Contract-Based Resource Verification for Higher-Order Functions with Memoization

Ravichandhran Madhavan

EPFL, Switzerland
ravi.kandhadai@epfl.ch

Sumith Kulal

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

Viktor Kuncak*

EPFL, Switzerland
viktor.kuncak@epfl.ch

Abstract

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

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

General Terms Languages, Performance, Reliability, Verification

Keywords complexity, lazy evaluation, dynamic programming

1. Introduction

Static estimation of performance properties of programs is an important problem that has attracted a great deal of research, and has

resulted in techniques ranging from estimation of resource usage in terms of concrete physical quantities [80] to static analysis tools that derive upper bounds on the abstract complexities of programs [1, 32, 35, 46]. Recent advances [7, 22, 32, 35, 74, 84] have shown that automatically inferring bounds on more algorithmic metrics of resource usage, such as the number of steps in the evaluation of an expression (commonly referred to as steps) or the number of memory allocations (alloc), is feasible on programs that use higher-order functions and datatypes, especially in the context of functional programs. However, most existing approaches aim for complete automation but trade off expressive power and the ability to interact with users. Many of these techniques offer little provision for users to specify the bounds they are interested in, or to provide invariants needed to prove bounds of complex computation, such as operations on balanced trees where the time depends on the height or weight invariants that ensure balance. This is in stark contrast to the situation in correctness verification where large-scale verification efforts are commonplace [34, 40, 41, 49]. Alternative approaches [18, 52] have started incorporating user specifications to target more precise bounds and more complex programs.

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

Although our objective is not to compute bounds on physical time, our initial experiments do indicate a strong correlation between the number of steps performed at runtime and the actual wall-clock execution time for our benchmarks. In particular, for a lazy, bottom-up merge sort implementation [4] one step of evaluation at runtime corresponded to 2.35 nanoseconds (ns) on average

* This work is supported in part by the European Research Council (ERC) Project *Implicit Programming* and Swiss National Science Foundation Grant *Constraint Solving Infrastructure for Program Analysis*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01...\$15.00
<http://dx.doi.org/10.1145/3009837.3009874>

```

1 private case class SCons(x: (BigInt, Bool), tfun: () => SCons) {
2   lazy val tail = tfun()
3 }
4 private val primes = SCons((1, true), () => nextElem(2))
5
6 def nextElem(i: BigInt): SCons = {
7   require(i ≥ 2)
8   val x = (i, isPrimeNum(i))
9   val y = i + 1
10  SCons(x, () => nextElem(y))
11 } ensuring(r => steps ≤ ? * i + ?)
12
13 def isPrimeNum(n: BigInt): Bool = {
14   def rec(i: BigInt): Bool = {
15     require(i ≥ 1 && i < n)
16     if (i == 1) true else (n % i != 0) && rec(i - 1)
17   } ensuring (r => steps ≤ ? * i + ?)
18   rec(n - 1)
19 } ensuring(r => steps ≤ ? * n + ?)
20
21 def isPrimeStream(s: SCons, i: BigInt): Bool = {
22   require(i ≥ 2)
23   s.tfun ≈ (() => nextElem(i)) }
24
25 def takePrimes(i: BigInt, n: BigInt, s: SCons): List = {
26   require(0 ≤ i && i ≤ n && isPrimeStream(s, i+2))
27   if (i < n) {
28     val t = takePrimes(i+1, n, s.tail)
29     if (s.x._2) Cons(s.x._1, t) else t
30   } else Nil()
31 } ensuring(r => steps ≤ ? * (n(i)) + ?)
32
33 def primesUntil(n: BigInt): List = {
34   require(n ≥ 2)
35   takePrimes(0, n-2, primes)
36 } ensuring(r => steps ≤ ? * n2 + ?)

```

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

with an absolute deviation of 0.01 ns, and for a real-time queue data structure implementation [59] it corresponded to 12.25 ns with an absolute deviation of 0.03 ns. These results further add to the importance of proving bounds even if they are with respect to the abstract resource metrics.

In this paper, we propose a system for specifying and verifying abstract resource bounds, such as steps and alloc, of programs written in a pure subset of Scala [57] with added support for memoization and new specification constructs. In our approach, users can specify the desired resource bound as templates with numerical holes e.g. as $\text{steps} \leq ? * \text{size}(l) + ?$ 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 spent in accessing the k^{th} element of an unsorted list l using a lazy, bottom-up merge sort algorithm [4] is bounded by $36(k \cdot \lceil \log(l.\text{size}) \rceil) + 53l.\text{size} + 22$. We empirically compared the number of steps used by this program at runtime against the bound inferred by our tool by varying the size of the list l from 10 to 10K and k from 1 to 100. Our results showed that the inferred values were 90% accurate for this example (section 5 presents more results). We now present an overview of how programs can be specified and verified in our system using the pedagogical example shown in Fig. 1 that creates an infinite stream of prime numbers.

Prime stream example. The class `SCons` shown in Fig. 1 defines a stream that stores a pair of unbounded integer (`BigInt`) and boolean, and has a generator for the tail: `tfun` which is a function from `Unit` to `SCons`. The lazy field `tail` of `SCons` evaluates `tfun()` when accessed

```

1 def concrUntil(s: SCons, i: BigInt): Bool =
2   if (i > 0) cached(s.tail) && concrUntil(s.tail, i-1)
3   else true
4
5 def primesUntil(n: BigInt): List = {
6   // see Fig. 1 for the code of the body
7 } ensuring{r => concrUntil(primes, n-2) &&
8   (if(concrUntil(primes, n-2) in inSt)
9    steps ≤ ? * n + ?
10   else steps ≤ ? * n2 + ?) }

```

Figure 2. Specifying properties dependent on memoization state.

the first time and caches the result for reuse. The program defines a stream `primes` that lazily computes for all natural numbers starting from 1 its primality. The function `primesUntil` returns all prime numbers until the parameter n using a helper function `takePrimes`, which recursively calls itself on the tail of the input stream (line 28). Consider now the running time of this function. If `takePrimes` is given an arbitrary stream s , its running time cannot be bounded since accessing the field `tail` at line 28 could take any amount of time. Therefore, we need to know the resource usage of the closures accessed by `takePrimes`, namely $s.\text{tail}^*.\text{tfun}$. However, we expect that the stream s passed to `takePrimes` is a suffix of the `primes` stream, which means that `tfun` is a closure of `nextElem`. To allow expressing such properties we revisit the notion of *intensional or structural* equivalence, denoted $\approx$, between closures [5].

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

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

Properties depending on memoization table state. The quadratic bound of `primesUntil` is precise only when the function is called for the first time. If `primesUntil(n)` is called twice, the time taken by the second call would be linear in n , since every access to `tail` within `takePrimes` will take constant time as it has been cached during the previous call to `takePrimes`. The time behavior of the function depends on the state of the memoization table (or cache)

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

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

Contributions. The following are the contributions of this paper:

- We propose a specification approach for expressing resource bounds of programs and the necessary invariants in the presence of memoization and higher-order functions (section 2).
- We propose a system for verifying the contracts of programs expressed in our language by combining and extending existing techniques from resource bound inference and software verification (sections 3 and 4).
- We use our system to prove asymptotically precise resource bounds of 17 benchmarks, expressed in a functional subset of Scala [57], implementing complex lazy data structures and dynamic programming algorithms comprising 5K lines of Scala code and 123 resource templates (section 5).
- We experimentally evaluate the accuracy of the inferred bounds by rigorously comparing them with the runtime values for the resources on large inputs. Our results show that while the inferred values always upper bound the runtime values, the runtime values for steps is on average 80% of the value inferred by the tool, and is 88% for alloc (section 5).

