Verifying Resource Bounds of Programs with Lazy Evaluation and Memoization

EPFL-REPORT-215783

Ravichandhran Madhavan\(^1\) and Viktor Kuncak\(^2\)

\(^1\) ravi.kandhadai@epfl.ch
\(^2\) viktor.kuncak@epfl.ch
EPFL

Abstract. We present a new approach for specifying and verifying re-
source utilization of functional programs that use lazy evaluation and
memoization. Our approach verifies asymptotically tight bounds of com-
explex programs that rely on deep sharing and aliasing of lazy heap refer-
ences, as well as algorithms that use memoization extensively. Our ap-
proach can also find counter-examples for incorrect specifications or im-
plementations. We demonstrate the effectiveness of the approach by veri-
fying the resource bounds of several complex benchmarks, some of which
have never been formally verified before. Examples include Okasaki’s
real-time queues, deques, lazy data structures based on numerical repre-
sentations such as lazy binomial heaps, catenable queues, dynamic pro-
gramming algorithms like weighted scheduling, and packrat parsing.

1 Introduction

This paper explores the area of computer-aided verification of resource bounds
for systems with unbounded data expressed in a functional programming model
that supports memoization and lazy evaluation.\(^3\) The suitability of functional
model for reasoning is witnessed by the fact that fragments of functional pro-
gramming languages are at the core of proof assistants ACL2, Coq, HOL, and
Isabelle, which have been used successfully to perform large-scale software and
hardware verification \([18, 22, 23, 27]\). In this paper, we consider programs that
further rely on lazy evaluation and memoization. These features are important
as they improve the running time, often by orders of magnitude, while preserving
the functional model for the purpose of reasoning about the result of the com-
putation. For instance, Okasaki \([30]\) showed that enqueuing into and dequeuing
from a persistent, immutable queue can be performed in worst-case constant time
using lazy evaluation. (Appendix A shows the implementation and specifications
of this data structure.) Such data structures are a good fit in concurrent and
parallel applications as they do not have observable side-effects on the result of
computation \([32]\). Lazy data structures based on numerical representations also

\(^3\) Throughout this paper, memoization specifically refers to caching of outputs of a
function for each distinct input encountered during an execution.
exhibit cache oblivious behavior \cite{24}. Increasingly, mainstream languages like Java 8, C#, and Scala, are adopting lazy evaluation through libraries such as LINQ, and Scala Streams. Memoization too is a widely used feature, particularly in dynamic programming algorithms, which itself have numerous applications. (Appendix A shows an implementation and specification of the Knapsack algorithm that is verified by our approach.)

However, the price we need to pay for this increased performance is that reasoning about resources usage, running time in particular, becomes more imperative and tricky. Theoretical analysis of running time bounds of lazy data structures have been subjects of separate research works, and have also remained open in some cases (e.g. lazy pairing heaps described in page 79 of \cite{30}). The problem is important in practice as well, as demonstrated by many unresolved, long-standing discussions on online forums concerning resource usage of lazy programs (bit.ly/1RIbumO, bit.ly/1mROFkU). This makes a strong case for enhancing tool support for verification of resource usage of such programs. But, a major impediment to automation is the imperative and higher-order reasoning required to handle these features in their entirety. Expressions that are lazily evaluated (called as suspensions) not only cache the result, but, like higher-order functions, may be stored in the heap and shared. A naive approach to model them at a low-level using only mutation has high annotation and verification overheads (see section 3.2).

Contributions of this paper are the following:

- We develop a specification language that is adequate for stating resource bounds and other state-dependent properties of programs with first-class suspensions. For instance, our approach allows expressing that every suspension in a datatype is evaluated when a procedure is called (function isConcrete of Fig. 1), or that a variable points to the first unevaluated node of a stream (function firstUneval of Fig. 8).

- We propose and implement a system for checking resource bounds and state-dependent properties such as isConcrete, and firstUneval. Our approach soundly handles expressive contracts of higher-order suspensions, and also discovers counter-examples for the contracts. Two key aspects of our approach are encoding closures as algebraic datatypes, and modeling the mutable part of the state differently from immutable parts, using sets, in order to generate formulas efficiently decidable by the state-of-the-art SMT solvers.

