Verifying Resource Bounds of Programs with Lazy Evaluation and Memoization

Ravichandhran Madhavan  
EPFL  
ravi.kandhadai@epfl.ch

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

Viktor Kuncak  
EPFL  
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 $\exists \forall \leq \ ? \ \ast \ \text{size}(l) + \ ?$ in the contracts of functions, as well as express invariants necessary for establishing the bounds that may possibly 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 (VCs) of the form $\exists \forall \leq \ ? \ \ast \ \text{size}(l) + \ ?$ 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 dequeue data structures that rely on aliasing of references to first-class functions; lazy data structures based on numerical representations such as the enqueue data structure of Scala’s data-parallel library, cyclic streams such as hamming number sequence, as well as dynamic programming algorithms like Levenshtein distance, Viterbi algorithm, and packrat parsing. Our evaluations show that when averaged over all benchmarks the actual runtime resource consumption is at least 80% of the value inferred by our tool when estimating the number of evaluation steps, and is at least 88% for the number of heap-allocated objects.

1. Introduction

Static estimation of performance properties of program is an important problem that has attracted great deal of research, and has resulted in techniques ranging from estimation of resource usage in terms of concrete physical quantities [54] to static analysis tools that derive upper bounds on the abstract complexities of programs [1, 21, 24]. Recent advances [4, 16, 21, 24, 50, 56] 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 ticks), 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 in the process. 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. This prevents many of the techniques from being applicable to formally reason about resource properties of implementations that have theoretically complex proofs for resource bounds, such as balanced trees where the time depends on the height or weight invariants that ensure balance. This is in stark contrast to the situation in verification of correctness properties where large-scale software and hardware verification efforts are commonplace [23, 27, 28, 33]. Alternative approaches [13, 35] have started incorporating user specifications to target more precise bounds and programs by making use of user-supplied program specific invariants.

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. We use the term memoization to refer to caching of outputs of a function for each distinct input encountered during an execution. By lazy evaluation we mean the usual combination of call-by-name and memoization [39], which we can view as wrapping values into memoized closures with a parameter of unit type. 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. For instance, dynamic programming algorithms, which have numerous practical applications e.g. in parsers such as packrat parser, are based on memoization. Other examples include lazy data structures proposed by Okasaki [39] that use lazy evaluation to support persistent operations, such as enqueue and dequeue, in worst-case constant time. A related but more complicated data structure Conqueue has been used to implement data-parallel operations in Scala efficiently [41, 42]. The (anonymous) Appendix C shows the verified implementations of lazy Real-time queue and Knapsack dynamic programming algorithm in our system. These features are being increasingly adopted by languages with large user base making their use increasingly prevalent. For example, Java 8, C#, and Scala all provide dedicated libraries like LINQ and Scala Streams that use lazy evaluation. 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 functional correctness—to the extent that precise running time bounds remain open in some cases (e.g. lazy pairing heaps described in page 79 of [39]). However, reasoning about correctness properties remains purely functional, which makes these features more attractive in comparison to imperative programming models from the view point of software verification.

We therefore believe that it is useful and important to develop algorithms and tools to formally verify resource usage of implementations of such algorithms with respect to abstract metrics such as steps and alloc. 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, on the program lazy, bottom-up merge sort one step of evaluation at runtime corresponded to 2.35 nanoseconds (ns) on average with...
an absolute deviation of 0.01 ns. In the Real-time queue benchmark 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 [37] with added support for memoization and new specification expressions. In our approach, users can specify the desired resource bound as templates with numerical holes e.g. as steps ≤ ? * size(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 of the function. For instance, our system was able to infer that 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 Stream datatype shown in Fig. 1 with two constructors SCons and SNil shows a definition of a stream, which is similar to a list datatype with constructors: Cons, and Nil. The first argument x of SCons is a pair of an unbounded integer (BigInt) and a boolean. The second argument tfun is a function from Unit to Stream. The type SCons has a lazy field tail (declared using lazy val in Scala syntax) that lazily evaluates tfun, i.e. computes tfun() once, and caches the result for reuse. The program defines a stream: primes that lazily computes for all natural numbers starting from 1 its primality, by creating an SCons whose second argument is a lambda term (anonymous function) that calls nextElem(2). Note that accessing the tail field of primes for the first time evaluates this call, which returns a new stream s such that s.tail invokes nextElem on the next natural number.

The function isPrimeNum(n) tests the primality of n by checking if any number greater than 1 and smaller than n divides n using an inner function rec. The number of steps it takes is linear in n. The function primesUntil returns all prime numbers until the parameter n using a helper function takePrimes, which is passed the primes stream, a counter i set to 0, and the number of elements to access from the primes stream initially set to n − 2. The function takePrimes recursively calls itself on the tail of the input stream, incrementing the index i as long as i < n (line 41). It then constructs an output list of prime numbers as the recursion unwinds. Consider now the running time of this function. Firstly, if takePrimes is given an arbitrary stream, its time (and hence steps) cannot be bounded because accessing the field tail at line 41 could take arbitrary amount of time. Therefore, we need to specify the structure of the closure tfun in the stream passed as input to takePrimes in order to prove its running time. This is accomplished by the function isPrimeS(x, i), which returns true if x is a SCons whose tfun parameter is equivalent to \((i := \text{nextElem}(i))\). Though the comparison at line 30 will always return false under reference equality, in our system, we relax this to allow for structurally equivalence of higher-order functions in order to allow specifying such properties. (We shortly explain the rationale for this choice in more detail, and formalize this equality in definition 1.) Using isPrimeS we can specify that the stream passed as input to takePrimes is an SCons whose tfun parameter invokes nextElem(i+2), as shown in the Fig. 1, and bound the steps of the function to \(O(n(n−i))\). Consequently, we can establish that

 sealed abstract class Stream
 private case class SCons(x: BigInt, tfun: () ⇒ Stream)
 extends Stream{
 lazy val tail = tfun() 
 }

 private case class SNil extends Stream
 private val primes = SCons((1, true), () ⇒ nextElem(2))

 def nextElem(i: BigInt): Stream = {
 require(i ≥ 2)
 val x = (i, isPrimeNum(i))
 val y = i + 1
 SCons(x, () ⇒ nextElem(y))
 } ensuring(r ⇒ steps ≤ ? * i + ?)

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

 def isPrimeS(s: Stream, i: BigInt): Bool = {
 require(i ≥ 2)
 s match {
 case SNil() ⇒ false
 case SCons(x, tfun) ⇒ tfun == ((i ⇒ nextElem(i)))
 }

 def primesUntil(n: BigInt): List = {
 require(n ≥ 2)
 takePrimes(0, n−2, primes)
 } ensuring(r ⇒ steps ≤ ? * n^2 + ?)

 def takePrimes(i: BigInt, n: BigInt, s: Stream): List = {
 require(0 ≤ i & & i < n & & isPrimeS(x, i+2))
 s match {
 case c@SCons((x, b), _ ) if i < n ⇒
 val t = takePrimes(i+1, n, c.tail)
 if(b) Cons(x, t) else t
 case _ ⇒ Nil()
 } ensuring(r ⇒ steps ≤ ? * (n(n−i)) + ?)

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

      sealed abstract class Stream
      private case class SCons(x: BigInt, tfun: () ⇒ Stream)
      extends Stream{
          lazy val tail = tfun() 
      }
      private case class SNil extends Stream
      private val primes = SCons((1, true), () ⇒ nextElem(2))

      def nextElem(i: BigInt): Stream = {
          require(i ≥ 2)
          val x = (i, isPrimeNum(i))
          val y = i + 1
          SCons(x, () ⇒ nextElem(y))
      } ensuring(r ⇒ steps ≤ ? * i + ?)

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

      def isPrimeS(s: Stream, i: BigInt): Bool = {
          require(i ≥ 2)
          s match {
              case SNil() ⇒ false
              case SCons(x, tfun) ⇒ tfun == ((i ⇒ nextElem(i)))
          }

      def primesUntil(n: BigInt): List = {
          require(n ≥ 2)
          takePrimes(0, n−2, primes)
      } ensuring(r ⇒ steps ≤ ? * n^2 + ?)

      def takePrimes(i: BigInt, n: BigInt, s: Stream): List = {
          require(0 ≤ i & & i < n & & isPrimeS(x, i+2))
          s match {
              case c@SCons((x, b), _ ) if i < n ⇒
                  val t = takePrimes(i+1, n, c.tail)
              if(b) Cons(x, t) else t
              case _ ⇒ Nil()
          } ensuring(r ⇒ steps ≤ ? * (n(n−i)) + ?)

      Figure 2. Specifying properties dependent on memoization state.
      primesUntil takes \(O(n^2)\) steps. For primesUntil, our tool inferred that steps ≤ 16n^2 + 4.

      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

      Figure 2. Specifying properties dependent on memoization state.
      primesUntil takes \(O(n^2)\) steps. For primesUntil, our tool inferred that steps ≤ 16n^2 + 4.

      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. The time behavior of the function depends on the state of the memoization table when it is invoked making the reasoning about resources imperative. To specify such functions 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 if the first i calls to the tail field of the stream s have been cached. (Accessing the lazy field c.tail is similar to calling a function tail(c) that is memoized for its arguments c.) 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) as shown in line 9 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 as shown in line 10 using an disjunctive resource bound. 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. Note that unlike in an imperative language, here the state (the cache) is implicit, and cannot be directly accessed by the programmers to specify properties on it. However, the upside is that 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 (see section 4).

Equality of closures. Supporting equality of closures is important for two reasons. (a) Firstly, interesting data structures based on lazy evaluation use aliased references to closures. Expressing invariants of such data structures requires equating closures. Fig. 3 pictorially depicts the invariants of the conqueue data structure [42]. In the figure, c represents a closure whose captured arguments are a digit (1 or 0) and a reference to a stream. sched is a list of references to closures reachable from head. (b) Secondly, it is convenient in specifications to state that two closures have the same behavior as was required in the example shown in Fig. 1. While reference equality is too restrictive for convenient specification, full semantic equality between closures is undecidable and tricky, especially considering that functions can have contracts in our language. Therefore, we resort to structural equivalence of closures wherein two closures are equivalent iff their abstract syntax trees are identical without unfolding named functions. This is formally defined in Def 1 in section 2. This also has the advantage that it allows modeling reference equality of closures by associating unique identifiers with closures that are incremented as the closures are created in the program. In fact, our system supports reference equality for closures as well.

Approach and Contributions. Our approach operates in two phases. In the first phase we generate an instrumented first-order program with specifications, referred to as the model, that accurately captures the higher-order control flow using defunctionalization [4, 44], and the effects of memoization on resources using sets and algebraic datatypes. During this translation process we ensure that the resource usage of the input program is modeled accurately without any abstraction (described in more detail in section 3). In the subsequent phase, we convert the problem of verifying contracts of the generated first-order programs to checking assertions using an assume-guarantee reasoning that exploits predicates that evolve monotonically with changes to the cache. We then encode the assume/guarantee assertions as \( \exists \forall \) formulas in theories that can be efficiently decided by state-of-the-art SMT solvers. We explain the verification approach in section 4. To summarise 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. We formally define an operational semantics for the constructs of our language that is parametric with respect to the resource usage (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 [37], implementing complex lazy data structures and dynamic programming algorithms comprising 4.5K 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 at least 80% of the value inferred by the tool, and is at least 88% for alloc. (section 5).