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

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

2. Language and Semantics

Fig. 3 show the syntax of a simple, strongly-typed functional language extended with memoization, contracts and specification constructs, that we will use to formalize our approach. Every expression has a static label belonging to *Labels* (omitted in Fig. 3). We use e^l to denote an expression with its label. To reduce clutter, we omit the label if it is not relevant to the context. *Tdef* shows the syntax of user-defined algebraic datatypes and *Fdef* shows the syntax of function definitions. A program P is a set of functions definitions in which every function identifier is unique, every direct call invokes a function defined in the program, and the labels of expressions are unique. As a syntactic sugar, we consider tuples as a special datatype, and denote tuple construction using $(x_1, \dots, x_n)$, and selecting the i^{th} element of a tuple using x_i .

In particular, our language supports a structural equality operator `eq`, direct calls to named functions: `f x`, and indirect calls or lambda applications: `x y`. We also define an if-else operation if `cond e1 else e2` that is similar to a match construct with two cases. The annotation `@memoize` serves to mark functions that have to be memoized. Such functions are evaluated exactly once for each distinct input passed to them at run time. The language uses call-by-value evaluation strategy. Nonetheless, lazy suspensions can be implemented using lambdas with unit parameter and memoized functions. Expressions that are bodies of functions can have contracts (or specifications). Such expressions have the form $\{e_1\} \ e \ \{e_2\}$ where e_1 and e_2 are the pre-and post-condition of e respectively. The syntax of specification expressions is given by E_{spec} . The postcondition of an expression e can refer to the result of e using the variable `res`, and to the resource usage of e using `steps` and `alloc`. Users can specify upper bounds on resources as templates $e_t \in E_{tmp}$ with holes. The holes always appear as coefficients of variables defined in the program, which could be bound to more complex expressions through let binders.

For ease of formalization we enforce the following syntactic restrictions without reducing generality. All expressions except lambda terms are in *A-normal form* i.e., the arguments of all operations/functions are variables. All lambdas are of the form: $\lambda x. f \ (x, y)$ where f is a named function whose argument is a pair (a two element tuple) and y is a captured variable.

Notation and Terminology. Given a domain A , we use $\bar{a} \in A^*$ to denote a sequence of elements in A , and a_i to refer to the i^{th} element. (Note that this is different from tuple selector x_i , which

For steps: $c_{miss} = 2, c_{match(i)} = i + 1, c_{var} = c_{let} = 0$, for every other operation op : $c_{op} = 1. \oplus = +$ For alloc: $c_{cons} = c_{\lambda} = c_{miss} = 1$, for every other operation op : $c_{op} = 0. \oplus = +$			
CST $\frac{c \in Cst}{\Gamma \vdash c \Downarrow_p c, \Gamma}$ c_{cst}	VAR $\frac{x \in Vars}{\Gamma : (C, \mathcal{H}, \sigma) \vdash x \Downarrow \sigma(x), \Gamma}$ c_{var}	PRIM $\frac{pr \in Prim}{\Gamma \vdash pr \ x \Downarrow pr(\sigma(x)), \Gamma}$ c_{pr}	EQUAL $\frac{v = \sigma(x) \approx_{\mathcal{H}} \sigma(y)}{\Gamma : (C, \mathcal{H}, \sigma) \vdash x \text{ eq } y \Downarrow v, \Gamma}$ c_{eq}
LET $\frac{\Gamma \vdash e_1 \Downarrow_p v_1, (C', \mathcal{H}', \sigma') \quad (C', \mathcal{H}', \sigma[x \mapsto v_1]) \vdash e_2 \Downarrow_q v_2, (C'', \mathcal{H}'', \sigma'')}{\Gamma : (C, \mathcal{H}, \sigma) \vdash \text{let } x := e_1 \text{ in } e_2 \Downarrow v_2, (C'', \mathcal{H}'', \sigma')}$ $c_{let \oplus p \oplus q}$		CONS $\frac{a = \text{fresh}(\mathcal{H}) \quad \mathcal{H}' = \mathcal{H}[a \mapsto (\text{cons } \hat{\sigma}(\bar{x}))]}{\Gamma : (C, \mathcal{H}, \sigma) \vdash \text{cons } \bar{x} \Downarrow a, (C, \mathcal{H}', \sigma)}$ c_{cons}	
MATCH $\frac{\mathcal{H}(\sigma(x)) = C_i \bar{v} \quad (C, \mathcal{H}, \sigma[\bar{x}_i \mapsto \bar{v}]) \vdash e_i \Downarrow_q v, (C', \mathcal{H}', \sigma')}{\Gamma : (C, \mathcal{H}, \sigma) \vdash x \text{ match } \{C_i \bar{x}_i \Rightarrow e_i\}_{i=1}^n \Downarrow v, (C', \mathcal{H}', \sigma')}$ $c_{match(i) \oplus q}$		CONCRETECALL $\frac{(C, \mathcal{H}, \sigma[\text{param}_{\Gamma}(f) \mapsto u]) \vdash \text{body}_{\Gamma}(f) \Downarrow_p v, (C', \mathcal{H}', \sigma')}{\Gamma : (C, \mathcal{H}, \sigma) \vdash f \ u \Downarrow_p v, (C', \mathcal{H}', \sigma)}$	
LAMBDA $\frac{a = \text{fresh}(\mathcal{H}) \quad clo = (\lambda x. f(x, y), [y \mapsto \sigma(y)])}{\Gamma : (C, \mathcal{H}, \sigma) \vdash \lambda x. f(x, y) \Downarrow_{C_{\lambda}} a, (C, \mathcal{H}[a \mapsto clo], \sigma)}$		NONMEMOIZEDCALL $\frac{f \in Fids \quad f \notin Mem_{\Gamma} \quad \Gamma \vdash (f \ \sigma(x)) \Downarrow_p v, \Gamma'}{\Gamma \vdash f \ x \Downarrow_{c_{call \oplus p}} v, \Gamma'}$	
INDIRECTCALL $\frac{\mathcal{H}(\sigma(x)) = (\lambda z. e, \sigma') \quad (C, \mathcal{H}, (\sigma \uplus \sigma')[z \mapsto \sigma(y)]) \vdash e \Downarrow_p v, (C', \mathcal{H}', \sigma')}{\Gamma : (C, \mathcal{H}, \sigma) \vdash x \ y \Downarrow_{c_{app \oplus p}} v, (C', \mathcal{H}', \sigma)}$		MEMOCALLHIT $\frac{f \in Mem_{\Gamma} \quad ((f \ \sigma(x)), v) \in_{\mathcal{H}} C}{\Gamma : (C, \mathcal{H}, \sigma) \vdash f \ x \Downarrow_{c_{hit}} v, \Gamma}$	
MEMOCALLMISS $\frac{f \in Mem_{\Gamma} \quad u = \sigma(x) \quad \neg((f \ u) \in_{\mathcal{H}} \text{dom}(C)) \quad \Gamma \vdash (f \ u) \Downarrow_p v, (C', \mathcal{H}', \sigma')}{\Gamma : (C, \mathcal{H}, \sigma) \vdash f \ x \Downarrow_{c_{miss \oplus c_{call \oplus p}}} v, C'[(f \ u) \mapsto v], \mathcal{H}', \sigma}$		CACHED $\frac{v \Leftrightarrow ((f \ \sigma(x)) \in_{\mathcal{H}} \text{dom}(C))}{\Gamma : (C, \mathcal{H}, \sigma) \vdash \text{cached}(f \ x) \Downarrow_0 v, \Gamma}$	
CONTRACT $\frac{\Gamma \vdash pre \Downarrow_p true, \Gamma_1 \quad \Gamma \vdash e \Downarrow_q v, \Gamma_2 : (C_2, \mathcal{H}_2, \sigma_2) \quad (C_2, \mathcal{H}_2, \sigma_2[R \mapsto p, \text{res} \mapsto v]) \vdash post \Downarrow_r true, \Gamma_3}{\Gamma \vdash \{pre\} e \{post\} \Downarrow_q v, \Gamma_2} \quad R \in \{\text{steps}, \text{alloc}\}$			