- We use our approach to verify (asymptotically) precise resource bounds of real-world programs, comprising more than 2K lines of Scala code with specifications, that extensively use memoization and lazy evaluation in sophisticated ways (see Fig 6). Our case studies include the Conqueue data structure, which is a core component of the data-parallel library of Scala \cite{32}, (http://bit.ly/1KdVvq), and a packrat parser described in \cite{15}. In fact, we also discovered (and fixed) a missing corner case of a constant-time, persistent deque implementation presented in \cite{30} from our tool’s failure to prove the exhaustiveness of match.
sealed abstract class Stream[T]

case class SCons[T](x: T, tail: $[Stream[T]]) extends Stream[T]

case class SNil[T]()
extends Stream[T]

def isConcrete[T](l: $[Stream[T]]): Boolean = {
  l.isEvaluated && (l ∗ match {
    case SCons(_, tail) => isConcrete(tail)
    case _ => true
  })
}

@invariant private def rotate[T](f: $[Stream[T]], r: List[T], a: $[Stream[T]]): Stream[T] = {
  require(r.size == (f ∗).size + 1 && isConcrete(f))
  (f.value, r) match {
    case (SNil(), Cons(y, _)) => SCons(y, a)
    case (SCons(x, tail), Cons(y, r1)) =>
      SCons(x, $(rotate(tail, r1, SCons(y, a)))) //creates a suspension of rotate
  }
}

ensuring (res => res.size == (f ∗).size + r.size + (a ∗).size && time <= 30)

Fig. 1. Definitions and specifications of a Stream, and a lazy rotate operation.

A limitation of our approach is that we cannot handle cyclic or infinite datatypes. However, this limitation can be overcome in some cases by finitizing an infinite data structure by parameterizing the number of elements that may be accessed, and by expressing use cases of cyclic datatypes using memoization.

2 Overview of the Approach

We present an overview of our approach using the stream data structure, and the functions shown in Fig. 1. The functions are a part of a larger real time queue benchmark verified by our approach. The verified implementation of the benchmark is shown in Appendix A. Since lazy evaluation performs memoization at reference level, in the following sections, we will focus on lazy evaluation and highlight the variations needed for memoization at the right points.

Suspension and Contract Syntax. Fig. 1 defines a lazy stream in our syntax, which is a minor extension of Scala [29]. Akin to a list, the Stream datatype is defined using two constructors SCons and SNil denoting non-empty and empty streams respectively. But, the second argument of SCons, denoting the tail of the stream, is a lazy reference. In our syntax, suspensions (i.e, references to lazily evaluated expressions) computing a value of type T have the type $[T]. Suspensions of an expression e (of type T) are created using $(e), which has type $[T]. Given a suspension s, s.value evaluates the suspension and returns its value (see line 13). As a side-effect, it also caches the value for reuse. We refer to this as forcing a suspension. (These constructs are based on the language described by Okasaki [30] for combining strict and lazy evaluation.) Our language supports
checking whether a suspension \( s \) has been forced through the \( \text{s.isEvaluated} \) construct. We also support a unary operator \( * \) that evaluates a suspension without caching the result for reuse. We enforce that \( \text{s.isEvaluated} \) and \( * \) are used only in expressions (or functions) used in the specifications. In specifications, using \( s* \) is better than using \( s\.value \) as the former does not have any side-effects, and hence could make verification easier. The contracts of the functions are specified using the \text{require} and \text{ensuring} constructs. In our language, specifications of functions can refer to the resources consumed by the function. The variable \text{time} used in the specification of the function \text{rotate} refers to the time consumed by an invocation of the function with respect to a user-configurable cost model. By default, it refers to the number of primitive operations performed by the function. Fig. 3 formally defines the syntax of the expressions and types for an indicative subset of our language.