2. Language and Semantics

Fig. 4 show the syntax of a simple higher-order function language extended with memoization, contracts and specification constructs, that we will use to formalize our approach. Expressions of the language consists of variables, constants (Cst), primitive operations (Prim) on integers and booleans, let expressions, match expressions, lambda terms, and applications x y. We distinguish between direct calls to named functions f x and indirect calls x y. A program is a set of functions definitions. We require that ev-


<table>
<thead>
<tr>
<th>CST</th>
<th>VAR</th>
<th>PRIM</th>
<th>EQUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>c ∈ Cst</td>
<td>x ∈ Vars</td>
<td>op ∈ Prim</td>
<td>v = s(x) ⇑ s(y)</td>
</tr>
<tr>
<td>Γ ⊢ c ⇑ c, Γ</td>
<td>Γ : (C, H, σ) ⊢ x ⇑ σ(x), Γ</td>
<td>Γ ⊢ op x ⇑ op(σ(x)), Γ</td>
<td>Γ : (C, H, σ) ⊢ x eq y ⇑ v, Γ</td>
</tr>
</tbody>
</table>

**LET**

Γ ⊢ e₁ ⊥ₚ v₁, (C', H', σ') = (e₂ ⊥ₚ v₂, (C'', H'', σ''))

Γ : (C, H, σ) ⊢ let x := e₁ in e₂ ⇑ v₂, (C', H', σ')

**MATCH**

H(σ(x)) = C₁ v

Γ : (C, H, σ) ⊢ x match {C₁, x ↘ e₁})ₚ₁ ⇑ v₁, (C', H', σ')

**LAMBDA**

a = fresh(H) eλ = (λx.f (x, y). [y → σ(y)])

Γ : (C, H, σ, F) ⊢ λx.f (x, y) ⇑ λx. (C, H[i → eλ], σ, F ∪ {f})

**MEMOCALL**

f ∈ MemΓ u = σ(x) = ¬(f(u) ∈ dom(C))

Γ : (C, H, σ) ⊢ f x ⇑ σ, (C', H', σ')

**MEMOCALL-HIT**

Γ : (C, H, σ) ⊢ f x ⇑ v, Γ'

**NONMEMOIZEDCALL**

f ∈ MemΓ f # MemΓ Γ : (f σ(x)) ⇑ v, Γ'

**CACHED**

v ⇑ ((f σ(x), v) ∈ H C

Γ : (C, H, σ) ⇑ cached(f x) ⊥ₚ₀ v, Γ

**CONCRETECALL**

(C, H, σ)[param₁(f) → u] body₁(f) ⇑ v, (C', H', σ')

Γ : (C, H, σ) ⊢ f u ⇑ v, (C', H', σ')

**MEMOCALLMISS**

f ∈ MemΓ u = σ(x) = ¬(f(u) ∈ dom(C))

Γ : (C, H, σ) ⊢ f x ⇑ σ, (C', H', σ')

**CONTRACT**

Γ : (C, H, σ) ⊢ pre ⇑ v, Γ₁

Γ : (C₂, H₂, σ₂) ⊢ (C₂, H₂, σ₂)[R → p, res → v] ⊑ post ⊥ₚ₀ v, Γ₂

**Figure 5.** Resource annotated operational semantics for the language defined in Fig. 4.

ery direct call invokes a function defined in the program, and indirect calls need not. Tdef shows the syntax of user-defined algebraic datatypes. The datatypes use structural equality formalized in definition 1. As a syntactic sugar, we consider tuples as a specification expressions is given by

For steps: \( c_{miss} = 2 \), \( c_{match} = i \), \( c_{op} = 1 \) for every other operation, and \( \oplus = + \).

For alloc: \( c_{cons} = c_{lambda} = c_{miss} = 1 \), \( c_{op} = 0 \) for every other operation, and \( \oplus = + \).

It naturally lends itself to a compositional reasoning on which our approach is based upon.

**Notation.** Given a domain \( A \), we use \( a \in A^* \) 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 \( A \to B \) to denote a partial function from \( A \) to \( B \). Given a partial function \( h, \tilde{h}(x) \) denotes the function that applies \( h \) point-wise on each element of \( x \), and \( h[a \mapsto b] \) denotes the function that maps \( a \) to \( b \) and every other value \( x \) in the domain of \( h \) to \( h(x) \). We use \( h[a \mapsto b] \) 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 \( (h_1 \uplus h_2)(x) = (x \in dom(h_2)) h_2(x) \) else \( h_1(x) \).

Let \( type(e) \) denotes the type of an expression \( e \). Given a lambda \( l \), we use \( FV(l) \) to denote free variable captured by \( l \), and \( target(l) \) to denote the function called in the body of the lambda. The operation \( e_{e} / f \) 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 \( \iota : T\text{Vars} \to \mathbb{Z} \), 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.

**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 \( C \), a heap \( H \), an assignment of variables to values \( \sigma \), and a set of function definitions. Formally,

\[
\begin{align*}
\{ \text{Val} \} & = \{ \text{Z} \cup \text{Bool} \cup Adr \} \\
\text{FVal} & = \text{Fids} \times \text{Val} \\
\text{DVal} & = \text{Cids} \times \text{Val}^* \\
\text{Clo} & = \text{Lam} \times \text{Store} \\
\text{Heap} & = \text{Adr} \to (\text{DVal} \cup \text{Clo}) \\
\sigma & = \text{Store} \to \text{Val} \\
C & = \text{Cache} \to \text{FVal} \to \text{Val} \\
\Gamma & = \text{Env} \subseteq \text{Cache} \times \text{Heap} \times \text{Store} \times \mathbb{Z}^{2|d|}
\end{align*}
\]
The cache component $C$ of the environments have the property that every key of the cache, which is a concrete function call in $F\text{Val}$, is mapped to the result of the call (Definition 2 formally defines this property). We define a few helper functions to operate on the semantic domains. Let $\text{fresh}(H)$ denote an element $a \in A \land a \notin \text{dom}(H)$. Let $\text{body}_H(f)$ and $\text{param}_H(f)$ denote the body and parameter of the functions defined in the environment $\Gamma$, and $\text{Mem}_V \subseteq \text{Fids}$ denote the set of memoized functions in the function definitions in $\Gamma$.

**Structural equivalence.** Below we formalize the notion of structural equivalence of datatypes and closures. Two datatypes are structurally equivalent iff they use the same constructor and their fields are equivalent. We define structural equivalence of closures similar to syntactic equality of lambdas modulo alpha renaming (but extended to captured variables). Two closures are structurally equivalent iff their lambdas are of the form $\lambda x.f(x, y)$ and $\lambda w.f(w, z)$, where the captured variables $y$ and $z$ are bound to structurally equivalent values. Two addresses are equivalent iff they are bounded to structurally equivalent values in the heap. Formally,

**Def 1.** Define a relation $\approx$ with respect to $H \in \text{Heap}$ as: (subscript omitted for clarity)

$\forall a \in \mathbb{Z} \cup \text{Bool}. a \approx a$

$\forall\{a, b\} \subseteq \text{Adr}. a \approx b \iff \text{H}(a) \approx \text{H}(b)$

$\forall f \in \text{Fids}.\{a, b\} \subseteq \text{Val}$. $f(a) \approx (f b) \iff a \approx b$

$\forall c \in \text{Cuds}.\{a, b\} \subseteq \text{Val}^\sigma.(c a) \approx (c b) \iff \forall i \in [1, n].a_i \approx b_i$

$\forall\{l_1 : \lambda x.f(x, y), l_2 : \lambda w.f(w, z)\} \subseteq \text{Lam}$.

$\forall\{\sigma_1, \sigma_2\} \subseteq \text{Store}.(l_1, \sigma_1) \iff (l_2, \sigma_2) \implies \sigma_1(y) \approx \sigma_2(z)$

Note that the above definition uses a simple structural recursion over semantic domains, and does not use the operational semantics.

**Judgements.** We use judgements of the form $\Gamma \vdash e \downarrow \upsilon, \Gamma'$ to denote that under an environment $\Gamma \in \text{Env}$, an expression $e$ evaluates to a value $\upsilon \in \text{Val}$, and results in a new environment $\Gamma' \in \text{Env}$, while consuming $p \in \mathbb{N}$ units of a resource. When necessary we expand $\Gamma$ as $\Gamma' : (\mathcal{C}, H, \sigma, F)$ to highlight the individual components of the environment. We also ignore any component of the judgement that is not relevant to the context when there is no ambiguity. In Fig. 5, in all rules other than the rule $\text{LAMBDA}$ we ignore the function definitions from the environment as they are not updated by the rules.

**Valid environments.** Using the semantics defined in Fig. 5 we now formally characterize the valid environments that may arise during an evaluation of an expression, with respect to a program $P$.

**Def 2.** A quadruple $\Gamma' : (\mathcal{C}, H, \sigma, F)$ is a valid environment belonging to $\text{Env}$ for a program $P$ iff all the following conditions hold: (a) Every address in the range of $\sigma$ and $H$ are bounded to a value in $\mathcal{H}$. (b) Every variable in $P$ is bound to a value that belongs to the type of the variable in $P$. (c) Each function definition in $F$ has a unique function identifier. (d) $F$ is transitively closed i.e., every function referred to in the body of $F$ is bound to a definition in $F$. (e) $F$ contains every function definition in $P$. (f) for every closure in $\mathcal{H}$ containing a lambda $\lambda x.f(x, y)$, $f$ is defined in $F$. (g) $\forall k \in \text{dom}(C), \Gamma \vdash k \downarrow \upsilon, (\mathcal{C}', H', \sigma', F') = \upsilon \approx \upsilon$.

We do not formalize the constraints in detail as they are well-understood invariants ensured by an interpreter for a strongly typed functional language. The constraints (a) and (b) ensures sanity of the stores and heaps. The constraints (c) to (f) ensures that every lambda that may be applied during an evaluation starting from $\Gamma$ could be evaluated, and (g) ensures the sanity of the cached values. In the rest of the paper, we only consider valid environments. That is, whenever we say $\Gamma \in \text{Env}$, it is implicit that $\Gamma$ is a valid environment with respect to the program under consideration.

When an expression is evaluated under a valid environment, there are only two reasons why an evaluation of an expression may be undefined as per the operational semantics: (a) the evaluation diverges, or (b) there is a contract violation 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 $C_{\text{op}}$ that returns the resource requirement of an operation $\text{op}$ such as $\text{cons}$ or $\text{app}$. $C_{\text{op}}$ may possibly have parameters. In particular, we use $c_{\text{match}}(i)$ to denote the cost of a match operation when the $i$'th case was taken, which should include the cost of failing all the previous cases. (b) A resource combinator $\oplus : \mathbb{Z}^* \rightarrow \mathbb{Z}$ that computes the resource usage of an expression by combining the resource 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, $C_{\text{op}}$ is 1 for almost every operation (except $c_{\text{miss}}$ and $c_{\text{match}}(i)$). We consider construction of datatypes and primitive operations, including those on big integers, as unitary steps. We define $c_{\text{match}}(i) = i$ as we need to include the cost of failing all the $i - 1$ match cases. In the case of alloc, $C_{\text{op}}$ is 1 for datatype and closure creations, and also for a cache miss since it allocates a cache entry. It is zero otherwise. For both resources $\oplus$ is $+$. Our implementation, however, supports other resources such as abstract stack space usage and number of recursions. These resources can also be defined using the parameters.

**Semantic rules.** For brevity, we skip the discussion of straightforward semantic rules (such as let and match) and focus on rules that are atypical. The rule $\text{FVAL}$ defines the semantics of a call whose arguments have been evaluated to concrete values (in $\text{Val}$). It models the call-by-value parameter passing mechanism: it binds the parameters to argument values, and evaluates the body (an expression with contracts) under the new binding. A call evaluates to a value only if the contracts of the callee are satisfied as given by the rule: $\text{CONTRACT}$ (discussed shortly).

**Memoized Call Semantics.** The semantics of calling a memoized function is defined by the rules: $\text{MEMOCALLMISS}$ and $\text{MEMOCALLHIT}$. 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 cache 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 $\exists_{\text{hit}}$ that uses structural equivalence to lookup the cache as follows: $\exists_{\text{hit}} = \forall \upsilon \in \mathcal{H}. \exists \upsilon' \in \text{Val}. \exists \upsilon'' \in \text{dom}(C) \land \upsilon' \approx \upsilon$.

We parameterize the cost of searching and updating the cache using the parameters $c_{\text{hit}}$ and $c_{\text{miss}}$. In particular, to calculate the steps resource we consider lookup and update as unitary steps, and hence define $c_{\text{miss}} = 2$ (as it involves a lookup and an update operation) and $c_{\text{hit}} = 1$. However, in general, $c_{\text{miss}}$ and $c_{\text{hit}}$ may depend on the values of the arguments.

**Specifications.** The construct $\text{cached}(f \text{ x})$ evaluates to true in an environment $\Gamma$ iff the call $f$ is cached for the value of $x$ in $\Gamma$. Observe that the resource consumption of this construct is zero. This is because the construct is syntactically excluded from being part of the implementation of functions (see Fig. 4) which renders its resource usage irrelevant. The rule $\text{CONTRACT}$ defines the semantics of an expression $e$ of the form $\{\text{pre}\} e \{\text{post}\}$. The expression evaluates to a value $\upsilon$ only if $\text{pre}$ holds in the input environment and $\text{post}$ holds in the environment resulting after evaluating $e$. Observe that the value, cache effects, and resource usage of $e$ 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. We show the semantics of the constructs in and inStalong with a couple of other specification constructs in Appendix A. Informally, 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.