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

is an expression of the language). We use $A \mapsto B$ to denote a partial function from A to B . Given a partial function h , $\hat{h}(\bar{x})$ denotes the function that applies h point-wise on each element of $\bar{x}$, and $h[a \mapsto b]$ denotes the function that maps a to b and every other value x in the domain of h to $h(x)$. We use $h[\bar{a} \mapsto \bar{b}]$ to denote $h[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. We omit h in the above notation if h is an empty function. We define a partial function $h_1 \uplus h_2$ as $(h_1 \uplus h_2)(x) = \text{if } (x \in \text{dom}(h_2)) \ h_2(x) \text{ else } h_1(x)$. Let labels_P denote the set of labels of all expressions in a program P . Let $\text{type}_P(e)$ denote the type of an expression e in a program P . Given a lambda e_{λ} , we use $FV(e_{\lambda})$ to denote the free variable captured by e_{λ} and $\text{target}(e_{\lambda})$ to denote the function called in the body of the lambda. The operation $e[e'/x]$ denotes the syntactic replacement of the free occurrences of x in e by e' . We use $[a, b]$ to denote a closed integer interval from a to b . Given a substitution $\iota : TVars \mapsto \mathbb{Z}$, we use $e \iota$ to represent substitution of the holes by the values given by the assignment. We also extend this notation to formulas later. We refer to programs and expressions without holes as *concrete* programs and expressions.

We now define the semantics of the language (Fig. 4) and subsequently define the problem of contract and resource verification. We use a big-step semantics (similar to Lauchbury's semantics for lazy evaluation [45]) as it naturally leads to a compositional reasoning, which is used by our approach. We also define a reachability relation on top of the big-step semantics to reason about environments that are reachable during an evaluation.

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

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

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

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

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

This equivalence satisfies congruence properties with respect to the result and resource usage of expressions (formalized in [50]).

Judgements. We use judgements of the form $\Gamma \vdash e \Downarrow_p v, \Gamma'$ to denote that under an environment $\Gamma \in Env$, an expression e evaluates to a value $v \in Val$ and results in a new environment $\Gamma' \in Env$,

while consuming $p \in \mathbb{Z}$ units of a resource. When necessary we expand Γ as $\Gamma : (\mathcal{C}, \mathcal{H}, \sigma, F)$ to highlight the individual components of the environment. We omit any component of the judgement that is not relevant to the discussion when there is no ambiguity. In Fig. 4, we omit the function definitions from the environment as they do not change during the evaluation.

Resource parametrization. We parametrize the operational semantics in a way that it can be instantiated on multiple resources using the following parametrization functions: (a) A cost function c_{op} that returns the resource requirement of an operation op such as *cons* or *app*. c_{op} may possibly have parameters. In particular, we use $c_{match(i)}$ to denote the cost of a match operation when the i^{th} case was taken, which should include the cost of failing all the previous cases. (b) A resource combinator $\oplus : \mathbb{Z}^* \rightarrow \mathbb{Z}$ 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_{let} and c_{var} are zero as the operations are normally optimized away or subsumed by a machine instruction. c_{op} is 1 for every other operation except c_{miss} and $c_{match(i)}$. We consider datatype construction and primitive operations on big integers as unitary steps. We define $c_{match(i)}$ proportional to i as we need to include the cost of failing all the $i - 1$ match cases. In the case of *alloc*, c_{op} is 1 for datatype and closure creations and also for a cache miss since it allocates a cache entry. It is zero otherwise. For both resources, the operation $\oplus$ is defined as addition ($+$). Our implementation, however, supports other resources such as abstract stack space usage and number of recursions.

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

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

state given by x , as illustrated by the example shown in Fig. 2. For brevity, we omit the formal semantics of the constructs *in* and *inSt*. The technical report [50] formalizes their semantics along with a match construct *fmatch* based on structural equality.

In the rest of the section, using the big-step semantics, we introduce a few concepts that are used in this paper, and formally define the problem of resource verification for open programs.

Reachability Relation. We define a relation $\rightsquigarrow$ (similar to the *calls* relation of Sereni, Jones and Bohr [37, 68]) that characterizes the environments that may reach an expression during an evaluation. Let $\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma', e' \rangle$ iff there exists a semantic rule shown in Fig. 4 of the following form for some (possibly empty) set of antecedents $A_1, \dots, A_n$ ($n \in \mathbb{N}$).

$$\frac{A_1 \dots A_i \quad \Gamma' \vdash e' \Downarrow_p v', \Gamma_1 \quad A_{i+1} \dots A_n}{\Gamma \vdash e \Downarrow_q v, \Gamma_2} i \in [1, n]$$

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

Valid environments. In reality, the environments under which an expression is evaluated satisfies several invariants which are ensured either by the runtime (like the invariant that the cached values of function calls correctly represent their results), or by the program under execution. Similar to prior works on data structure verification [39], we define the problem of contract/resource verification only with respect such valid environments under which an expression can be evaluated. Let $P_c = P' \| P$ denote a closed program obtained by composing a client P' with an open program P . The evaluation of a closed program P_c starts from a distinguished entry expression e_{entry} (such as a call to the main function) under an initial environment $\Gamma_{P_c} : (\emptyset, \emptyset, \emptyset, F)$ where F is the set of function definitions in the program P_c . We define the valid environments of an expression e belonging to an open program P , denoted $Env_{e,P}$, as $\{\Gamma \mid \exists P'. \langle \Gamma_{P'} \| P, e_{entry} \rangle \rightsquigarrow^* \langle \Gamma, e \rangle\}$.

When an expression belonging to a type correct program is evaluated under a valid environment, there are only two reasons why its evaluation may be undefined as per the operational semantics (provided the primitive operations are total): (a) the evaluation diverges, or (b) there is a contract violation during the evaluation.

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

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

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

3. Generating Model Programs

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

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

Illustrative Example. We use the constant-time take operation on a stream shown in Fig. 6 to illustrate the construction of the model, and later in section 4 to illustrate the verification of the model. Fig. 6(a) shows the take operation in the toy language used in the formalism, and Fig. 6(b) shows the model program explained in this section. In a real language, the function tail would be implemented as a lazy field of the *SCons* constructor as shown in Fig. 1. But for the purpose of verification, we treat it as a memoized function with a single argument as shown here. The function `concrUntil`, which is omitted, is similar to the Scala function shown in Fig. 2 that checks if the tail function is memoized for the first n suffixes of a stream. Observe that the lazy take operation (unlike `takePrimes`) returns a (finite) stream with the first element and a suspension of take, which when accessed constructs the next element. It requires that the input

Expression Translation

$$\begin{aligned} \llbracket x \rrbracket_P st &= (x, st, c_{var}) \\ \llbracket pr\ x \rrbracket_P st &= (pr\ x, st, c_{pr}) \quad \text{if } pr \in Prim \\ \llbracket x\ eq\ y \rrbracket_P st &= (x\ eq\ y, st, c_{eq}) \\ \llbracket C\ \bar{x} \rrbracket_P st &= (C\ \bar{x}, st, c_{cons}) \quad \text{if } C \in Cids \\ \llbracket let\ x := e_1\ in\ e_2 \rrbracket_P st &= \\ &\quad let\ u := \llbracket e_1 \rrbracket_P st\ in \\ &\quad let\ w := \llbracket e_2[u.1/x] \rrbracket_P u.2\ in\ (w.1, w.2, c_{let} \oplus u.3 \oplus w.3) \\ \llbracket x\ match\ \{ C_i\ \bar{x}_i \Rightarrow e_i \}_{i=1}^n \rrbracket_P st &= x\ match\ \{ \\ &\quad (C_i\ \bar{x}_i \Rightarrow let\ u := \llbracket e_i \rrbracket_P st\ in\ (u.1, u.2, c_{match(i)} \oplus u.3))_{i=1}^n \} \end{aligned}$$