**Expressing Stateful Properties.** Consider the function \text{rotate}. It reverses the list \( r \) and appends it to the lazy stream \( f \), using the stream \( a \) as a temporary storage, which is initially set to empty. Essentially, \( \text{rotate}(f,r,a) = f++\ \text{reverse}(r++)+a \). \( (f \) and \( r \) actually represent the front and rear parts of the real-time queue data structure described in Appendix A.\) However, the function performs its work lazily: every call to \text{rotate} constructs the first element of the result, and returns a stream whose tail is a suspended recursive call. The specifications of \text{rotate} assert properties of the function that hold before and after the execution. Consider the property on the sizes of the arguments. This property is independent of the state i.e., it does not depend on whether the suspensions in the input list are forced or not. \( \text{size} \) is a straightforward recursive function that traverses the stream, and is shown in Fig. 8 of Appendix A.\) In contrast, the property \text{isConcrete} is state dependent: it returns \text{true} if every node of the argument stream has been forced, \text{false} otherwise. Notice that the postcondition also asserts a constant time bound for the function \text{rotate}. The requirement that \text{isConcrete}(f) holds at the beginning of the function is crucial for proving the time bound. Otherwise, forcing \( f \) at line 13 (using \( f\.value \)) may invoke a previously suspended call to \text{rotate}, thus resulting in a cascade of forces. The need to account for such chaining of lazy suspensions is one complication in verifying resources bounds of lazy programs. In our model (discussed shortly), such chains would reduce to a sequence of possibly mutually recursive function invocations. Our system verifies all the specifications shown in Fig. 4 including the resource bounds of the functions.

**Overview of the Model.** Our approach consists of three main phases. In the first phase, we translate the source program to a new program, referred to as \text{model}, that has only first-order recursive functions without suspensions. We also introduce additional lemmas (expressed as recursive functions that return true for all inputs) in the model that assert auxiliary properties required for the soundness. In the subsequent phases, we instrument the model to track resources, and create verification conditions that can be decided using \text{k-induction}, and off-the-shelf SMT solvers. Fig. 2 shows snippets of the model generated in the initial phase for the program shown in Fig. 1. The model uses two new constructs that are not a part of the source language: \text{assume} and \text{fun}. The predicate \text{assume}(cond)
sealed abstract class Stream[T]
case class SCons[T](x: T, tail: LazyS[T]) extends Stream[T]
case class SNil[T]() extends Stream[T]

sealed abstract class LazyS[T]
case class Rotate[T](f: LazyS[T], r: List[T], a: LazyS[T], v: Stream[T]) extends ...
case class Eager[T](v: Stream[T]) extends LazyS[T]

def rotate_m[T](f: LazyS[T], r: List[T], a: LazyS[T], st: Set[LazyS[T]]) = {
  require(r.size == eval'(f).size + 1 && isConcrete_m(f, st))
  val (v, nst) = eval(f, st)
  (v, r)
    match {
      case (SNil(), Cons(y, _)) => (SCons(y, a), nst)
      case (SCons(x, tail), Cons(y, r1)) =>
        val na = Eager(SCons(y, a))
        val result = fun(tail, r1, na)
        (SCons(x, Rotate(tail, r1, na, result), nst)) // closure creation
      } ensuring (res => assume(res._1 == fun(f, r, a)) &&
                  st == res._2 && ... && time <= 30)
}

def eval(cl: LazyS[T], st: Set[LazyS[T]]) = cl match {
  case Eager(v) => (v, st)
  case Rotate(f, r, a, v) =>
    assume(r.size == eval'(f).size + 1 && isConcrete_m(f, st)) // pre of rotate_m
    val rotres = rotate_m(f, r, a, st)
    assume(rotres._1 == v)
    (rotres._1, rotres._2 union { cl })
  }
  def eval'(cl: LazyS[T]) = eval(cl, fun()).1

Fig. 2. Snippets of the model generated by our tool for the program of Fig.1

calculates that the predicate cond holds at the specified point. We use this to encode
those facts that are guaranteed to hold in the source but are not inferrable from
the model. The construct fun(x) denotes an uninterpreted function over x. Below
we present an overview of the two main aspects of the model: (a) modeling
the higher-order semantics of suspensions, and (b) capturing the side-effects of
forcing suspensions.

Modeling Higher-Order Aspects. We use algebraic datatypes (ADTs) to
represent suspensions. The datatype LazyS with constructors Rotate and Eager
models the suspensions created in the program shown in Fig.1. The constructor
Rotate models the suspensions of the rotate function (e.g. created at line 16 of
Fig.1). It accepts as arguments all the parameters of the rotate function, and an
argument representing the result of rotate that is initially bounded to an uninter-
preted value. (This succinctly encodes acyclicity of the stream as discussed in
section 3.) It is instructive to think of the instances of Rotate as closures of the
rotate function. The constructor Eager models the suspensions created during implicit lifting of eagerly-evaluated expressions to suspensions, which is supported by our language (e.g. expression SCons(y, a) in line 16 of Fig. 1).