**Contract verification problem.** Given a (possibly open) program \( P \). The contract verification problem is to decide for every function \( \text{def } f \ x = \{ \text{pre } e \ \text{post } p \} \) defined in the program \( P \) whether in every valid environment with respect to \( P \), that has a binding for the parameter \( x \), in which pre does not evaluate to false, \( \{ \text{pre } e \} \) evaluates to a value. Formally, \( \forall \Gamma : (C, H, \sigma, F) \in \text{Env s.t. } x \in \text{dom}(\sigma) \exists v. \ (\Gamma \vdash \text{pre } \downarrow \text{false}) \lor \Gamma \vdash \text{pre } \downarrow v. \) (We omit the quantification on \( v \), and the constraint that \( x \in \text{dom}(\sigma) \) 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 \( \iota \) 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 \( e \ i \) denotes substituting the holes in an expressions \( e \) with the assignment given by \( \iota \). The resource bound inference problem is to find an assignment \( \iota \) such that for every function \( \text{def } f \ x = \{ \text{pre } e \ \text{post } p \} \) where \( \text{post } p \) may contain holes, \( \forall \Gamma \in \text{Env}. \ (\Gamma \vdash \text{pre } \downarrow \text{false}) \lor \Gamma \vdash \text{pre } \downarrow e \ \text{post } \downarrow v. \) (We omit the quantification on \( v \), and the constraint that \( x \in \text{dom}(\sigma) \) when there is no ambiguity).

### 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 source 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 \([44]\). (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. 5, by establishing a bisimulation between the source program and the model (Theorem 2). In contrast to related works \([4]\), which use defunctionalization as a means to estimate the resource usage of source 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 does not use higher-order features, memoization, or special specification constructs. However, we introduce two features that were not a part of the source language: (a) set values and set primitives such as union and inclusion, and (b) \( \text{error } \) construct that halts the evaluation.

\[
\begin{align*}
\text{em} & \in \text{Eval} := x \mid \text{Set } \mid \text{prim’ } x \mid f x \mid C \bar{x} \\
\text{prim’ } & := \text{prim } \mid \{ \text{em} \} \cup \{ \text{em} \} \mid \text{error }
\end{align*}
\]

The values of the model language includes sets \((\text{Set } = 2^\text{Val} \subseteq \text{Val})\). The environments of the model does not have the cache component: \( \Gamma \in \text{Env} = \text{Heap } \times \text{Store } \times 2^{\text{PDef}} \).

**Expression Translation**

\[
\begin{align*}
[e]_p \text{ st } & = (x, st, e_{\text{cost}}) \\
[\text{op } f]_p x \text{ st } & = (\text{op } f, st, e_{\text{cost}}) \quad \text{if } \text{op } \in \text{Prim} \\
[C \bar{x}]_p \text{ st } & = (C \bar{x}, st, e_{\text{cost}}) \quad \text{if } C \in \text{Cids} \\
[\text{let } x := e]_p \text{ st } & = \\
& \text{let } u := [e]_p \text{ st } \in \\
& \text{let } w := [e_1]\downarrow u_2 \text{ st } \in (u_1, u_2, c_{\text{let } + u_3 + u_3}) \\
[x \text{ match } (C_i \ x_i \Rightarrow e_i)]_p \text{ st } & = \ x \text{ match } \\
& (C_i \ x_i \Rightarrow \text{let } u := [e_i]_p \text{ st } \in (u_1, u_2, c_{\text{match(i) } + u_3}) \mid 1 = 1) \\
\end{align*}
\]

**Call and Lambda Translation**

\[
\begin{align*}
[f x]_p \text{ st } & = \quad \text{if } f \text{ does not have } @\text{memoize} \text{ annotation} \\
& \text{let } w := f_m (x, st) \text{ in } (u_1, u_2, c_{\text{call } + u_3}) \\
[f x]_p \text{ st } & = \quad \text{if } f \text{ has } @\text{memoize} \text{ annotation} \\
& \text{let } w := f_m (x, st) \text{ in } \\
& \text{let } x_{\text{cost}} = \text{if } (C_f x) \in \text{st} \{ c_{\text{init } } \} \text{ else } (c_{\text{miss } + c_{\text{call } + u_3}}) \\
& \text{in } (u_1, u_2, \{ (C_f x), x_{\text{cost}} \}) \\
[i : \lambda \ x.f \ (x, y)]_p \text{ st } & = (C_i y, st, e_{\lambda y}), \text{ where, } i = \text{label}(l/ x) \\
[x y]_p \text{ st } & = \\
& \text{let } w := \text{App}(e_{\text{type}(e), \ (x, y), st}) \text{ in } (u_1, u_2, c_{\text{app } + u_3}) \\
\text{Specification Construct Translation**} \\
\text{cached}(f x)]_p \text{ st } & = \{(C_f x) \subseteq \text{st}, \text{ st }, 0\} \\
\text{in}\{e, x\}_p \text{ st } & = [e]_p \text{ x} \\
\text{Dispatch Functions} \\
\text{def } App_r (cl, x, st) = \{ \\
& \text{cl match } \{ C_i, y \Rightarrow [e_i]_p \text{ st } : \cdots ; C_i, y_n \Rightarrow [e_{\text{res}_2}]_p \text{ st } \text{ st } c_{\text{res}_2}, y_1 \} \\
\text{Contract Translation} \\
& ([\text{pre } e]_p \text{ post } 1) \text{ st } = \quad \text{if } R \in \{ \text{steps, alloc}\} \\
& \{(\text{pre } e_1)_p \text{ st } . 1\} \\
& [e_1]_p \text{ st } \\
& \text{let } y = \text{post}[\text{res}_1/\text{res}_2/\text{st}/\text{inSt}/\text{res}_1/\text{R}]_p \text{ res } 2 \text{ in } y.1 \} \\
\text{Function Definition Translation} \\
\text{def } f x = e_p \text{ st } = \text{def } f_m (x, st) = [e]_p \text{ st }
\end{align*}
\]

**Figure 6.** Resource and cache-state instrumentation of the expressions of the language shown in Fig. 4.

Figures 6 formally defines the translation function \( [\cdot]_p \), that translates expressions of a source program \( P \) to the model program \( P^t \). Fig. 7(b) illustrates the translation on an implementation of a lazy take operation show in Fig. 7(a), which is a part of the scheduling-based Dseqe data structure described in [39] in Page 112, Fig. 8.4. The take operation returns the first \( n \) elements of the input stream, and requires that the input stream is memoized at least until \( n \) in order to achieve a constant time bound. The function conc\textit{Until} is shown in Fig. 2.

**Closure encoding.** We represent closures using algebraic datatypes in a way that preserves the structural equivalence of closures. We say two lambdas \( l_1 : \lambda x.f_1 (x, y) \) and \( l_2 : \lambda x.f_2 (x, z) \) are compatible, denoted \( l_1 \cong l_2 \), iff they invoke the same targets i.e., \( f_1 = f_2 \). This relation is interesting because during any evaluation two closures could be structurally equivalent iff their lambdas are compatible i.e., \( l_1 \cong l_2 \) iff \( \exists H, \sigma_1, \sigma_2 \text{ s.t. } (l_1, \sigma_1) \approx_H (l_2, \sigma_2) \). In the generated model we ensure that the closures with lambdas that
sealed abstract class Stream
private case class SCons(x: BigInt, tfun:() ⇒ Stream) extends Stream
  lazy val tail = tfun()
private case class SNil() extends Stream
def takeLazy(n: BigInt, s: Stream): Stream = {
  require(concrUntil(s, n))
  s match {
    case c: SCons ⇒
      val t = c.tail
      val n1 = n - 1
      SCons(c.x, () ⇒ takeLazy(n1, t))
    case SNil() ⇒ s
  }
} ensuring(r ⇒ steps ≤ ?)
sealed abstract class tStream
case class TakeLazy(y: BigInt, s: Stream) extends tStream
sealed abstract class Stream
case class SCons(x, tfun: tStream) extends Stream
sealed abstract class SNil() extends Stream
sealed abstract class Dcache
case class Tail(c: Stream) extends Dcache
def tail(c: SCons, st: Set) = app(c.tfun, st)
def app(tfun: tStream, st: Set) = {
  tfun match {
    case TakeLazy(n1, s1) ⇒ takeLazy(n1, s1, st)
  }
}
def takeLazy(n:BigInt, s:Stream, st:Set): (Stream, Set) = {
  require(concrUntil(s,n, st))
  s match {
    case c : SCons ⇒
      val u = tail(c, st)
      val nst = u_2 ∪ { Tail(c) }
      val n1 = n - 1
      (SCons(x, TakeLazy(n1, u_1)), nst,
       (if (Tail(c) ∈ st) 1 else u_3 + 3) + 9)
    case _ ⇒ (Nil(), st, 6)
  }
} ensuring(r ⇒ r.s ≤ ?)
def concrUntil(s: Stream, i: BigInt, st: Set): Boolean = {
  s match {
    case c : SCons if i > 0 ⇒
      Tail(c) ∈ st & & concrUntil(tail(c, st), i-1, st)
    case _ ⇒ true
  }
}

Figure 7. Illustration of the translation shown in Fig. 6.

are compatible are represented using the same datatype. For each lambda \( l \) let \( l/\equiv \) denote the representative of the equivalence class with respect to \( \equiv \). For each function type \( \tau \) used in \( P \), we add a datatype \( \tau \) to the model defined as follows. Let \( \{ i_1 \mid i \in [1, n] \} \) be the labels of the representatives (with respect to \( \equiv \)) of the lambda terms in the program that are of type \( \tau \). The datatype \( \tau \) has \( n + 1 \) constructors denoted \( C_i, i \in [1, n] \) and \( C_\tau \). That is, type \( \tau := (C_1, \tau_1, \cdots, C_n, \tau_n, C_\tau \) \). The \( i \)th constructor \( C_i \) represents the closure of the \( i \)th lambda term. The parameter of the constructor represents the free variable of the lambda term. The \( (n + 1) \)th constructor \( C_\tau \) of \( \tau \) is a stub for a closure created outside the program under analysis, and serves to handle an error case (explained shortly). Every function type \( \tau ⇒ \tau \) is replaced by the datatype \( \tau_{\text{rec}} \). In Fig. 7(b), the datatype \( \tau \) Stream defined at line 17 represents the closures of lambda types \( \tau \) Stream. The constructor TakeLazy represents the closure of \( () ⇒ takeLazy(n1, t) \) created at line 13. We replace the occurrences of the lambda with the construction of TakeLazy as shown by line 39. For this program we do not create a stub \( C_\tau \) as the type Unit ⇒ Stream is encapsulated and cannot be assigned a closure created outside the program.

**Cache encoding.** We instrument the expressions of the source 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 \( Dcache := (C_1, \tau_1, \cdots, C_n, \tau_n) \), where \( \tau_i \)'s are functions in the program annotated with \( \text{@memoize} \), and \( \tau_i \) is the type of the parameter of \( \tau_i \). In the illustration shown in Fig. 7 the datatype \( Dcache \) with one constructor: Tail(c) corresponds to this datatype. Though tail is a lazy field of the Stream datatype, we treat it as a memoized function with a single argument that refers to the receiver.

**Translation of expressions.** Fig. 6 shows the translation of expressions and types in the program. The translation of an expression \( e \) is given by \( [e]_P \), which takes a state expression \( st \) representing the keys of the cache before the evaluation of \( e \), and returns the translated expression (say \( e_n \)), which is a triple, where the first part \( e_{m,1} \) corresponds to the value of \( e \), the second part \( e_{m,2} \) corresponds to the keys of the cache after evaluation of \( e \), and the last part \( e_{m,3} \) corresponds to the resource usage of \( e \). Similar to the operational semantics shown in Fig. 5, we parametrize the computation of resources of an expression \( e \) using the cost function \( c_{op} \) and the combinator \( ⊕ \). However, here \( ⊕ \) is applied over expressions of the model that track resource usages instead of integers. The instrumentation of resource usage of expressions closely mimics the computation of resources in the operation semantics. It proceeds bottom-up, first instrumenting the sub-expressions of an expression \( e \), and then using the resource usage components of the instrumented sub-expressions (\( e_{m,3} \)) to instrument \( e \). However, instrumentation of a call to a memoized function is handled differently, and is explained shortly.

Fig. 7(b) shows the translated code for the function takeLazy as outputted by our tool. The code shown in Fig. 7 is obtained after a few straightforward static simplifications. For instance, the constants such as \( 9 \) and \( 3 \) that appear in the resource expressions are the result of summing up the operation costs along the static path containing the resource expressions.