Call and Lambda Translation

$$\begin{aligned} \llbracket f\ x \rrbracket_P st &= \quad \text{if } f \text{ does not have @memoize annotation} \\ &\quad let\ w := f^\#(x, st)\ in\ (w.1, w.2, c_{call} \oplus w.3) \\ \llbracket f\ x \rrbracket_P st &= \quad \text{if } f \text{ has @memoize annotation} \\ &\quad let\ w := f^\#(x, st)\ in \\ &\quad let\ x_{cost} = \text{if } (C_f\ x) \in st\ c_{hit} \text{ else } c_{miss} \oplus c_{call} \oplus w.3 \\ &\quad in\ (w.1, w.2 \cup \{(C_f\ x)\}, x_{cost}) \\ \llbracket e_\lambda \rrbracket_P st &= (C_l\ FV(e_\lambda), st, c_\lambda) \quad \text{if } e_\lambda / \simeq_{\neq, P} \text{ has label } l \\ \llbracket (x\ y)^l \rrbracket_P st &= \\ &\quad let\ w := App_l(x, y, st)\ in\ (w.1, w.2, c_{app} \oplus w.3) \end{aligned}$$

Specification Construct Translation

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

Contract Translation

$$\begin{aligned} \llbracket \{pre\} e \{post\} \rrbracket_P st &= \quad \text{if } R \in \{\text{steps}, \text{alloc}\} \\ &\quad \{(\llbracket pre \rrbracket_P st).1\} \\ &\quad \llbracket e \rrbracket_P st \\ &\quad \{let\ y = \llbracket post[res.1/res][st/inSt][res.3/R] \rrbracket_P res.2\ in\ y.1\} \end{aligned}$$

Function Definition Translation

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

Dispatch Functions

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

$\text{def } App_l(cl, w, st) :=$

$$cl\ match\ \{ C_{l_1}\ y_1 \Rightarrow \llbracket e'_1 \rrbracket_P st; \dots; C_{l_n}\ y_n \Rightarrow \llbracket e'_n \rrbracket_P st \\ C_\tau\ y \Rightarrow \text{error} \}$$

where, $\forall i \in [1, n]$. C_{l_i} are constructors of d_τ ,

$$(\lambda a_i. e_i)^{l_i} \text{ is a lambda in } P \text{ and } e'_i = e_i[y_i/z_i][w/a_i]$$

Figure 5. Resource and cache-state instrumentation.

stream is memoized at least until n in order to achieve a constant time bound. Otherwise, the call to tail at line 26 may result in a cascade of calls to take (via `app`). The challenge here is to verify that such cascade of calls cannot happen. The take operation with these contracts is in fact used by the Okasaki's persistent *Deque* data structure ([59] Page 111) that runs in worst-case constant time.

Closure encoding. We represent closures using algebraic datatypes in a way that preserves the structural equivalence of closures. We say two lambdas $e_\lambda = \lambda x. f(x, y)$, and $e_{\lambda'} = \lambda x. f'(x, z)$ are compatible, and denote it as $e_\lambda \cong e_{\lambda'}$, iff they invoke the same targets i.e., $f = f'$. This relation is interesting because during any evaluation two closures could be structurally equivalent iff their lambdas are compatible i.e., $e_\lambda \cong e_{\lambda'}$ iff $\exists \mathcal{H}, \sigma, \sigma'$ s.t. $(e_\lambda, \sigma) \approx_{\mathcal{H}} (e_{\lambda'}, \sigma')$. In the generated model we ensure that the closures of lambdas that are compatible are represented using the same datatype. For each lambda e_λ , we define a representative denoted $e_\lambda / \simeq_{\neq, P}$ of the equivalence class with re-

```

1  type Stream := (SCons (BigInt, Unit  $\Rightarrow$  Stream), SNil)
2  @memoize
3  def tail s = s match { SNil  $\Rightarrow$  SNil;
4    SCons (x, tfun)  $\Rightarrow$  (tfun Unit);
5  }
6  def take (n, s) =
7    { concrUntil(s, n) }
8    if (n  $\leq$  0) SNil else (s match {
9      SNil  $\Rightarrow$  SNil;
10     SCons (x, tfun)  $\Rightarrow$ 
11       let t := tail s in
12       let n1 := n - 1 in SCons(x,  $\lambda a$ .take (n1, t));
13   }){ steps  $\leq$  ? }
14   (a) A constant-time, lazy take operation

15 type tStream := (Take (BigInt, Stream), Other BigInt)
16 type Stream := (SCons (BigInt, tStream), SNil)
17 type Dcache := (Tail Stream)

18 def tail# (s, st) = s match { SNil  $\Rightarrow$  SNil;
19   SCons (x, tfun)  $\Rightarrow$  app (tfun, Unit, st); }
20 def app (cl, x, st) = cl match { Take (n1, s1)  $\Rightarrow$  take# (n1, s1, st); }
21 def take# (n, s, st) =
22   { concrUntil# (s, n, st) }
23   if (n  $\leq$  0) (SNil, st, 3) else (s match {
24     SNil  $\Rightarrow$  (SNil, st, 5);
25     SCons (x, tfun)  $\Rightarrow$ 
26       let u := tail# (s, st) in
27       let nst := u.2  $\cup$  { (Tail s) } in
28       let ucost := if ((Tail s)  $\in$  st) 1 else u.3 + 3 in
29       let ns := (SCons (x, Take (n - 1, u.1)) in
30         (ns, nst, ucost + 10);
31   }){ res.3  $\leq$  ? }

```

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

spect to $\cong$ that belongs to a program P . (It is undefined if P does not have a compatible lambda.) For each function type $\tau = A \Rightarrow B$ used in P , we add a datatype d_τ to the model (defined shortly), and replace every use of τ in the input program by the datatype d_τ .

Let $\{e_{\lambda_i} \mid i \in [1, n]\}$ be the representatives (with respect to $\cong$) of the lambda terms in the program P that are of type τ , and let $\{l_i \mid i \in [1, n]\}$ be their labels. The datatype d_τ has $n + 1$ constructors denoted C_{l_i} , $i \in [1, n]$ and C_τ . That is, d_τ is of the form: type $d_\tau := (C_{l_1} \tau_1, \dots, C_{l_n} \tau_n, C_\tau \text{Int})$. The i^{th} constructor C_{l_i} represents the closure of the i^{th} lambda term e_{λ_i} . The parameter of the constructor represents $FV(e_{\lambda_i})$. The type τ_i is obtained by recursively replacing the function types by their closure datatypes in $\text{type}_P(FV(e_{\lambda_i}))$. The $(n + 1)^{\text{th}}$ constructor C_τ of d_τ is a stub for a closure created outside the program under analysis and serves to handle an error case (explained shortly). In Fig. 6(b), the datatype $tStream$ defined at line 14 represents the closures of lambdas of type $\text{Unit} \Rightarrow \text{Stream}$. The constructor Take of $tStream$ represents the closure of λa .take (n1, t) created at line 12. As shown at line 29, the lambda is replaced by an instance of Take in the model. The constructor Other represents the stub closure c_τ .

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

Translation of expressions. Fig. 5 formally defines the transformation $\llbracket \cdot \rrbracket_P$ that maps expressions of a input program P to a model program $P^\#$. For every expression e , $\llbracket e \rrbracket_P$ takes a state expression st representing the keys of the cache before the evaluation of e and returns the translated expression denoted $e^\#$. $e^\#$ is a triple where the first element $e^\#.1$ corresponds to the value of e , the second element $e^\#.2$ corresponds to the keys of the cache after evaluation of e , and the last element $e^\#.3$ corresponds to the resource usage of e , which are explained in the sequel.