Notice that in the model for rotate (shown by the function rotate_m) creation of suspensions is replaced by ADT constructions (lines 15 and 17), and that the types $\text{Stream}[T]$ are replaced by LazyS. Forcing of suspensions using s.value is modeled by the function eval. The function pattern matches on every case of LazyS, and invokes the corresponding target with the arguments stored in the closure. For the Eager case, we simply return the result value stored in the closure. As shown in the body of eval (line 24), we assume the preconditions of the targets of the closure at the invocation points. They are checked independently at the point of creation of closures. Interestingly, observe that the possible cascading of forces in the rotate method discussed earlier translate to the mutually recursive call sequence between rotate_m and eval in the model.

Capturing Side-effects of Caching. We use a set of closures to capture the side-effects of caching, and treat it similar to a global state. At any given point, the set contains all the closures that have been forced until that point. Every function that queries/creates/forces suspensions accept the state as a parameter and return a new state if the state is updated (e.g. rotate_m). For instance, the parameter st of rotate is the state at the beginning of an invocation of the function, and the second component of the result (res._2) is the state after the invocation. The state is updated by the invocations of eval, which models forcing of suspensions. As shown by lines 25 and 27, the state returned by eval is the union of the state returned by the target and the closure that is evaluated.

Notice that we assume in the postcondition of rotate_m that the result of evaluation of rotate is an uninterpreted function of the non-state arguments of rotate (line 18). This essentially encodes that the state has no effect on the result of the computation, which is otherwise lost in the translation. In general, we also assert that the state increases monotonically ($\text{st} \subseteq \text{res._2}$) in the postconditions. However, in the case of rotate, we assert (and verify) a stronger property that the states are equal: $\text{st} == \text{res._2}$ as the function was annotated with @invstate (see Fig. 1).

Instrumenting for Resources. We instrument the model for time by accumulating the cost of primitive operations performed by the expressions in the model. Section 3.2 outlines our instrumentation approach. We specially handle the function eval so that it returns a constant value if the closure cl belongs to the input state st, otherwise it returns the time taken by the target. It returns a constant value in the Eager case regardless of the state. We use the publicly available Leon verifier [6] to verify the instrumented code. With this overview we explain the approach in more detail in the next section.

3 The Approach

Fig. 3 show the syntax of an indicative subset of the language handled by our implementation. Our language has constructs exclusively available only in spec-
ifications. Their syntax is described by the non-terminal \textit{spec} in Fig. 3. The expression $e.\text{isSuspOf}(f)$ checks whether $e$ is a suspension of a call to the function $f$. This is particularly useful in preconditions to constrain the possible types of suspensions that can be passed as parameters to the functions. The constructs \texttt{inSt}, \texttt{outSt}, and \texttt{withSt} are useful for accessing the implicit state of the functions. \texttt{time} refers to the running time of the function. By default it counts the number of primitive operations executed by the function. Our implementation supports other resources such as the stack space usage, and \texttt{depth[,]}, which is a measure of parallel execution time. The annotation \texttt{@memoize} marks functions that need to be memoized, and the annotation \texttt{@invstate} instructs the underlying verifier to establish and utilize the state invariance of the annotated function. For brevity, we omit the discussion of standard constructs, and also those described in section 2. Without loss of generality, we assume that the expressions are in A-normal form (in which the arguments of operations are variables), also that the argument of $\$($ is a function invocation, and that it is not already a suspension.

\begin{verbatim}
prog ::= fundef+

fundef ::= annot def f(vars) = pre? e post?

pre ::= require(e)

post ::= ensuring(v => e)

vars ::= v* 

pat ::= v | C(pat*)

model ::= assume(e) | fun(vars)

Types
T ::= B | $[B]
B ::= Int | Bool | A[T]

Fig. 3. Syntax of the language and types used in formalizations. The syntax is based on Scala. $v, f, A,$ and $C$ denote the set of variables, function names, ADTs and ADT constructors, respectively. \texttt{op} is a set of primitive operations.