**Cache-state propagation.** The instrumentation of cache state proceeds top down 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. The state parameter is used in two places: by calls to memoized functions to model their resource usage, and by the cached construct to check whether the call given as argument is memoized. Consider the translation of a call to a memoized function: \( f x \) shown in Fig. 6. It uses the input state parameter \( st \) to check whether the call would be a cache hit by testing if \( st \) contains \( C_1 \) which represents the call \( f x \). The resource usage in the cache hit case is given by \( c_{call} \), whereas in the miss case it is a combination of \( c_{m}, c_{m,3} \), the cost of the call \( c_{call} \), and the resource usage of the callee: \( w_3 \). Finally, \( (C_1 \ x) \) is added to the output state to record that the call is memoized (regardless of whether or not it was memoized before). Notice that 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 body \( res_2 \), as in the operational semantics. Moreover, any changes to the state caused by the contracts are discarded at the end of the contracts. Any uses of \( res \) in the post-condition is replaced with \( res_1 \), uses of a resource \( R \) by \( res_3 \), and
uses of inSt that represents the input state by st. Fig. 7 shows the result of propagating the state through the body of takeLazy function as outputted by our tool. Our tool eliminates propagation through expressions, and function that are statically inferred as not affecting the state. For instance, concatList does not return a state as it was statically determined to not have any effect on the state. Note that after the call tail(e, st) at line 36, an instance of Tail(c) is added to the output state to record that the call is memoized, and that the computation of steps at line 40 depends on whether or not Tail(c) belongs to the input state st.

**Defunctionalization.** We translate all function applications to first-order calls by creating a dispatch function AppP, for each function type τ in the program. The function takes a closure cl, the argument of the function x, and a state parameter st. The dispatch function pattern matches on the closure cl. Each constructor pattern of the form Ci, where i, is the label of a lambda, e dispatches to the expression $[e']$ st where e' is the result of replacing e, the parameter of the lambda a with x and the free variables of the lambda with the fields of the closure. If the closure matches Ci, the model halts with an error, as this case corresponds to the scenario where a function not defined within the program being analyzed is applied to an argument. Such a function, being arbitrary, 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 encapsulation) that the targets of the closures are strictly within the program under analysis.

We replace every application of the form $x y$ where type(x) = τ by a call to AppP, as shown in Fig. 6. Notice that in the illustration shown in Fig. 7, the application of tfunc inside the function tail is translated to a call to the dispatch function app.

**Soundness and Completeness of the Model.** We now establish the soundness and completeness of the model for verification of contracts of the source program. We start by defining a relation $\sim$ that relates an environment $\Gamma_1 \in Env$ of the source program with an environment $\Gamma_2 \in Env_{mod}$ of the model, much like a bisimulation relation between transition systems. However, some what unique to our setting, $\Gamma_1$ is actually simulated by the pair $(\Gamma_2, S)$ where $S \in Set$ denotes the key of the cache as formalized below. Let $P$ be a program and $\{H_1, H_2\} \subseteq H_{eap}$. We define a relation $\sim$ on the semantic domain as follows: (subscriptions omitted below for clarity)

1. $\forall a \in Z \cup Bool. a \sim a$
2. $\forall c \in Cds, \{a, b\} \subseteq Val^n. c \sim c \land b \iff \forall i \in [1, n], a_i \sim b_i$
3. $\forall (\ell, \sigma) \in Closure, v \in Val. (\ell, \sigma) \sim C_{\ell} v \iff (\sigma(FV((1))) \sim v \land (\exists l' \in Lam(P), l' \sqsubseteq l \land x = label(l'/\sqsupseteq)))$
4. $\forall f \in Fids, \{a, b\} \subseteq Val. f a \sim C_f b \iff a \sim b$
5. $\forall c \in Cache, S \in Set. C \sim c \iff |dom(c)| = |S| \land \forall x \in dom(c). \exists y \in S. x \sim y$
6. $\forall \{a, b\} \subseteq Aadr. a \sim b \iff H_{a}(a) \sim H_{b}(a)$
7. $\forall \{\sigma_1, \sigma_2\} \subseteq Store. \sigma_1 \sim \sigma_2 \iff \forall x \in dom(\sigma_1). \sigma_1(x) \sim \sigma_2(x)$

The relation formally captures that a cache is simulated by a set of instances of Dcache (rules S 5) and a closure is simulated by an instance of the datatype representing closures if the lambda l of the closure is compatible to a lambda in the program P (rule 3). We define a relation $\sim$ between an environment $\Gamma_1 \in Env$ and a pair in $Env_{mod} \times Set$ as follows:

**Def 3 (Simulation Relation).** For all $\Gamma_1 : (C_1, H_1, \sigma_1) \in Env$, and $S \in Set. \Gamma_1 \sim (F_2, S)$ iff $C_{\sigma_1} \sim S$, and $\sigma_1 \sim \sigma_2$.

The following theorem states that for every environment $\Gamma_1 \in Env$ and expression e, the translation of e (viz. $[e]$ st) with respect to a state st that evaluates to a value S under an environment $\Gamma_1 \in Env_{mod}$, correctly models (a) the resource usage of e, (b) the set of keys of the cache at the end of e, and (c) preserves the $\sim$ relation between the output environments and the output cache state, when evaluated under $\Gamma_2$, and $\Gamma_1 \sim (\Gamma_2, S)$.

**Theorem 1 (Bisimulation.).** Let $e \in Env_{mod}, \Gamma_1 \in Env$, and $\Gamma_2 \in Env_{mod}$ such that $\Gamma_2 \vdash \Gamma_1 \sim (\Gamma_2, S)$.

(a) If $\Gamma_1 \vdash e p v_1, \Gamma_1$ then $\exists \Gamma'_2 \in Env_{mod}. \exists u \in DVal st.$

\[ \Gamma_2 \vdash ([e] st) \downarrow u, \Gamma_2, \text{ and} \]

\[ \Gamma'_1 \sim (\Gamma_2, u, 2) \quad \Gamma_2 \sim \bigcup \sigma_1 , u_1 \quad p = u_3 \]

(b) If $\Gamma_2 \vdash ([e] st) \downarrow u, \Gamma_2$ and $u \in DVal then \exists \Gamma'_1 \in Env$ such that $\Gamma_1 \vdash e p \Gamma_1, \text{ and}$

\[ \Gamma'_1 \sim (\Gamma_2, u, 2) \quad \Gamma_2 \sim \bigcup \sigma_1 , u_1 \quad p = u_3 \]

Appendix B shows proof sketches of the theorems. Using the above property we now establish that for every function f in the program P, verifying the contracts of its translation f_n, will imply that the contracts of f hold, and vice-versa. A tricky aspect here is that there exists environments $\Gamma \in Env$ that binds variables to lambdas that are 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 $\sim$) in the model of P. The following theorem holds despite this because in such cases it can be shown that neither the contracts of f or f_n hold for all environments, as there exists an environment each in Env and Env_{mod} that results in a contraction violation in f and enforces the error condition in f_n.

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

\[ \forall \Gamma_2 \in Env_{mod}. \exists \Gamma_1 \vdash p \not\downarrow \Gamma_2 \vdash e_m \not\downarrow \Gamma_1 \vdash e \not\downarrow \Gamma_1 \vdash v \]

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 i, for every function def f_m (x, st) = e_m, \forall \Gamma_2 \in Env_{mod}. \exists \Gamma_1 \vdash p \not\downarrow \Gamma_2 \vdash e_m \not\downarrow \Gamma_1 \vdash e \not\downarrow \Gamma_1 \vdash v.

4. **Model Verification and Inference**

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

**Modular reasoning with higher-order functions.** Approaches based on function-level modular reasoning verify the postcondition of each function in the program under the assumption that the precondition of the function and the contracts of the callees (including itself) hold. The precondition of each function is verified at their call sites, independently. (The recursive functions in the program have to be well-founded.) We now formalize this reasoning, and subsequently present an extension for handling higher-order functions more effectively. We use the model shown in Fig. 7(b) as the running example.

Let A denote the set of function definitions in the program, and $e_1$ and $e_2$ be two properties i.e, boolean-valued expressions. Let $e_1 \rightarrow e_2$ denote that whenever $e_1$ does not evaluate to false, $e_2$ evaluates to true i.e, $\forall \Gamma \in Env_{mod}. \Gamma \vdash e_1 \not\downarrow \Gamma \vdash e_2 \not\downarrow \Gamma \vdash true$. (The operation $\rightarrow$ can be considered as an implication with respect to the operational semantics of the model language.) We use $\models a e_1 \rightarrow e_2$ to denote that under the assumption that the contracts of all functions in A holds $e_1 \rightarrow e_2$ is guaranteed. The modular
reasoning we described above corresponds to the following two rules:

**Function-level modular reasoning:**

(I) For each `def f = {pre} e {post}`, `|=A pre → post[e/res]`

(II) For each call site `c` of the form `f x` in the program, `|=A path(c) → pre(c)` where the variable `res` refers to the result of the expression `e` in the postcondition of `f`, and `path(c)` denotes the static `path` (possibly with disjunctions) to `c` from the entry of the function containing `c`. For instance, the path condition of the call `tail(c, st)` at line 36 of the program shown in Fig. 7(b) is shown below:

```
concUntil(s,n, st) ∧ s = c ∧ c instanceofOf(SCons)
```

For convenience we use `pre(c)` where `c` is a call of the form `f x` to denote the precondition of `f` (say `pre`) translated to arguments of the call i.e., `pre/f[param(f, p)]`. When the resource bounds in the programs have holes (in `TVars`), the assume/guarantee assertions generated as above would also have holes. The goal is to find an assignment `c` for holes such that all the assume/guarantee assertions of all functions are valid.

While this modular reasoning is applicable to the first-order model described in section 3, it dramatically increases the specification/verification overhead when applied as such to the model. For instance, consider the call to `takeLazy` within app at line 30 in the example shown in Fig. 7(b). With a plain modular reasoning, the precondition of app is not strong enough to show that the precondition of `takeLazy` namely `concUntil(s1, n1, st)` holds for the call. In order to verify this example, we need to assert that the property `concUntil(s, n, st)` holds for the values stored in every instance of `TakeLazy` transitively reachable from the recursive datatype `Stream`, which is stated by following function.

```
def pre(cl, st): Boolean = {cl match {
case TakeLazy(n,s1) ⇒ concUntil(s1,n1, st) &
  (s1 match {case SCons(_, t) ⇒ pre(t, st); case _ ⇒ true })
case _ ⇒ true }
}
```

What complicates this further is that to ensure this precondition at the call to `app` at line 27, the precondition of the function tail and all its transitive callers (including `takeLazy`) should be modified similarly. This scenario happens very often when dealing with lazy data structures. For instance in the `Real-time queue` data structure shown in Appendix C. Our initial attempts to use a precondition such as the above 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 `concUntil` actually holds at the points where the closure `TakeLazy` is created (namely at line 39 in Fig. 7(b)), and is monotonic with respect to the changes in the cache that happen during the program.

**Cache Monotonic Properties.** To mitigate the specification burden, we verify and utilize properties that are cache monotonic. Informally, a property `p ∈ E_{spec}` is cache monotonic iff whenever it holds in an environment with cache `C`, it also holds in all environments where the cache has more entries than `C`. 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 [19]).

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. For example, the `concUntil` property. Intuitively, this phenomenon seems to result from anti-monotonicity of resource usage: 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 `E ⊆ Stream` be a partial order on `Γ ∈ Env` defined by: `Γ₁ ⊆ Γ₂` iff `dom(Γ₁) ⊆ dom(Γ₂)` where `Γ₁` and `Γ₂` are the cache components of the environments respectively. A property `p ∈ E_{spec}` is cache monotonic iff `∀{Γ₁, Γ₂} ⊆ Env. Γ₁ ⊆ Γ₂ ⇒ Γ₁ ⊢ p ↓ true ⇒ Γ₂ ⊢ p ↓ true`. To check if a property `p ∈ E_{spec}` is cache monotonic it suffices to check the following property on the translation of `p` with respect to `Γ₁` defined in Fig. 6: `|=A (st₁ ⊆ st₂ ∧ [pr] st₁) ⇒ [pr] st₂`.