Cache-state propagation. The propagation of cache state proceeds top down in a store-passing style following the control flow of the program. To every function definition in the model, we add a fresh parameter st (of type $\text{Set}[Dcache]$) that represents the state of the cache at the beginning of the function (see translation of function definitions). This parameter is propagated through the bodies of the function recording all the calls that are memoized along the way. (E.g. see the translation of let expression.) The state parameter is used at two places: (a) by calls to memoized functions, and (b) by the cached construct to check whether the call given as argument is memoized. Consider the translation of a call to a memoized function shown in Fig. 5. It uses the input state parameter st to check whether the call would be a cache hit by testing if st contains $(C_f x)$ which represents the $FVal: (f x)$. The resource usage in the cache hit case is given by c_{hit} , whereas in the miss case it is a combination of c_{miss} , the cost of the call c_{call} and the resource usage of the callee $w.3$. Finally, $(C_f x)$ is added to the output state to record that the call is memoized. Observe that the call always happens in the model regardless of whether or not it was memoized before. This encodes the referential transparency of memoized functions i.e., the value of the call is always equal to the result of the invoked function, and avoids having to specify an invariant on the cache. (Recall that we are not interested in the resource usage of the model.) During the translation of contracts, the precondition is translated using the initial state st and the postcondition using the state resulting after the translation of the body res.2 , as in the operational semantics. Any changes to the state caused by the contracts are discarded at the end of the contracts. The uses of res in the postcondition is replaced by res.1 , the uses of a resource R by res.3 , and the uses of inSt , representing the input cache state, by st .

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

Resource Instrumentation. The instrumentation for resources closely mimics the computation of resources in the operational semantics. It proceeds bottom-up, first instrumenting the sub-expressions of an expression e , and then using the resource usages of the sub-expressions to instrument e , with the exception of a call to a memoized function, which uses the state (propagated top down) reaching the call expression to handle the cache hit case. The model shown in Fig. 6(b) is obtained after a few straightforward static simplifications performed by our tool. For instance, the constants such as 10 and 5 that appear in the resource expressions are the result of adding up all the constants in the instrumented expressions along the same branch (or match case) in the program.

Defunctionalization. We translate an indirect call: $x y$ to a guarded disjunction of direct calls through a process known as defunctionalization [64]. We replace every indirect call $x y$ with label l by a call to a dispatch function App_l constructed as follows. The parameters of the function are (a) a closure cl of type d_τ where $\tau = \text{type}_P(x)$, (b) the argument of the call w , and (c) a state parameter st denoting the state of the cache at the entry of the func-

tion. The dispatch function matches the closure cl to each possible constructor and in each case C_{l_i} , where l_i is the label of the lambda $\lambda a_i.e_i$ represented by the constructor, invokes the expression $\llbracket e'_i \rrbracket_P st$ where e'_i is the result of replacing in e_i the parameter of the lambda a_i with x and the free variable of the lambda with the field of C_{l_i} . If the closure matches C_τ , the model halts with an error as this case corresponds to the scenario where a function not defined within the program P is applied to an argument. Such a function, being arbitrary, may either not terminate or can have a precondition that is violated by the arguments it is applied to. The model soundly flags this case as an error. We eliminate this case if we can statically infer (based on type encapsulation) that the targets of the closures are strictly within the program under analysis. Observe that in Fig. 6 the call to tfun inside the function tail is translated to a call to the dispatch function app . (The case Other is omitted in app as we assume that the call is encapsulated.) Even though the set of possible cases in the function App_l could be large, many of those cases that are not feasible at runtime are not explored by our underlying verifier (section 4) which uses *targeted unfolding* [52] to unfold calls only along *satisfiable (abstract) paths*.

Soundness and Completeness of the Model. We now establish the soundness and completeness of the model for verification of contracts of an input program P . The proofs of all theorems that follow are presented in the technical report [50]. Let P be a program. Let $\{\mathcal{H}, \mathcal{H}^\sharp\} \subseteq \text{Heap}$. Define a relation $\sim_{\mathcal{H}, \mathcal{H}^\sharp, P}$ on the semantic domains as follows: (subscripts omitted below for clarity)

1. $\forall a \in \mathbb{Z} \cup \text{Bool}. a \sim a$
2. $\forall c \in \text{Cids}, \{\bar{a}, \bar{b}\} \subseteq \text{Val}^n. c \bar{a} \sim c \bar{b} \text{ iff } \forall i \in [1, n]. a_i \sim b_i$
3. $\forall (e_\lambda, \sigma) \in \text{Closure}, v \in \text{Val}, l \in \text{labels}_P. (e_\lambda, \sigma) \sim C_l v \text{ iff } \sigma(FV(e_\lambda)) \sim v \wedge (e_\lambda / \simeq_P \text{ is defined and has label } l)$
4. $\forall f \in \text{Fids}, \{a, b\} \subseteq \text{Val}. f a \sim C_f b \text{ iff } a \sim b$
5. $\forall C \in \text{Cache}, S \in \text{Set}. C \sim S \text{ iff } |\text{dom}(C)| = |S| \wedge \forall x \in \text{dom}(C). \exists y \in S. x \sim y$
6. $\forall \{a, b\} \subseteq \text{Adr}. a \sim b \text{ iff } \mathcal{H}(a) \sim \mathcal{H}^\sharp(a)$
7. $\forall \{\sigma, \sigma^\sharp\} \subseteq \text{Store}. \sigma \sim \sigma^\sharp \text{ iff } \text{dom}(\sigma) = \text{dom}(\sigma^\sharp) \wedge \forall x \in \text{dom}(\sigma). \sigma(x) \sim \sigma^\sharp(x)$

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

Theorem 1 (Bisimulation.). *Let P be a program and e an expression. Let st be an expression of the model language. Let $\Gamma \in \text{Env}$ and $\Gamma^\sharp \in \text{Env}^\sharp$ be such that $\Gamma^\sharp \vdash st \downarrow S$ and $\Gamma \sim_P (\Gamma^\sharp, S)$.*

- (a) *If $\Gamma \vdash e \downarrow_P v, \Gamma_o$ then $\exists \Gamma_o^\sharp \in \text{Env}^\sharp, u \in \text{DVal}$ such that $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P st) \downarrow u, \Gamma_o^\sharp$ and*
 - $\Gamma_o \sim_P (\Gamma_o^\sharp, u.2)$
 - $v \sim_{\mathcal{H}_o, \mathcal{H}_o^\sharp, P} u.1$
 - $p = u.3$
- (b) *If $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P st) \downarrow u, \Gamma_o^\sharp$ then $\exists \Gamma_o \in \text{Env}, v \in \text{Val}, p \in \mathbb{N}$ such that $\Gamma \vdash e \downarrow_P v, \Gamma_o$ and*
 - $\Gamma_o \sim_P (\Gamma_o^\sharp, u.2)$
 - $v \sim_{\mathcal{H}_o, \mathcal{H}_o^\sharp, P} u.1$
 - $p = u.3$

Using the above theorem, we now establish that for every function f in the program P , verifying the contracts of its translation

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

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

$$\begin{aligned} \forall \Gamma^\sharp \in \text{Env}^\sharp_{e', P^\sharp}. \exists u. \Gamma^\sharp \vdash p' \downarrow \text{false} \vee \Gamma^\sharp \vdash \tilde{e}' \downarrow u \text{ iff} \\ \forall \Gamma \in \text{Env}_{\tilde{e}, P}. \exists v. \Gamma \vdash p \downarrow \text{false} \vee \Gamma \vdash \tilde{e} \downarrow v \end{aligned}$$

The technical report [50] has the proofs of the above theorems. A corollary of the above theorem is that the model is sound and complete for the inference of resource bounds. That is, for any assignment to holes ι , $\forall \Gamma^\sharp \in \text{Env}^\sharp_{e', P^\sharp}. \Gamma^\sharp \vdash p' \downarrow \text{false} \vee \Gamma^\sharp \vdash (\tilde{e}' \iota) \downarrow u \text{ iff } \forall \Gamma \in \text{Env}_{\tilde{e}, P}. \Gamma \vdash p \downarrow \text{false} \vee \Gamma \vdash (\tilde{e} \iota) \downarrow v$.