3.1 Modeling Lazy Evaluation and Memoization

In this section, we describe the source-to-source translation that given a program belonging to the language shown in Fig. 3 models it using a subset of the language that does not use lazy evaluation or memoization. The generated model is \textit{sound} for proofs: if every function in the model is verified, then the source program can be considered verified. It is also \textit{sound} for counter-examples, i.e, counter-examples to the functions in the model yield counter-examples to the specifications of the source program. The latter property is particularly useful in large programs, where it is easy to introduce bugs not only in the implementation, but in specifications as well.

Representing Suspensions as ADTs. For every type $\$[B]$ in the source program we create an ADT denoted \texttt{LazyB}. We define the constructors of the ADT
as follows. Let $f_1, \ldots, f_n$ denote the functions returning a value of type $B$ whose invocations are suspended (i.e., passed as arguments to $\xi$) in the source program. (Recall that we required that only function calls could be suspended.) For each function $f_i$ we introduce a new constructor $C_{f_i}$ of LazyB. The parameters of $C_{f_i}$ include all the parameters of $f_i$ and two additional parameters: one that denotes the result of the suspension, and another denoting an optional unique identifier. The unique identifier field of the ADTs enables switching between reference equality and value equality for comparing suspensions. While lazy suspensions use reference equality, suspensions created out of memoized functions (discussed shortly) use value equality to mimic the effect of a cache from arguments to the result. We also introduce a default constructor $\text{Eager}[B]$ of LazyB with one argument to model implicit lifting of eagerly-evaluated expressions to suspensions.

**Set Abstraction of the Heap Effects.** Conceptually, lazy suspensions are restricted form of heap-allocated objects with mutable fields. Generic approaches for verifying programs with mutable heap, typically, represent them as global maps from locations to values \cite{5}. In contrast, in our approach, we separate the heap into a logically immutable part, which is represented using an algebraic datatype, and a mutable part, which is represented as a set. Fig. 4 pictorially depicts the concrete heap, and the abstract representation used by our approach for the stream data structure shown in Fig. 1.

**Concrete Heap**

![Concrete Heap Diagram]

**Abstract Heap**

![Abstract Heap Diagram]

Fig. 4. Abstraction of the heap used by our approach. Nodes labelled $f(a_i)$ are suspensions (or thunks) that cache the result using a mutable field res. $F(a_i)$ are the ADTs representing closures of $f(a_i)$ in our model. v is an immutable field denoting the result of $f(a_i)$, and initialized to an uninterpreted value.

In Fig. 4 the heap locations related by tail form the immutable part. The mutable part of the heap are the fields of the suspensions (or thunks) that cache the result of evaluation (shown as res in Fig. 4). The mutable fields logically act as a map from suspensions to their result. However, since the values stored in the map are known a priori – they are given by the result of the suspension – we can collapse the mutable part to the keys of the cache, which is the set of suspensions that are evaluated. In summary, the heap at a given point in the program is modeled using an immutable list, and the set of suspensions that are evaluated at that point, as shown by the Fig. 4. The set of suspensions grow monotonically during the execution.

This separation of the heap based on mutability (a) allows exploiting the efficient decision procedures for ADTs for the immutable parts, and (b) reduces specification and verification overhead by capturing state-invariance of properties.
depending only on the immutable part (e.g. the size property in the precondition of rotate in Fig. 1). We also exploit the monotonicity of the set representation of the mutable state while verifying the contracts as discussed in section 3.2. (This idea of using different abstractions for modeling distinct heap regions has been explored by prior works 5,42, but not for lazy evaluation.)

Fig. 5 formally defines the translation of expressions of the source program to the model. We define a translation function $\llbracket \cdot \rrbracket : (Expr \times State) \rightarrow Expr$ that, given a state $st$, maps a source language expression $e$ to an expression of the model $e_m$. If $e$ updates the state, e.g. by forcing a suspension, then $e_m$ would be a pair, where the first part $e_m..1$ computes the value of $e$, and the second part $e_m..2$ returns the new state. In Fig. 5 for brevity, we assume that $st$ is a set of ADTs representing suspensions. However, in the general case where the source program may use multiple lazy types $\llbracket B \rrbracket$, the state is a tuple of sets, one for each type of suspension $\llbracket B \rrbracket$ used in the program. (Collapsing the state into a single set