**Creation-dispatch rule for encapsulated types.** Let `⇒` be the set of encapsulated function types in a program which cannot be assigned closures created outside the program. For instance, the type `() ⇒ Stream` in Fig. 7(a) is an encapsulated function type. Let `{lᵢ | i ∈ [1, n]}` be the lambdas in the program of type `τ`, and `{Cᵢ xᵢ | i ∈ [1, n]}` be the closure constructions in the model representing the lambdas. Let `DispCalls = {fᵢ yᵢ | i ∈ [1, n]}` denote the calls made within the dispatch functions, representing the indirect calls to the lambdas. For instance, for the model shown in Fig. 7(b) the sets would be defined as follows:

```
l₁ = () ⇒ takeLazy(n₁,t)
(Cᵢ xᵢ) = TakeLazy(n₁,rᵢ)1 constructed at line 39
(fᵢ yᵢ) = takeLazy(n₁,sᵢ1) st called at line 30
```

Let `Props = {Pᵢ | i ∈ [1, n]}` such that `FV(Pᵢ) ⊆ xᵢ ⊆ st` be a collection of properties that are defined on the captured arguments of the closures (namely `Cᵢ`, `s₁`) and on a set-valued variable `st` representing the keys of the cache. Given an expression `e`, let `stvar(e)` be the state expression that statically reaches `e`. Note that there is exactly one state expression reaching every point in the program by the definition of the translation shown in Fig. 6. For instance, the state expression reaching the line 30 of Fig. 7(b) is `st`, whereas the state expression reaching the line 39 is `st`. We now extend the function-level assume/guarantee rules to include the following rule: if each of the properties `Pᵢ` are cache monotonic, and hold at the point of creation of the closure `Cᵢ` for the state reaching the creation point, then it can be assumed to hold at the point of dispatch. This is formally expressed as shown below:

**Modular reasoning with creation-dispatch rule**

(I) For each `def f = {pre} e {post}`, `|=A pre → post[e/res]`

(II) For each call site `c` of `DispCalls` of the form `f x` `|=A path(c) → pre(c)`

(III) (Cache monotonicity) For each `Pᵢ ∈ Props` `|=A (st₁ ⊆ st₂ ∧ [Pᵢ] st₁/st) ⇒ Pᵢ(st₂/st)`

(IV) For each closure construction site `c` of the form `Cᵢ xᵢ` `|=A path(c) → Pᵢ[stvar(e)/st]`

(V) For each call site `c` of `DispCalls` of the form `fᵢ yᵢ` `|=A path(c) ∧ Pᵢ[yᵢ/xᵢ][stvar(e)/st] → pre(c)`
While the above is reasoning holds irrespectively of how the properties $P_i$ are chosen for each lambda $l_i$, we use a particular strategy in our implementation. For each $l_i = \lambda x.f z$, we choose the property $P_i$ to be the disjuncts of the precondition of the call $f z$ that only refer to the captured variables $FV(l_i)$. For the example shown in Fig. 7(b), our approach would verify that (a) $\text{concUntil}$ is a cache monotonic property, which is encoded by the following assertion:

$$\models_{\mathcal{A}} s_1 \subseteq s_2 \land \text{concUntil}(s, i, s_1) \rightarrow \text{concUntil}(s, i, s_2)$$

and (b) that the property $\text{concUntil}(u, n_1, n, s_1)$ holds at point of creation of the closure $\text{takeLazy}(n_1, u, 1)$ at line 39. This is encoded by the following assertion:

$$\models_{\mathcal{A}} \left( \text{concUntil}(s, n, s) \land s = c \land c \text{ instanceof } \text{SCons} \land u = \text{tail}(c, st) \land \text{nst} = u \cdot \{ \text{Tail}(c) \land n1=n-1 \} \rightarrow \text{concUntil}(u, n, n, s) \right)$$

The property $\text{concUntil}(s_1, n_1, n, s_3)$ is assumed to hold while checking the precondition of call to $\text{takeLazy}$ at line 30. With this extension we do not need any more preconditions than what is stated in the program to verify the program.

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 in previous research [35], and implemented in the publicly available system [9, 48]. Fig. 8 shows a block diagram of the inference algorithm which we briefly describe in the sequel.

Given an assume/guarantee assertion $\models_{\mathcal{A}} e_1 \rightarrow e_2$ 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 $\bar{e}$ for $\bar{a}$ such that $\phi(\bar{x}, \bar{a})$ is unsatisfiable. (The VC could be thought of as a formula in the form: $\exists \bar{a}. \forall \bar{x}. \neg \phi$, but where the uninterpreted function symbols are universally quantified.) The VC's 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. 4), the VC's would be linear parametric formulas [34] in which every nonlinear term is of the form $a \cdot x$ from some $a$ belonging to $\bar{a}$ and $x$ belonging to $\bar{x}$. Converting an assume/guarantee assertion to a formula is fairly straightforward. Every primitive operation is mapped to a corresponding theory operation. Match expressions are converted to disjunctions, and let expressions to equalities. The bodies (with contracts) of function calls in the expressions are unfolded upto a certain depth, and treated uninterpreted. Nonlinear operation over $\bar{x}$ are axiomatized in the VC. The resulting 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.

Preserving source-level properties. Functions in the source-program (not specifications) are referentially transparent (i.e, functional) with respect to the result of evaluation. However, in the model program this is not immediately obvious as the functions are instrumented with the state. In fact, the axiom $\forall x, s_1, s_2, f_m(x, s_1), s_1 = f_m(x, s_2), s_1$ holds in the model but is not explicitly instantiated. We instantiate this property automatically in a complete way by encoding it using uninterpreted functions in the generated VC. This helps achieve a functional reasoning for properties on the results of expressions. We also automatically instantiate that the state at the end of a function is a super set of the state at the beginning.

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 $\phi$ unsatisfiable is computed using an iterative but terminating algorithm that progresses in two phases: an existential solving phase (phase I), and a universal solving phase (phase II). Phase I discovers candidate assignments $\bar{e}$ 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 $\bar{e}$ makes $\phi$ unsatisfiable. That is, if $\phi(\bar{x}, \bar{a})$ is unsatisfiable. If not, it chooses a disjunct $d(\bar{x}, \bar{a})$ satisfiable under $\bar{e}$ that only has numerical variables by axiomatizing uninterpreted functions and algebraic datatypes in a complete way [35]. 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 [46], 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 [17] and CVC4 [5] 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 [35]. Below we extend this result to include sets. (Appendix B shows the proof sketch.)

Theorem 3. 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 $\bar{e}$ such that $\text{dom}(\bar{e}) = [\bar{a}]$ and $(\phi \bar{e})$ is $\mathcal{T}$-unsatisfiable is decidable.

5. Evaluation

We implemented the approach described in the previous sections, and used our system to verify resource bounds of the many data structures and 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
Evaluation of accuracy of the inferred bounds. We instrumented the benchmarks for tracking steps and alloc resources as defined by the operational semantics, and executed them on concrete inputs that were likely to expose the worst case behavior. We varied the sizes of the inputs in fixed intervals up to 10k for most benchmarks. However, for those benchmark with nonlinear behavior we used smaller inputs that scaled within a cutoff time of 5 min, as tabulated in the column \( T \) of Fig. 10. For scheduling based data structures (discussed shortly) we varied the input in powers of two until the reach of prior works [16, 50]. For each benchmark, the figure shows the ratio between the runtime resource usage and the static estimate for each function when averaged over all top-level functions.

The column \( \text{dynamic/static} \times 100 \) of Fig. 10 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 80% of what was inferred statically for steps, and is 88% for alloc. In all cases, the inferred resource usage 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 \( \text{prims} \) benchmark shown in Fig. 1 the function \( \text{PrimeNum}(n) \) 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.)

### Benchmark statistics.

Fig. 9 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 \( \text{rtq} \) (shown in Appendix C) has been mentioned as being outside the reach of prior works [16, 50]. For each benchmark, the figure shows the number of functions with resource bound and \( \text{rtq} \) bounds of scheduling-based lazy data structures viz. \( \text{rtq}, \text{deque}, \text{num} \), and \( \text{conq} \) due to their complexity.

### Table: Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>BC</th>
<th>T</th>
<th>S</th>
<th>AT</th>
<th>Resource bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lazy data-structures</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lazy Selection Sort (sel)</td>
<td>70</td>
<td>36kb</td>
<td>4</td>
<td>1</td>
<td>1m</td>
<td>steps ≤ 15k ( \cdot l.\text{size} + 8k + 13 ) alloc ≤ 2k ( \cdot l.\text{size} + 2k + 2 )</td>
</tr>
<tr>
<td>Prime Stream (prims)</td>
<td>95</td>
<td>51kb</td>
<td>7</td>
<td>2</td>
<td>1m</td>
<td>steps ≤ 16n(^2) + 4 alloc ≤ 6n − 11</td>
</tr>
<tr>
<td>Fibonacci Stream (fib) [8]</td>
<td>199</td>
<td>59kb</td>
<td>5</td>
<td>5</td>
<td>1m</td>
<td>steps ≤ 45n + 4 alloc ≤ 4n</td>
</tr>
<tr>
<td>Hamming Stream (hams) [8]</td>
<td>223</td>
<td>78kb</td>
<td>8</td>
<td>6</td>
<td>2m</td>
<td>steps ≤ 129n + 4 alloc ≤ 16n</td>
</tr>
<tr>
<td>Stream library (slib) [49]</td>
<td>408</td>
<td>0.1mb</td>
<td>22</td>
<td>5</td>
<td>1m</td>
<td>steps ≤ 25l.\text{size} + 6 alloc ≤ 4l.\text{size}</td>
</tr>
<tr>
<td>Real time queue (rtq) [38, 39]</td>
<td>207</td>
<td>69kb</td>
<td>5</td>
<td>6</td>
<td>1m</td>
<td>steps ≤ 37, steps ≤ 40 alloc ≤ 8, alloc ≤ 7</td>
</tr>
<tr>
<td>Lazy Mergesort (msort) [2]</td>
<td>290</td>
<td>0.1mb</td>
<td>6</td>
<td>8</td>
<td>1m</td>
<td>steps ≤ 36k ( \cdot \log (l.\text{size}) ) + 56l.\text{size} + 22 alloc ≤ 6(k ( \cdot \log (l.\text{size}) )) + 6l.\text{size} + 3</td>
</tr>
<tr>
<td>Deque (deq) [38, 39]</td>
<td>426</td>
<td>0.1mb</td>
<td>16</td>
<td>7</td>
<td>3m</td>
<td>steps ≤ 580, steps ≤ 970 alloc ≤ 50, alloc ≤ 78</td>
</tr>
<tr>
<td>Lazy Numerical Rep.(num)[39]</td>
<td>546</td>
<td>0.1mb</td>
<td>6</td>
<td>25</td>
<td>1m</td>
<td>steps ≤ 106 alloc ≤ 15</td>
</tr>
<tr>
<td>Conqueue (conq) [41, 42]</td>
<td>880</td>
<td>0.2mb</td>
<td>12</td>
<td>33</td>
<td>3m</td>
<td>steps ≤ 124 alloc ≤ 23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>steps ≤ 29(</td>
</tr>
<tr>
<td><strong>Dynamic Programming</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCS (lex)</td>
<td>121</td>
<td>37kb</td>
<td>4</td>
<td>4</td>
<td>1m</td>
<td>steps ≤ 33mn + 33m + 33n + 31 alloc ≤ 2mn + 2m + 2n + 3</td>
</tr>
<tr>
<td>Levenshtein Distance(levd)</td>
<td>110</td>
<td>37kb</td>
<td>4</td>
<td>4</td>
<td>1m</td>
<td>steps ≤ 39mn + 39m + 39n + 37 alloc ≤ 2mn + 2m + 2 + 3</td>
</tr>
<tr>
<td>Hamming Numbers (hm)</td>
<td>105</td>
<td>44kb</td>
<td>3</td>
<td>3</td>
<td>3m</td>
<td>steps ≤ 71n + 70 alloc ≤ 3n + 4</td>
</tr>
<tr>
<td>Weight Scheduling (ws)</td>
<td>133</td>
<td>44kb</td>
<td>3</td>
<td>5</td>
<td>1m</td>
<td>steps ≤ 20joi + 19 alloc ≤ 2joi + 3</td>
</tr>
<tr>
<td>Knapsack (ks)</td>
<td>122</td>
<td>48kb</td>
<td>5</td>
<td>4</td>
<td>1m</td>
<td>steps ≤ 17(w ( \cdot \text{i.size} ) + 18w + 17i.\text{size} + 18, alloc ≤ 2w + 3</td>
</tr>
<tr>
<td>Packrat Parsing (pp) [20]</td>
<td>249</td>
<td>73kb</td>
<td>7</td>
<td>5</td>
<td>1m</td>
<td>steps ≤ 73n + 70 alloc ≤ 11n + 11</td>
</tr>
<tr>
<td>Viterbi (vit) [52]</td>
<td>191</td>
<td>63kb</td>
<td>6</td>
<td>7</td>
<td>2m</td>
<td>steps ≤ 34k(^2) + 38k(^2) − 8k + 26k + 276l + 30 alloc ≤ 2k + 2k + 4l + 5</td>
</tr>
</tbody>
</table>

Figure 9. Selected benchmarks comprising of \( \sim \)4.5K lines of Scala code and 123 resource bounds each for steps and alloc.

i7 processor, having 32GB RAM and running Ubuntu operating system.
<table>
<thead>
<tr>
<th>Function</th>
<th>Accuracy (in wc)</th>
<th># of worst-case calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotateRev (deq)</td>
<td>92%</td>
<td>24 out of 31</td>
</tr>
<tr>
<td>takeLazy (deq)</td>
<td>94%</td>
<td>45 out of 148</td>
</tr>
<tr>
<td>rotateDrop (deq)</td>
<td>84%</td>
<td>2 out of 72</td>
</tr>
<tr>
<td>pAdd (pp)</td>
<td>100%</td>
<td>100 out of 10K</td>
</tr>
</tbody>
</table>

Figure 11. Behavior of inner functions of deq and pp benchmarks.