4. Model Verification and Inference

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

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

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

Function-level modular reasoning:

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

Recall that the variable res refers to the result of e in the postcondition of e . For a call $c = f x$, we use $pre(c)$ to denote the precondition of f after parameter translation. The path condition $\text{path}(c)$ denotes the *static path* (possibly with disjunctions and function calls) to c from the entry of the function containing c . For instance, the path condition of the call $\text{tail}^\sharp(s, st)$ at line 26 of the program shown in Fig. 6(b) is: $\text{concrUntil}^\sharp(s, n, st) \wedge n > 0 \wedge s = \text{SCons}(x, \text{tfun})$. For programs with templates, the assume/guarantee assertions generated as above would have holes (TVars). The goal is then to find an assignment ι for holes such that all assume/guarantee assertions of all functions are valid. (For brevity,

we have omitted the formal definition of the assumptions and *path* as they are commonly known. The technical report [50] presents their formal definition.)

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

```
def pre (cl, st) = cl match {
  Take (n1,s1) => concrUntil# (s1,n1,st) ^
    (s1 match { SCons(x, t) => pre (t, st); SNil => true});
}
```

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

Cache Monotonic Properties. Informally, a property $p \in E_{spec}$ is cache monotonic iff whenever it holds in an environment with a cache C_1 , it also holds in all environments where the cache has more entries than C_1 . These properties are interesting because once established they can be assumed to hold at any subsequent point in the evaluation (similar to *heap-monotonic* type states introduced by Fähndrich and Leino [27]). We find that in almost all cases the properties that are needed to establish resource bounds are (or can be converted to) cache monotonic properties. Intuitively, this phenomenon seems to result from anti-monotonicity of resource usage i.e., the resource usage of an expression cannot increase when it is evaluated under a cache that has more entries. Below we formalize cache monotonicity and later describe how we exploit it in verification. Let $\Gamma_1 : (C_1, \mathcal{H}_1, \sigma_1, F)$ and $\Gamma_2 : (C_2, \mathcal{H}_2, \sigma_2, F)$. We say $\Gamma_1 \sqsubseteq \Gamma_2$ iff every component of Γ_2 has more entries than the corresponding component of Γ_1 i.e., $(k, v) \in C_1 \Rightarrow (k, v) \in C_2$, where C could be $\mathcal{H}$, $\mathcal{C}$ or σ . A property pr is cache monotonic iff $\forall \{\Gamma_1, \Gamma_2\} \subseteq Env. (\Gamma_1 \sqsubseteq \Gamma_2 \wedge \Gamma_1 \vdash pr \Downarrow true) \Rightarrow \Gamma_2 \vdash pr \Downarrow true$. To check if a property pr is cache monotonic it suffices to check the following property on the translation of pr with respect to $\llbracket \cdot \rrbracket_P$ defined in Fig. 5: $(st_1 \subseteq st_2 \wedge \llbracket pr \rrbracket_P st_1) \rightarrow \llbracket pr \rrbracket_P st_2$.

Creation-dispatch rule for encapsulated calls. Recall that each indirect call $x \ y$ has a set of target lambdas that are estimated at the time of model construction based on $type_P(x)$. Let $\Lambda = \{e_i \mid i \in [1, n]\}$, where $e_i = \lambda x. f_i(x, y_i)$, be the lambdas in the program that are the possible targets of encapsulated calls in a program P (defined in section 2). Let $Clo^\# = \{C_i w_i \mid i \in [1, n]\}$ be the closure constructions in the model of P representing the lambdas Λ . In the model program, the dispatch functions `Appi`

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

corresponding to the encapsulated calls invoke the function $f_i^\#$ (the translation of f_i) in each case $C_i w_i$ (see Fig. 5 and the illustration Fig 6(b)). Let $DispCalls = \{f_i^\#(x, z_i) \mid i \in [1, n]\}$ be the calls invoked by such `Appi` functions. Let $Props = \{\rho_i \mid i \in [1, n]\}$ be a set of boolean-valued expressions (properties) in E_{spec} defined on the captured argument y_i of the lambda $e_i \in \Lambda$ (i.e., ρ_i has only y_i as free variable). We augment the function-level assume/guarantee rules with the following condition: if each property ρ_i is cache monotonic, and hold at the point of creation of the lambda e_i for the state of the cache at that point, it can be assumed to hold at the point of dispatch. Formally,

Modular reasoning with creation-dispatch rule

- I. For each `def` $f \ x := \{pre\} \ e \ \{post\}$, $\models_P pre \rightarrow post[e/res]$
- II. For each call site $c \notin DispCalls$, $\models_P path(c) \rightarrow pre(c)$
- III. (Cache monotonicity) For each $\rho_i \in Props$
 $\models_P (st_1 \subseteq st_2 \wedge \llbracket \rho_i \rrbracket_P st_1) \rightarrow \llbracket \rho_i \rrbracket_P st_2$
- IV. For each closure construction site $c = C_i w_i$ in $Clo^\#$
 $\models_P path(c) \rightarrow (\llbracket \rho_i \rrbracket_P st(c))$
- V. For each call site $c = f_i^\#(x, z_i)$ in $DispCalls$
 $\models_P (path(c) \wedge \llbracket \rho_i[z_i/y_i] \rrbracket_P st(c)) \rightarrow pre(c)$

In the above rules, $st(c)$ denotes the cache-state expression propagated by the translation function $\llbracket \cdot \rrbracket_P$ to an expression c in the model program. Note that there is exactly one cache-state expression reaching every point in the model program by the definition of the translation shown in Fig. 5. For instance, the state expression reaching the line 20 of Fig. 6(b) is `st`, whereas the state expression reaching the line 29 is `nst`.

While the above reasoning holds irrespective of the how the properties ρ_i are chosen for each lambda e_i , we use a particular strategy in our implementation. For each $e_i = \lambda x. f_i(x, y_i)$, we choose ρ_i to be the disjuncts of the precondition of the call $f_i(x, y_i)$ that only refer to the captured variable y_i . E.g. for the model shown in Fig. 6(b), our approach would verify that (a) `concrUntil` is a cache monotonic property: $\models_P (st_1 \subseteq st_2 \wedge \text{concrUntil}(s, i, st_1)) \rightarrow \text{concrUntil}(s, i, st_2)$, and (b) that the property `concrUntil(u, 1, n-1, nst)` holds at the point of creation of the closure `Take(n-1, u, 1)` at line 29. The property `concrUntil(s1, n1, st)` is assumed to hold while checking the precondition of call to `take` at line 20. With this extension we do not need any more preconditions than what is stated in the program to verify the program.

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

Solving parametric verification conditions. To solve the assertions generated by assume/guarantee reasoning and infer values for

the holes, we extend the template inference algorithm proposed by us in previous research [51, 52] and implemented in the Leon verification and synthesis system [13, 71] (leondevel.epfl.ch). Fig. 7 shows a block diagram of the inference algorithm which we briefly describe in the sequel. Given an assume/guarantee assertion $\models_P e_1 \rightarrow e_2$ the VC generation phase converts it to a quantifier-free formula (VC) of the form $\phi(\bar{x}, \bar{a})$, where the variables $\bar{a}$ corresponds to the numerical holes, such that the assume/guarantee assertion holds if there exists a assignment ι for $\bar{a}$ such that $\phi \iota$ is unsatisfiable. (The VC could be thought of as a $\exists \forall$ formula where the holes are existentially quantified, and the rest including uninterpreted function symbols are universally quantified.)

