\{C_{l_i}\ y_i \Rightarrow [\![e_i]\!]_P\ st\}_{i=1}^n,$$
$$\text{where } l_i \text{ is the label of } \lambda x_i. f_i\ (x_i, y_i)/_{\cong, P}$$
$$[\![e^\star]\!]_P\ st = ([\![e]\!]_P\ st).\mathit{1}$$

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