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 \( \mathcal{O}(k \cdot \log(l.size)) \) resource bound of `msort` benchmark. In any actual run, as \( k \) increases the size of the stream that is accessed (which is initially \( l \)) decreases. Hence, \( \lceil \log(l.size) \rceil \) 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. 10 shows this metric for each benchmark when averaged over all pareto optimal bounds of all top-level functions in the benchmark. A high percentage for this metric is an indication that any inaccuracy is due to imprecise templates, whereas a low percentage indicates a possible incompleteness in the resource inference algorithm, which is often due to non-linearity, or absence of sufficiently strong invariants. As shown in Fig. 10, the constants inferred by the tool were 91% accurate for steps, and 94% accurate for alloc, when compared to the pareto optimal values that fit 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 as Viterbi and primes). In the sequel, we discuss the benchmarks, and the results of their evaluation in more detail.

Cyclic streams. The benchmarks `fib` 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 [50]. In comparison to the related 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. The data structures differ significantly in their internal representation, invariants, resource usage, and the operations they support. These are one of the most efficient persistent data structures. For instance, the `rtq` benchmark takes a few nanoseconds to persistently enqueue an element into a queue of size \( 2^{30} \). The `conq` data structure [41] is used to implement data-parallel operations of the standard Scala library efficiently.

The data structures consist of one or two streams, referred to as `spine`, that store the content, and a `schedule`, which is a list of references to closures possibly nested deep within the spine. The content can be other data structures. In the case `conq` the content is a AVL-like balanced tree called `ConcTree` [41, 42]. The schedules correspond to unfinished operations like `enqueue` initiated previously. Every operation on the data structure is performed lazily in increments that complete in a constant number of steps. Whenever a new operation is initiated, the schedules are forced so that an increment of a previous operation is performed. A complex invariant ensures that the pending operations do not cascade to result in non-constant time worst-case behavior. A complete implementation of the data structure is shown in the Appendix C. 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 [39], which was unrayeved by the system.

As the results in Fig. 10 show, the inferred bounds were at least 83% accurate for `rtq` and `num` benchmarks, but have low accuracy for `deq` and `conq`. On further analysis of `deq` we found that the bounds inferred by our system for all inner functions (like `rotateDrop` and `rotateRev` functions of `deq`) were, in fact, reached in the dynamic runs but manifested only occasionally when invoked from the top-level functions. Fig. 11 shows the accuracy i.e., the worst-case runtime step count as a percentage of the inferred bound for the inner functions of `deq`, and the number of calls from top-level functions that resulted in the worst-case behavior. The low accuracy seems to result from the lack of sufficient invariants for the top-level functions that prohibits 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`, `takeWhile`, `cycle` etc. The operations were chosen from the Haskell stream library [49]. (We excluded functions such as `filter` and `drop` that can potentially diverge on infinite streams.) 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 [2], 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 [15] shown in Fig. 9 by expressing them as memoized recursive functions. The benchmarks `lcs` and `levd` implement the algorithms for finding the longest common subsequence and Levenshtein distance between two strings (represented as lists), respectively. The benchmark `ks` implements the algorithm for packing a list of items, each of value \( v_i \) and weight \( w_i \), into a knapsack of capacity \( w \) in a way that maximizes the total value of the items in the knapsack. (We show an implementation of the benchmark in Appendix C.) `hs` is a memoized version of the hamming stream benchmark that computes a sorted list of numbers of the form \( 2^3 \cdot 3^5 \cdot 5^k \). `ws` implements a weighted scheduling algorithm that optimally schedules \( n \) jobs with (overlapping) start and finish times so that the total value of the scheduled jobs is maximized. The benchmark `pp` is a memoized implementation of a packrat parser presented by [20], and uses the same parsing expression grammar as in that work. `vit` is an implementation of the Viterbi algorithm for finding the most likely sequence of hidden states in the hidden Markov models. As show in Fig. 10, 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 `aloc` for all benchmarks except `pp`. In the case of `vit` the main reason for inaccuracy stems from the cubic nature of the `cubic` template (shown in Fig. 9), as highlighted by the results of comparison with the pareto optimal bound shown in Fig. 10. In the case of `pp`, the evaluations were performed on random strings as we were unable to precisely deduce the worst-case input. Nevertheless, the bounds inferred were accurate for the inner functions: `pAdd`, `pMul`, and `pPrim` as shown by Fig. 11.

6. Related Work

Static Resource Analysis for Lazy Evaluation. Danielsson [16] present a lightweight type-based analysis for verifying time complexity of lazy functional programs, and applies it to implicit
defined templates for iteration pointers, written for embedded systems. Madhavan and Kunour approach. Carbonneaux et al. [13] present a system to verify fully automated, these approaches target simpler programs and simpler bounds, and has been evaluated for relatively large input sizes. The benchmarks listed also highlight the fact that this is demonstrated on complex, real-world programs. Resource-Aware ML [24, 26] is a type-driven approach for inferring resource bounds of ML programs. Other automated systems for resource bounds inference include Speed [21, 41, 56], and Costa [1]. These approaches do not seem to directly support lazy evaluation or memoization. While fully automated, these approaches target simpler programs and simpler bounds that depend on less complex invariants compared to our approach. Carbonneaux et al. [13] present an system to verify stack space bounds of C programs, without recursive calls and function pointers, written for embedded systems. Madhavan and Kunčak [35] present an approach that infers resource bounds from user-defined templates for first-order, non-lazy functional programs. Leino and Moskal [32] use coinduction to verify programs with (possibly) infinite lazy data structures. They do not consider resource properties of such programs. Blanchette et al. [10, 11] present a formal framework for soundly mixing recursion and corecursion in the context of interactive theorem provers. Most approaches for imperative programs target a homogeneous, mutable heap. Using mutation directly to model caches dramatically increases the contract overhead in our benchmarks, especially due the presence of higher-order functions. In this work, we consider an almost immutable heap except for the cache, and devise a verification approach to handle mutations to the cache efficiently. We believe that similar separation of heap into mutable and immutable parts can be extended to other forms of restricted mutation such as write-once fields [3] and Unique references to reduce the contract overhead. LiquidHaskell [51] supports verification of complex specifications expressed as type refinements. However, it does not allow specifying resource properties. The Leon system supports verification of programs with higher-order functions [53], but it does not support lazy evaluation or memoization in its present form.

References

A. Additional Specification Constructs

We support a few more specification constructs in our language for expressing properties about the implementation, which we discuss below. The semantics of these constructs are shown in Fig. 12. As mentioned in section 2, we support a construct $\mathit{inSt}$ to access the state of the cache at the beginning of a function in the postcondition of the function. Analogously, we support a construct $\mathit{outSt}$ to refer to the state of the cache at the end of the function in the postcondition. However, to define the semantics of these expressions, we slightly modify the domain of values $\mathit{Val}$ to also include a cache. That is, $\mathit{Cache} \subseteq \mathit{Val}$. Below we describe a couple of other specification constructs. The construct $e^*$ computes the result of an expression $e$ without caching the result of $e$ for reuse. This is a side-effect-free operation that is to be used in places where only the result of the expression is relevant. We support a construct $\mathit{fnmatch}$ of the form: $x \mathit{fnmatch} \{ \lambda x_i. f_i (x_i, y_i) \} \equiv e_i \}_{i=1}^m$ 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: $(\lambda x_i. f_i (x, y), [y \mapsto u])$. 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, and redefines the semantics of the contracts in the presence of these constructs.

B. Proofs Sketches of Theorems

Theorem 1. (Bisimulation) Let $e \in E_{en}$, $st \in E_{mod}$, $\Gamma_1 \in Env$, and $\Gamma_2 \in Env_{mod}$ such that $\Gamma_1 \vdash st \Downarrow S$, $\Gamma_2 \sim (\Gamma_2', S)$.

(a) If $\Gamma_1 \vdash e \Downarrow_p v$, then $\Gamma_1' \in Env_{mod}$, $\exists u \in DVal s.t.
\Gamma_2 \vdash (\langle e \rangle st) \Downarrow u$, $\Gamma_2'$, and

- $\Gamma_1' \sim (\Gamma_2', u)$
- $v \sim_{H_{1}', H_{2}'} u$.
- $p = u_\lambda_3$.

(b) If $\Gamma_2 \vdash (\langle e \rangle st) \Downarrow u$, $\Gamma_2'$ and $u \in DVal$ then $\exists \Gamma_1' \in Env$

such that $\Gamma_1' \vdash e \Downarrow_p v$, $\Gamma_1'$,

- $\Gamma_1' \sim (\Gamma_2', u)$
- $v \sim_{H_{1}', H_{2}'} u_1$.
- $p = u_\lambda_3$.

Proof Sketch. To prove this we use structural induction over the expressions of the language. We prove that the claim holds for every expression, assuming that it holds for the subexpressions. We sketch the proof for one semantic rule. Others can be proved similarly. The proof depends on following two important properties: (a) the heaps in the environments evolve monotonically with respect to the evaluation of expressions. That is, the evaluation can only add more entries to the heap, and existing entries cannot be updated. This follows from the operational semantics, and more intuitively because the language is functional. (Recall that the caches are modeled separately.) Given two heaps $H_1$ and $H_2$, we say $H_1 \sqsubseteq H_2$ iff $H_2$ has more entries than $H_1$. (b) The relation $\sim_{H_1, H_2}$ is mono-
tonic with respect to $\sqsubseteq$. That is, if $x \sim_{H_1, H_2} y$, and $H_1 \sqsubseteq H_1'$, and $H_2 \sqsubseteq H_2'$ then, $x \sim_{H_1', H_2'} y$.


Now consider the part(a) of the theorem. We refer to the individual components of the program using the corresponding subscripts and primes. Consider the rule `MEMOCALLMISS` shown in Fig. 5. In this case, \( e = f \). Let \( r = [\llbracket e \rrbracket] \), \( p \). Consider the property \( \Gamma_1' \sim (\Gamma_2', r, 2) \) which is one of the three properties that need to be established. By the definition of the semantics, the store components of \( \Gamma_1' \) and \( \Gamma_2' \) are same as that of \( \Gamma_1 \) and \( \Gamma_2 \), respectively. This implies that \( \sigma_1' \sim_{H_1', H_2'} \sigma_2' \), as both \( H_2' \subseteq H_1' \) and \( H_2 \subseteq H_1 \), and \( \sim \) is monotonic with respect to the heaps. Notice that the bisimulation relation between two environments \( \sim \) defined in Def. 3 does not constrain the heap components of the environments. Therefore let us focus on the cache component.

In the case of \( \Gamma_1' \), the cache component is obtained by the evaluation of \( f \sigma_1(x) \) on \( \Gamma_1 \), and then adding a mapping \( (f w) \mapsto v \) (see Fig. 5). In the case of \( \Gamma_2' \), \( r, 2 \) is defined as the state obtained by the evaluation of \( f_m(\sigma_2(x), st) \) on \( \Gamma_2 \), and the union of \( C_f \) \( \sigma_2(x) \) (see Fig. 6). However, by inductive hypothesis, we can assume that the theorem holds for the body of the function \( f \) and \( f_m \) (given that parameter translation preserves the claim of the theorem). Therefore, we can assume that \( C_f \sim r, 2 \) with respect to the heaps at the exit point of the functions \( f \) and \( f_m \), respectively. Now, we need to show that the newly added key \((f \sigma_1(x)) \sim C_f \sigma_2(x) \). But, this follows from the definition of \( \sim \), the fact that \( \sigma_1(x) \sim_{H_1, H_2} \sigma_2(x) \), and the monotonicity of the relation \( \sim \) with respect to the heaps. Therefore, \( \Gamma_1' \sim (\Gamma_2', 2, r) \). Similarly, other parts of the claim can be proven.  