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

Solving linear parametric formulas with sets. Given a linear parametric VC of the form: $\phi(\bar{x}, \bar{a})$, the solution for $\bar{a}$ that will make ϕ unsatisfiable is computed using an iterative but terminating algorithm that progresses in two phases: an existential solving phase (phase I), and a universal solving phase (phase II). Phase I discovers candidate assignments ι for the free variables $\bar{a}$. It initially starts with an arbitrary guess, and subsequently refines it based on the counter-examples produced by Phase II. Phase II checks if the candidate assignment ι makes ϕ unsatisfiable. That is, if $\phi \iota$ is unsatisfiable. If not, it chooses a disjunct $d(\bar{x}, \bar{a})$ satisfiable under ι that has only numerical variables by axiomatizing uninterpreted functions and algebraic datatypes in a complete way [52]. This numerical disjunct is then given back to phase I. Phase I generates and solves a quantifier-free nonlinear constraint $C(\bar{a})$, based on Farkas’ Lemma [20], to obtain the next candidate assignment for $\bar{a}$ that will make $d(\bar{x}, \bar{a})$ and other disjuncts previously seen unsatisfiable. Each phase invokes the Z3 [25] and CVC4 [8] SMT solvers in portfolio mode on quantifier-free formulas. This algorithm was shown to be complete for linear parametric formulas belonging to the combined theory of real arithmetic, uninterpreted functions and algebraic datatypes [52]. Below we extend this result to include sets. (Proof detailed in the technical report [50].)

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

Encoding Runtime Invariants and Optimizations. For improving automation and performance, we explicitly encode certain invariants (described below) ensured by the runtime that are not captured by the model, during VC generation. (a) We encode the refer-

<i>B</i>	<i>I</i>	(dynamic/static) * 100		(optimal/static) * 100	
		steps	alloc	steps	alloc
<i>sel</i>	10k	99	99	100	100
<i>prims</i>	1k	60	89	82	100
<i>fibs</i>	10k	99	99	100	100
<i>hams</i>	10k	86	83	98	100
<i>slib</i>	10k	65	75	85	88
<i>rtq</i>	2^{20}	93	83	97	87
<i>msort</i>	10k	90	91	96	97
<i>deq</i>	2^{20}	48	48	59	62
<i>num</i>	2^{20}	94	97	96	100
<i>conq</i>	2^{20}	72	54	82	72
<i>lcs</i>	1k	88	100	95	100
<i>levd</i>	1k	90	100	96	100
<i>hmem</i>	10k	79	100	92	100
<i>ws</i>	10k	99	100	100	100
<i>ks</i>	1k	94	100	99	100
<i>pp</i>	10k	77	70	88	84
<i>vit</i>	100	42	100	86	100
Avg.		81	88	91	94

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

ential transparency of the functions in the input program (namely, that the result of the function is independent of the cache state) in the VC in the following way. In principle, this corresponds to the axiom $\forall x, st_1, st_2. (f^\sharp(x, st_1))_{.1} = (f^\sharp(x, st_2))_{.1}$ for every function $f^\sharp$ in the model. We encode this axiom efficiently by adding the predicate $f^\sharp(x, st_1) = UF_f(x)$ for every application of $f^\sharp$ in the VC, where UF_f is a unique uninterpreted function for $f^\sharp$. This helps achieve a completely functional reasoning for correctness properties needed for proving resource bounds. (b) We encode the monotonic evolution of the cache by adding the predicate: $st \subseteq f^\sharp(x, st)_{.2}$ for every application of $f^\sharp$ in the VC. (c) Also, whenever the counter-example guided solving fails, we unfold only calls along the disjuncts $d(\bar{x}, \bar{a})$ encountered during the solving phase (referred to as *targeted unfolding* [52]). This prevents unfolding along paths known to be unsatisfiable in the VC thus mitigating the overheads due to defunctionalization.

5. Evaluation

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

Benchmark statistics. Fig. 8 shows selected benchmarks that were verified by our approach. Each benchmark was implemented and specified in a purely functional subset of Scala extended with our specification constructs. We carefully picked some of the most challenging benchmarks from the literature of lazy data-structures and dynamic programming algorithms. For instance, the benchmark *rtq* has been mentioned as being outside the reach of prior works (section Limitations of [22]). For each benchmark, the figure shows the total lines of Scala code and the size of the compiled JVM byte code in columns *LOC* and *BC*. The benchmarks comprise a total of 4.5K lines of Scala code and 1.2MB of bytecodes.

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

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

The column *T* shows the number of functions with resource bound templates, and the column *S* the number of specification functions. We do not verify resource bounds of specification functions but only verify their termination [78]. The column *AT* shows the time taken by our system rounded off to minutes to verify the specifications and infer the constants. As shown by the figure, all benchmarks were verified within a few minutes. The column *Resource bounds* shows a sample bound for steps and alloc resource. The constants in the bound were automatically inferred by the tool.

We verified a total of 123 bounds each for steps and alloc. Many bounds used recursive functions, and almost 20 bounds had nonlinear operations. (Nonlinear operations like $\lfloor \log \rfloor$ are expressed as a recursive function that uses integer division: $\log(x) = \text{if}(x \geq 2) \log(x/2) + 1$ else (base cases). Their properties like monotonicity are manually proved and instantiated.) A few bounds were disjunctive (like the bound shown in Fig. 1, and *conq*). However, in our experience, the most challenging bounds to prove were the constant time bounds of scheduling-based lazy data structures viz. *rtq*, *deg*, *num*, and *conq* due to their complexity.

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

time resource usage was 81% of what was inferred statically for steps, and is 88% for alloc. In all cases, the inferred bounds were sound upper bounds for the runtime resource usage. We now discuss the reasons for some of the inaccuracy in the inferred bounds.

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

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

due to templates is a primary contributor for inaccuracy, especially in benchmarks where the accuracy is lower than 80% (such *Viterbi* and *prims*). In the sequel, we discuss the benchmarks and the results of their evaluation in more detail.

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

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

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

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

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

6. Related Work

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

Static Resource Bounds Analysis. Automatic static inference of resource bounds of programs has been an actively researched area. Some of the recent works include [1, 2, 7, 17, 29, 32, 35, 38, 46, 53, 70, 84]. Being fully automated, these approaches target simpler programs and bounds that depend on less complex invariants compared to our approach. Another related line of work include semi-automatic formal frameworks amenable to deriving machine-checked proofs of resource bounds [9, 23, 66, 67]. In particular, Sands [66, 67] present a theoretical framework for reasoning about lazy evaluation. We are not aware of any machine-checked proofs for the resource bounds of the lazy data structures considered in our study. Recent works on resource analysis have started incorporating user specifications. Alonso et al. [3] presented an approach where resource bounds are specified by users as templates. Carbonneaux et al. [18] presented a system to verify stack space bounds of C programs written for embedded systems using a quantitative Hoare logic. Previously, we proposed an approach [52] for inferring resource bounds using user-defined templates and specifications for first-order, non-lazy functional programs with algebraic datatypes.

Coinductive datatypes. Leino and Moskal [48] use coinduction to verify programs with possibly infinite lazy data structures. They do not consider resource properties of such programs. Blanchette et al. [14, 15] present a formal framework for soundly mixing recursion and corecursion in the context of interactive theorem provers.

Imperative and Higher-order Verification. Verification Systems such as [16, 26, 36, 47, 60, 65, 83] and interactive theorem provers [10, 19, 56] have been used to verify complex, imperative programs. Automation in our system appears above the one in interactive provers, and could be further improved using quantifier instantiation, induction, and static analysis [11, 33, 63]. While most approaches for imperative programs target a homogeneous, mutable heap, in this work we consider an almost immutable heap except for the cache, and use a set representation to handle mutations to the cache efficiently. We believe that similar separation of heap into mutable and immutable parts can benefit other forms of restricted mutation like *write-once* fields [6].

Works such as [28, 42–44, 54, 55, 73, 75, 77–79, 81, 82] target correctness verification of higher-order, functional programs. Many of these systems allow users to write contracts on function-valued parameters, or refinement predicates on function types [28, 75]. We are not aware of any contract-based verifiers for higher-order programs that allow specifying resource properties, as in our approach. Our approach allows named functions, with contracts and resource templates, to be used inside lambdas. However, it disallows contracts on function-valued parameters and instead provides intensional-equality-based constructs to specify their properties. Though this makes the contracts very specific to the implementation, it has the advantage of reducing specification burden for closed or encapsulated programs. Supporting contracts on function-valued parameters that can refer to resource bounds would be an interesting future direction to explore.

References

- [1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.
- [2] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis Symposium, SAS*, pages 117–133, 2010.
- [3] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *Static Analysis Symposium, SAS*, pages 405–421, 2012.
- [4] H. Apfelmus. Quicksort and k-th smallest elements. 2009.
- [5] A. W. Appel. Intensional equality $\Rightarrow$ for continuations. *SIGPLAN Not.*, 31(2), Feb. 1996.
- [6] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, Oct. 1989.
- [7] M. Avanzini, U. D. Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *International Conference on Functional Programming, ICFP*, pages 152–164, 2015.
- [8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification, CAV*, pages 171–177, 2011.
- [9] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1):79 – 103, 2004.
- [10] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
- [11] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *Computer Aided Verification, CAV*, 2013.
- [12] R. Bird and P. Wadler. *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd., 1988.
- [13] R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An overview of the Leon verification system. In *Sacana Workshop*, 2013.
- [14] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: a proof assistant perspective. In *International Conference on Functional Programming, ICFP*, pages 192–204, 2015.
- [15] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In *European Symposium on Programming, ESOP*, pages 359–382, 2015.
- [16] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: making parametric shape analysis competitive. In *Computer Aided Verification, CAV*, pages 221–225, 2007.
- [17] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, pages 13:1–13:50, 2016.
- [18] Q. Carbonneaux, J. Hoffmann, T. Ramanandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation, PLDI*, 2014.
- [19] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation, PLDI*, pages 234–245, 2011.
- [20] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification, CAV*, 2003.
- [21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.
- [22] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of Programming Languages, POPL*, pages 133–144, 2008.
- [23] N. Danner, J. Paykin, and J. S. Royer. A static cost analysis for a higher-order language. In *Workshop on Programming languages meets program verification, PLPV*, pages 25–34, 2013.
- [24] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2008.
- [25] L. M. de Moura and N. Björner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, pages 337–340, 2008.
- [26] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for java. In *Object-oriented Programming Systems Languages and Applications, OOPSLA*, pages 213–226, 2008.
- [27] M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming, IWACO*, page 58, 2003.
- [28] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming, ICFP*, pages 48–59, 2002.
- [29] A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Programming Languages and Systems - 12th Asian Symposium, APLAS*, pages 275–295, 2014.
- [30] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *International Conference on Functional Programming ICFP*, pages 36–47, 2002.
- [31] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, Feb. 2011.
- [32] S. Gulwani, K. K. Mehra, and T. M. Chilikimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages, POPL*, 2009.
- [33] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Computer Aided Verification, CAV*, 2015.
- [34] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
- [35] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource Aware ML. In *Computer Aided Verification, CAV*, pages 781–786, 2012.
- [36] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of NASA Formal Methods, NFM*, pages 41–55, 2011.
- [37] N. D. Jones and N. Bohr. Termination analysis of the untyped lambda-calculus. In *Rewriting Techniques and Applications, RTA*, pages 1–23, 2004.
- [38] S. Jost, K. Hammond, H. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Principles of Programming Languages, POPL*, pages 223–236, 2010.
- [39] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *Foundations of Software Engineering, FSE*, pages 105–116, 2006.
- [40] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
- [41] G. Klein, P. Derrin, and K. Elphinstone. Experience report: Sel4: Formally verifying a high-performance microkernel. In *International Conference on Functional Programming, ICFP*, pages 91–96, 2009.
- [42] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, Feb. 2010.
- [43] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Principles of Programming Languages, POPL*, pages 416–428, 2009.
- [44] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CE-GAR for higher-order model checking. In *Programming Language Design and Implementation, PLDI*, pages 222–233, 2011.
- [45] J. Launchbury. A natural semantics for lazy evaluation. In *Principles of Programming Languages, POPL*, 1993.
- [46] D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, Apr. 1988.
- [47] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

- [48] K. R. M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *Formal Methods, FM*, pages 382–398, 2014.
- [49] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.
- [50] R. Madhavan, S. Kulal, and V. Kuncak. Verifying resource bounds of programs with lazy evaluation and memoization, EPFL-REPORT-215783. Technical report, EPFL, 2016.
- [51] R. Madhavan and V. Kuncak. Symbolic resource bound inference, EPFL-REPORT-190578. Technical report, EPFL, 2014.
- [52] R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In *Computer Aided Verification, CAV*, pages 762–778, 2014.
- [53] J. A. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *International Conference on Logic Programming, ICLP*, pages 348–363, 2007.
- [54] P. C. Nguyen and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *Programming Language Design and Implementation, PLDI*, pages 446–456, 2015.
- [55] P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Soft contract verification. In *international conference on Functional programming, ICFP*, pages 139–152, 2014.
- [56] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
- [57] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.
- [58] C. Okasaki. Simple and efficient purely functional queues and dequeues. *Journal of Functional Programming*, 5:583–592, 10 1995.
- [59] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [60] R. Piskac, T. Wies, and D. Zufferey. Grasshopper - complete heap verification with mixed specifications. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 124–139, 2014.
- [61] A. Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, EPFL, 2014.
- [62] A. Prokopec and M. Odersky. Conc-trees for functional and parallel programming. In *Languages and Compilers for Parallel Computing, LCPC*, pages 254–268, 2015.
- [63] A. Reynolds and V. Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 80–98, 2015.
- [64] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
- [65] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles of Programming Languages, POPL*, pages 105–118, 1999.
- [66] D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Imperial College, University of London, 1990.
- [67] D. Sands. Complexity analysis for a lazy higher-order language. In *European Symposium on Programming, ESOP*, pages 361–376, 1990.
- [68] D. Sereni. *Termination analysis of higher-order functional programs*. PhD thesis, University of Oxford, UK, 2006.
- [69] H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *International Conference on Functional Programming, ICFP*, pages 165–176, 2012.
- [70] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Computer Aided Verification CAV*, pages 745–761, 2014.
- [71] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Symposium on Static Analysis SAS*, 2011.
- [72] W. Swierstra. Stream: A library for manipulating infinite lists. <https://hackage.haskell.org/package/Stream-0.4.7.2/docs/Data-Stream.html>. 2015.
- [73] S. Tobin-Hochstadt and D. V. Horn. Higher-order symbolic execution via contracts. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 537–554, 2012.
- [74] P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In *European Symposium on Programming, ESOP*, 2015.
- [75] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *International Conference on Functional Programming, ICFP*, pages 269–282, 2014.
- [76] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967.
- [77] N. Voirol, E. Kneuss, and V. Kuncak. Counter-example complete verification for higher-order functions. In *Symposium on Scala*, pages 18–29, 2015.
- [78] N. Voirol and V. Kuncak. Automating verification of functional programs with quantified invariants, EPFL-REPORT-222712. Technical report, EPFL, 2016.
- [79] D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. In *Principles of Programming Languages, POPL*, pages 431–442, 2013.
- [80] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.
- [81] D. N. Xu. Hybrid contract checking via symbolic simplification. In *Workshop on Partial Evaluation and Program Manipulation, PEPM*, pages 107–116, 2012.
- [82] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for haskell. In *Principles of Programming Languages, POPL*, pages 41–52, 2009.
- [83] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *Programming Language Design and Implementation, PLDI*, 2008.
- [84] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis Symposium, SAS*, pages 280–297, 2011.