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

\[
\forall \Gamma_2 \in Env_{\text{mod}}.\exists v'. \ \Gamma_2 \vdash p_m \llbracket e \rrbracket false \vee \Gamma_2 \vdash e_m \llbracket v \rrbracket false
\text{iff } \forall \Gamma_1 \in Env.\exists v. \ \Gamma_1 \vdash p \llbracket e \rrbracket false \vee \Gamma_1 \vdash e \llbracket v \rrbracket false
\]

**Proof.** The proof of this theorem is more involved as not every \( \Gamma_1 \in Env \) can be simulated by a \( \Gamma_2 \) and a set pair. This is because the definition of \( \sim \) is with respect to lambdas and functions defined in a program \( P \), and \( \Gamma_2 \) may contain bindings that involves lambdas created outside the program. However, if this not the case, for example, in closed programs, or programs with complete encapsulation where closures cannot be assigned values from outside, the proof follows directly from the Theorem 2 as explained below.

**Proof for closed programs, and programs with encapsulated function types.** Let us consider the if direction. That is, say \( \forall \Gamma_1 \in Env. \Gamma_1 \vdash p \llbracket e \rrbracket true \vee \Gamma_1 \vdash e \llbracket v \rrbracket true \). Consider a \( \Gamma_2 \in Env_{\text{mod}} \). Let \( \sigma_2(st) = S \), where \( st \) is the state parameter of the function \( f_m \). Let \( \Gamma_1 \in Env \) such that \( \Gamma_1 \sim (\Gamma_2, S) \). Such a \( \Gamma_1 \) can be constructed by mapping all constructors in the `dom(\( \Gamma_2 \)` of the form `\( \lambda x.f \)` representing closure of a lambda `\( \text{lam} \)` with label \( I \) to their corresponding concrete closures: `\( \text{lam} \)[`\( FV(\text{lam}) \Rightarrow u] \)`), and all constructors of the form `\( (\text{lam} u) \in S \)` to a binding in the cache for the key (`\( f \ u) \in FVal \)` (This is in essence the concretization function.) We are given that \( \Gamma_1 \vdash p \llbracket e \rrbracket false \vee \Gamma_1 \vdash e \llbracket v \rrbracket false \). By Theorem 1, \( \Gamma_1 \vdash p \llbracket e \rrbracket false \vee \Gamma_1 \vdash e \llbracket v \rrbracket false \) implies that \( \Gamma_2 \vdash [p_m]_{\text{st}} \llbracket v \rrbracket false \), and \( v_1 \sim v_2 \) with respect to the heaps at the end of the evaluation. However, by the definition of translation, \( [p_m]_{\text{st}} = p_m \) (see Fig. 6), and since \( p \) is boolean valued, by the definition of `\( \sim \), \( v_1 \sim v_2 \). Similarly, it can be shown that, given \( \Gamma_1 \vdash e \llbracket v \rrbracket true \), \( \Gamma_2 \vdash [e]_{\text{st}} \llbracket v' \rrbracket holds. By definition, the body of \( f_m \) namely \( e_m \) is `\( [e]_{\text{st}} \)` `st`. Hence, the claim holds along the if direction. The other direction can be similarly proved.

**Proof of more general case.** To argue about the more general case, where closures can be assigned from outside the program \( P \), we need to reason about potentially diverging computations, and those that crash. For brevity, we only explain the proof informally for this case. Consider the only if direction. The other direction is similar. That is, say \( \forall \Gamma_2 \in Env_{\text{mod}}.\exists v'. \ \Gamma_2 \vdash p_m \llbracket e \rrbracket false \vee \Gamma_2 \vdash e_m \llbracket v' \rrbracket \). Now for every \( \Gamma_1 \in Env \) such that \( \Gamma_1 \sim (\Gamma_2, S) \), where \( S \in \Sigma_{\text{key}} \), it can be shown that the claim holds, as illustrated above, using Theorem 1. Now, consider a \( \Gamma_1 \in Env \) that is not related by `\( \sim \). The only way this could happen is either the keys of the cache \( C_f \) or the closures in \( H_1 \) do not belong to the program \( P \). Both cases are somewhat similar, since the syntactic restriction of the language requires that all named functions directly invoked are defined in program. So the only way a function `\( f \)` not defined in the program could be invoked is indirectly through a lambda application. Therefore, any cache entries `\( f \ u \)`, where `\( f \)` does not belong to the program would either not be used, or be looked up only from a lambda created outside the program. So let us only consider cases where closures could be created outside the program, and invoke functions not in the scope of the program. We refer to such closures external closures. Now, consider a \( \Gamma_1 \in Env \) such that evaluation of some expression `\( e \)` in `\( P \)` is not defined, either because no semantics rule applies, or because the expression `\( e \)` can diverge. However, we can still reason about the steps in the evaluation of `\( e \)`, though they may be unbounded. We now show that when the contracts of `\( f_m \)` hold for all inputs, evaluation of `\( e \)` on `\( \Gamma_1 \)` cannot be undefined, by deriving a contradiction in all the three following cases.

**Case (a).** An expression `\( e \)` belonging to `\( P \)` does not use any external closure when evaluated on `\( \Gamma_1 \)`. This is the simplest case. Clearly, we can remove the binding for external closures from `\( \Gamma_1 \)` and still execute `\( e \)` and produce the same behavior. Moreover, such a reduced `\( \Gamma_1 \)` would be related to a `\( \Gamma_2 \)`, and `\( S \in \Sigma_{\text{key}} \)` by `\( \sim \)`. Hence by Theorem 1, and the given facts, execution of `\( e \)` on `\( \Gamma_1 \)` must produce a value, which is a contradiction.

**Case (b).** An expression `\( e \)` belonging to `\( P \)` uses an external closure when evaluated on `\( \Gamma_1 \)`, but does not apply the lambda at any point.
This case would result if the closures are used only in equalities, but never applied to a value. Let \( x \) and \( y \) be two external closures that are ever compared during the evaluation of \( e \) on \( \Gamma_1 \), and have the same type \( \tau \) (otherwise type checking will fail). In the model, this environment \( \Gamma_1 \) can be simulated by an environment \( \Gamma_2 \) where \( x \) and \( y \) are mapped to instances of \( C \). If the \( x \) and \( y \) are equal in \( \Gamma_1 \) then we make the instances of \( C \) have the same parameters (note that \( C \) takes one integer parameter as defined in section 3), otherwise we assign different parameters to the constructors \( C' \). Now, execution of \( \Gamma_2 \) on \([e]_p\), \( st \) will simulate the execution of \( \Gamma_1 \). Hence from the given facts, execution of \( e \) on \( \Gamma_1 \) must produce a value, which is again a contradiction. Note that equating an external closure \( x \) (calling a function defined outside the program) with a closure created inside the program \( y \) is bound to fail both in \( P \) and \( P_m \) as we use structural equality for closures.

Case (c). An expression \( e \) belonging to \( P \) applies an external closure when evaluated on \( \Gamma_1 \).

Let \( y \) be the external closure that is applied first, before any other external closure. Clearly, the (dynamic) path \( p \) from the beginning of \( e \) to the point of application did not have any other external application. Therefore, as described in the cases (a) and (b), there exists a \( \Gamma_2 \) such that \( \sigma_2(\gamma) \) is an instance of \( C \), where \( \gamma \) is the type of the closure \( y \), and evaluating \( \Gamma_2 \) on \([p]_p\), \( st \) is defined. That is, it produces an output state, say \( \Gamma'_2 \). By the definition of the translation shown in Fig. 6, this application is modeled using \( App_f \), which on seeing \( \Gamma'_2 \) will halt the execution with an error, contradicting the assumption that the contracts hold for all \( \Gamma_2 \in Env_{mod} \) for the translated expression \([e]_p\), \( st \).

**Theorem 3.** Given a linear parametric formula \( \phi(\bar{x}, \bar{a}) \) with free variables \( \bar{x} \) and \( \bar{a} \), belonging to a theory \( T \) that is a combination of quantifier-free theories of uninterpreted functions, algebraic datatypes, and sets, and either integer linear arithmetic or real arithmetic, finding a assignment \( i \) such that \( \text{dom}(i) = [\bar{a}] \), and \( (\phi \ i) \) is \( T \)-unsatisfiable is decidable.

**Proof Sketch.** We express the problem as trying to decide the validity of a formula of the form: \( \exists \bar{x}. \forall \bar{z}. (\forall f. \phi'(\bar{z}, f, \bar{a})) \land (\forall s. \phi_{\text{set}}(\bar{x}', \bar{s})) \), where \( \bar{x} \) are the uninterpreted function symbols in \( \phi \), \( \bar{a} \) are variables of set sort, \( \bar{x}' \) are variables of other sorts, and \( \phi_{\text{set}} \) is a formula in \( T_{\text{set}} \) that has only set operations. This is possible because the existentially quantified variables \( \bar{a} \) are only numerical variables. Since the theory of set admits decidable quantifier elimination [29], 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 [35], and depicted in Fig. 8.

**C. Sample Implementations and Specifications of Benchmarks**

The following are two selected benchmarks verified by our tool. Okasaki’s Real-time queue falls in the category of lazy data structure and Knapsack falls in the category of dynamic programming problems.

**Okasaki’s Real-Time Queue.** Fig. 13 shows a complete implementation of the Okasaki’s Real-time queue data structure [38, 39] in our syntax. Consider the function rotate. It reverses the list \( r \) and appends it to the lazy stream \( f \), using the stream \( a \) as a temporary storage, which is initially set to empty. Essentially, rotate(\( f \), \( r \), \( a \)) = \( f \) ++ reverse(\( r \)) ++ \( a \). (\( f \) and \( r \) actually represent the front and rear parts of the real-time queue data structure.) However, the function performs its work lazily: every call to rotate constructs the first element of the result, and returns a stream whose tail is a suspended recursive call.

The specifications of rotate assert properties of the function that hold before and after the execution. Consider the property on the sizes of the arguments. This property is independent of the state i.e., it does not depend on whether the closures in the input list are forced or not. In contrast, the property isConcrete is state dependent: it returns true if every node of the argument stream has been forced, false otherwise. Notice that the postcondition also asserts a constant time bound for the function rotate. The requirement that isConcrete(f) holds at the beginning of the function is crucial for proving the time bound. Otherwise, forcing \( f \) at line 65 may invoke a previously suspended call to rotate, thus resulting in a cascade of forces.

As shown in Fig. 13, the real time queue data structure has three components: a lazy stream \( f \) denoting the front of the queue, a list \( r \) denoting the rear of the queue, and a lazy stream \( s \) denoting the schedule. We define the data structure invariants using the boolean-valued function valid. Every public queue operation, namely enqueue and dequeue, require that the valid property holds for the input queue, and also ensures that the property holds for the output queue (see the definitions of the functions in Fig. 13).

Consider the property firstUneval(f) == firstUneval(s) that relates the schedule and the front streams that is a part of the definition of valid. The definition of firstUneval is shown in Fig. 13. It returns the first node in the stream that has not been forced. This property states that the unevaled nodes of \( f \) and \( s \) are equal. In addition to this, the data structure also maintains the invariant that the size of the front is greater than the size rear, and that the size of the schedule is equal to the difference between the sizes of the front and the rear. These are succinctly captured by the second predicate of the function valid. The specification of the firstUneval function asserts a few interesting properties of the function that are needed for verification.

The data structure uses the same idea as a simple immutable queue that uses two lists, namely front and rear, that has a constant, amortized running time for ephemeral (i.e., non-persistent) usage. The elements are enqueued to the rear list and dequeued from the front list. Once in a while, when there are very few or no elements in the front list, the dequeue operation would reverse the rear and append it to the front. This is captured by the rotate function of Fig. 13. The real time queue data structure uses a similar strategy, but it exploits lazy evaluation to perform the costly rotate operation incrementally, alongside the enqueue and dequeue operations. It thus achieves constant running time, in the worst case, for all operations even under persistent usage. For this purpose, it augments the queue with a schedule which is a reference to a closure that corresponds to the next step of an unfinished rotate operation. The rotate operation itself is performed lazily: every call to rotate constructs the first element of the result, and returns a stream whose tail is a suspended recursive call.

During every enqueue and dequeue operation, if the schedule is non-empty, the head of the schedule is forced (line 127 of the function createQ). This corresponds to performing one step of the rotate operation. On the other hand, if the schedule is empty, which implies that there are no pending rotate operations, a new rotate operation is initiated (lines 129 to 130). Hence, whenever a rotate operation is initiated every node of the argument \( f \) is forced. This is asserted by the isConcrete(f) predicate used in the precondition of the rotate function, which is critical for proving the O(1) time bound of rotate. Our system verifies the complete program shown in Fig. 13.

**Memoized Knapsack Program.** Knapsack is a standard problem with a dynamic programming solution. Given a list of items with values and weight pairs. The goal is to maximize the values subject to total weight of the sack. Figure 16 shows the verified implementation of the Knapsack program. The entry point of this program
is the function knapsack(w, items) shown in Fig. 16. It computes the optimal values of filling knapsacks of capacity lesser than or equal to w using items. In each iteration, it invokes the function solveForWeight (through a helper function). The function requires that the optimal solutions of smaller weights have been computed and cached using the predicate deps. Given this property, it computes the optimal way of filling a knapsack of weight w by traversing the list of items, and choosing the items to include in the list. The time complexity of the algorithm is quadratic in items.size and w. Fig. 9 presents values inferred by the tool for the top-level function in this program.

```scala
object RealTimeQueue {
  sealed abstract class Stream[T] {
    @inline
def isEmpty: Boolean = this == SNil[T]()

def tail: Stream[T] = {
  @require(isEmpty)
  this match {
    case SCons(_, tailFun, _) => tailFun()
  }
}

def size = this match {
  case SCons(_, r) => r
  case SNil() => BigInt(0)
}

/**
 * A property that is true if 'sz' field decreases for the tail of the stream. 'sz' is a well-founded ordering.
 */
def valid: Boolean = {
  this match {
    case c @ SCons(_, tailFun, _)
      if (c.tail).valid
        && isConcrete(c.tail)
        && (c.tail).size + 1 & (c.tail).valid
        && sz == (c.tail).size + 1 & (c.tail).valid
        && sz > 0 &
        && isConcrete(c.tail)
        && sz => true
      }
  }

private case class SCons[T](x: T, tailFun: () => Stream[T], sz: BigInt) extends Stream[T]
private case class SNil[T] extends Stream[T]

/**
 * A property that holds for stream where all elements have been memoized.
 */
def isConcrete[T](l: Stream[T]): Boolean = {
  require(l.valid)
  l match {
    case c @ SCons(_, tailFun, _)
      if (c.tail).valid
        && isConcrete(c.tail)
        && sz == (c.tail).size + 1 & (c.tail).valid
        && sz => true
      }
  }

/**
 * A function that lazily performs an operation equivalent to 'f ++ reverse(r) ++ a'. Based on the implementation in Pg.88 of Functional Data Structures by Okasaki [39].
 */
@invisibleBody
@invstate
def rotate[T](f: Stream[T], r: List[T], a: Stream[T]): Stream[T] = {
  require(r.size == f.size + 1 & f.valid && a.valid && isConcrete(f))
  (f, r) match {
    case (SNil(), Cons(y, _)) =>
      SCons[T](y, lift(a), a.size + 1)
    case (c @ SCons(x, _), Cons(y, r1)) =>
      val newa = SCons[T](y, lift(a), a.size + 1)
      val ftail = c.tail
      val rot = () => rotate(ftail, r1, newa)
      SCons[T](x, rot, f.size + r.size + a.size)
  }
} ensuring (res => res.valid &&
  res.size == f.size + r.size + a.size &&
  !res.isEmpty &&
  steps <= ?)
```

17 2016/9/1
sealed abstract class List[T] {
  val size: BigInt = {
  this match {
  case Nil() => BigInt(0)
  case Cons(_, t) => 1 + t.size
  }
}
ensuring (_.size == 0)
}
}
case class Nil[T] extends List[T]
}
case class Cons[T](c: T, tail: List[T]) extends List[T]
}

/**
 * Returns the first element of the stream whose tail is not
 * memoized.
 */
def firstUneval[T](l: Stream[T]): Stream[T] = {
  require(l.valid)
  
  l match {
  case c @ SCons(_, _) =>
    if (cached(c.tail))
      firstUneval(c.tail)
    else l
  case _ => l
  }
}
ensuring (res =>>
  // (a) the returned stream is valid
  res.valid &
  // (b) no lazy closures implies stream is concrete
  (res.isConcrete || isConcrete(l)) &
  // (c) after evaluating the firstUneval closure in `l`
  // we can access the next unevaluated closure
  (res match {
    case c @ SCons(_, _) =>
      firstUneval(l) == firstUneval(c.tail)
    case _ => true
  }))/
}

case class Queue[T](f: Stream[T], r: List[T], s: Stream[T]) {
  @inline
  def isEmpty = f.isEmpty
  
  def valid = {
    f.valid &
    // invariant: firstUneval of `f` and `s` are the same.
    (firstUneval(f) == firstUneval(s)) &
    s.size == f.size - r.size // invariant: |s| = |f| - |r|
  }
}

/**
 * A helper function for enqueue and dequeue methods that
 * forces the schedule once
 */
@inline
def createQ(f: Stream[T], r: List[T], s: Stream[T]) = {
  s match {
  case c @ SCons(_, _) => Queue(f, r, c.tail) // force
  case _ =>
    val rotes = rotate(f, r, SNil[T]())
    Queue(rotes, Nil(), rotes)
  }
}

/**
 * Creates an empty queue, with an empty schedule
 */
def empty[T] = {
  val a: Stream[T] = SNil()
  Queue(a, Nil(), a)
}
ensuring (res =>> res.valid & steps <= ?)

/**
 * Reads the first elements of the queue without removing it.
 */
def head[T](q: Queue[T]): T = {
  require(q.isEmpty && q.valid)
  q.f match {
    case SCons(x, _) => x
  }
}
ensuring (res =>> steps <= ?)

/**
 * Appends an element to the end of the queue
 */
def enqueue[T](x: T, q: Queue[T]): Queue[T] = {
  require(q.valid)
  createQ(q.f, Cons(x, q.r), q.s)
}
ensuring { res =>>
  funeMonotone(q.f, q.s, inSt[T], outSt[T]) &
  res.valid & steps <= ?
}

/**
 * Removes the element at the beginning of the queue
 */
def dequeue[T](q: Queue[T]): Queue[T] = {
  require(q.isEmpty && q.valid)
  q.f match {
    case c @ SCons(x, r) =>
      createQ(c.tail, q.r, q.s)
  }
}
ensuring { res =>>
  funeMonotone(q.f, q.s, inSt[T], outSt[T]) &
  res.valid & steps <= ?
}

/**
 * Properties of 'firstUneval'.
 */
// We use 'fune' as a shorthand for 'firstUneval'

/**
 * st1.subsetOf(st2) == fune(l, st2) = fune(fune(l, st1), st2)
 */
@traceInduct
def funeCompose[T](l1: Stream[T], st1: Set[Fun[T]],
                   st2: Set[Fun[T]]): Boolean = {
  require(st1.subsetOf(st2) && l1.valid)
  (firstUneval(l1) in st2) ==
  (firstUneval(firstUneval(l1) in st1) in st2)
} holds

/**
 * Monotonicity of 'firstUneval' with respect to the state.
 */
@invisibleBody
def funeMonotone[T](l1: Stream[T], l2: Stream[T],
                    st1: Set[Fun[T]], st2: Set[Fun[T]]): Boolean = {
  require(l1.valid && l2.valid &&
  (firstUneval(l1) in st1) == (firstUneval(l2) in st1) &&
  st1.subsetOf(st2))
  // implies: fune(l1, st2) == fune(fune(l1, st1), st2)
  funeCompose(l1, st1, st2) &
  // implies: fune(l2, st2) == fune(fune(l2, st1), st2)
  funeCompose(l2, st1, st2) &
  (firstUneval(l1) in st2) ==
  (firstUneval(l2) in st2)
} holds

Figure 13. Okasaki’s Real-time queue data structure
object Knapsack {

sealed abstract class List { // a list of pairs: (weight, value)
    def size: BigInt = {
        this match {
            case Cons(_, tail) => 1 + tail.size
            case Nil() => BigInt(0)
        }
    }
}

// Monotonicity of deps with respect to the state
@traceInduct
def deps(i: BigInt, items: List): Boolean = {
    require(i > 0)
    cached(solveForWeight(i, items)) &&
    (if (i < 0) true
     else {
       deps(i - 1, items)
     })
}

// Monotonicity of deps with respect to the state
@traceInduct
def depsMono(i: BigInt, items: List, st1: Set[Fun[BigInt]]
    st2: Set[Fun[BigInt]]) = {
    require(i > 0)
    (st1.subsetOf(st2) &&
     (deps(i, items in st1))) =>> (deps(i, items in st2)
    ) holds
}

// A property that holds if the 'solveForWeight'
// is cached for all weights lesser than or equal to 'i'.
@traceInduct
require(
    def solveForWeight(i: BigInt, items: List): BigInt = {
        require(i > 0)
        (items.size + ?)
    }

@memoize
def solveForWeight(w: BigInt, items: List): BigInt = {
    require(w > 0) && (w == 0 ||
    solveForWeight(i, items)))
    if (w == 0) BigInt(0)
    else {
        maxOverItems(items, w, items)
    }
} ensuring (_ => steps <= ? * items.size + ?)

/**
 * Computes the optimal value of filling a knapsack of
 * 'weight' 'w' using 'remItems'
 */
@invstate
def maxOverItems(items: List, w: BigInt,
    remItems: List) = BigInt {
    require((w == 0 || (w > 0 &&
        solveForWeight(w - 1, items))) &&
    // lemma inst
    (remItems match {
        case Cons((wi, vi), _) =>>
            if (wi <= w && wi > 0) {
                val maxWithItem =
                vi + solveForWeight(w - wi, items)
                if (maxWithItem >= maxWithoutItem)
                    maxWithItem
                else
                    maxWithoutItem
            } else
                maxWithoutItem
    })
    case Nil() => true
})

remItems match {
    case Cons((wi, vi), tail) =>>
        val maxWithoutItem = maxOverItems(items, w, tail)
        if (wi <= w && wi > 0) {
            val maxWithItem =
            vi + solveForWeight(w - wi, items)
            if (maxWithItem >= maxWithoutItem)
                maxWithItem
            else
                maxWithoutItem
        } else
            maxWithoutItem
} ensuring (_ => steps <= ? * remItems.size + ?)

/**
 * A helper function that specifies properties ensured by
 * an invocation of 'solveForWeight'.
 */
@traceInduct
def solveForWeightHelper(i: BigInt, items: List) = {
    require(i == 0 ||
    solveForWeight(i, items))
    if (i == 0) {
        val maxMono(i - 1, items, inSt[BigInt], outSt[BigInt])
        &
    }
    steps <= ? * items.size + ?
} ensuring (_ => steps <= ? * items.size + ?)}

Figure 14. Knapsack program part I.

Figure 15. Knapsack program part II.
// Computes the optimal solution for all weights up to 'w'.
def solveUptoWeight(w: BigInt, items: IList): IList = {
  require(w >= 0)
  if (w == 0)
    Cons((w, solveForWeightHelper(w, items)), Nil())
  else {
    val tail = solveUptoWeight(w - 1, items)
    Cons((w, solveForWeightHelper(w, items)), tail)
  }
}

// Computes the list of optimal solutions
// for all weights up to 'w'
def knapsack(w: BigInt, items: IList) = {
  require(w >= 0)
  solveUptoWeight(w, items)
}

@ignore
def main(args: Array[String]) {
  import scala.util.Random
  // pick some random weights and values
  val input1 = (1 to 10).foldRight(Nil(): IList) { (i, acc) => Cons((i, i), acc) }
  val reslist1 = knapsack(100, input1)
  println("Optimal solutions: " + reslist1.toString)
  
  val l = ((1 to 10) zip (10 to 1 by -1))
  val input2 = l.foldRight(Nil(): IList) { (i, j, acc) => Cons((i, j), acc) }
  val reslist2 = knapsack(100, input2)
  println("Optimal solutions for set 2: " + reslist2.toString)
}

Figure 16. Knapsack program part III.