<table>
<thead>
<tr>
<th>Base Expressions:</th>
<th>Lazy Expressions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[v] st$</td>
<td>$\llbracket f(vars) \rrbracket st$</td>
</tr>
<tr>
<td>$[v_1 op v_2] st$</td>
<td>$\llbracket 5(f(vars)) \rrbracket st$</td>
</tr>
<tr>
<td>$[if(v) e_1 else e_2] st$</td>
<td>$\llbracket 5(f(vars)) \rrbracket st$</td>
</tr>
<tr>
<td>$[{val v = e_1; e_2}] st$</td>
<td>$\llbracket 5(f(v)) \rrbracket st$</td>
</tr>
<tr>
<td>Function Definitions:</td>
<td>(when pre, body, and post updates state)</td>
</tr>
<tr>
<td>$[def f(p_1, \cdots, p_n) =$</td>
<td>$\ldef f_m(params, st) =$</td>
</tr>
<tr>
<td>$\require(pre)$</td>
<td>$\require([{pre} st]..1)$</td>
</tr>
<tr>
<td>$body$</td>
<td>$\require(body) st$</td>
</tr>
<tr>
<td>$\ensuring(r=&gt;post)]$</td>
<td>$\ensuring(r=&gt;&gt;)$</td>
</tr>
<tr>
<td>$\ensuring(r=&gt;&gt;)$</td>
<td>$\ensuring(r=&gt;&gt;)$</td>
</tr>
</tbody>
</table>

Fig. 5. Translation of expressions of the source program to the model.
would result in loss of type information available at the source level, and would also pose problems in encoding VCs into a sorted logic.) The function $eval(v, st)$ pattern matches on the possible cases of the ADT denoting the suspensions, and invokes the corresponding target with the arguments stored in the ADT, similar the one shown in Fig. 2. In the general case, we create a separate $eval$ function for every lazy type in the source program.

Observe that we model an invocation of a memoized function $f(vars)$ as creating a suspension of $f$ and immediately forcing it. This essentially adds a closure of $f(args)$ to the state thereby recording that it has been evaluated. It also accurately captures the resource usage of the call, which depends on the state. Moreover, in the case of memoized functions, the additional overhead of creating and storing fresh identifiers in the ADTs (using fresh) is eliminated. Implementing fresh requires another global state. (In our system, we, by default, disable the fresh identifier creation in all cases for efficiency, at the expense of deviating, in rare cases, from the Scala semantics.) Below we discuss some of the important aspects of the model.

**Translation of Contracts.** Notice that in the translation of a function definition shown in Fig. 5, the state updated by the contracts are ignored (e.g. we select only the first part of the tuple returned by $([\text{pre}] st)$). Hence, specifications that rely on suspensions forced in the contracts cannot be proven in our system. Though this is a deviation from the semantics of Scala, this is a desirable property. Notice that the $\text{inSt}$ and $\text{outSt}$ constructs are bounded to the states at the entry and exit of the function. These provide more control over the specification. For instance, it allows providing hints to the underlying verifier by manually instantiating axiomatic specifications \cite{20}, such as the monotonicity of $\text{isConcrete}$: $\text{st1} \subseteq \text{st2} \land \text{isConcrete}(\text{lst1}) \Rightarrow \text{isConcrete}(\text{lst2})$, over the input and output states. Such instantiations were often necessary in our benchmarks.

**Encoding Acyclicity.** We encode the acyclicity of data structures in the source program by associating a result field with closures. The field is initialized to an uninterpreted value, and is assumed to be equal to the result of evaluation of the closure in the $eval$ function (e.g. see line 26 of Fig. 2). (We initialize it to an uninterpreted value to avoid eagerly introducing new functions calls, which may modify the resource consumption.) We now illustrate how this succinctly captures acyclicity of the data structures with suspensions. Say we are interested in proving that for a stream $l$ defined in Fig. 1, $l \neq l.\text{tail}.\text{value}$. The translation function converts the expression to $l \neq eval(l.\text{tail}, \text{st})._1$ The assumption $\text{rotres}._1 == v$ in the body of $eval$ implies, in the above context, that $l.\text{v} == eval(l.\text{tail}, \text{st})._1$ which entails the property (because the ADT axioms imply that $l \neq l.\text{v}$).

### 3.2 Verification of the Model

**Higher-Order Contract Checking.** The presence of suspensions with contracts introduces additional dimensions to contract checking, namely, (a) verifying that the preconditions of the suspensions hold at indirect call sites, and
(b) utilizing the postconditions of the suspensions at indirect call sites. The latter requirement is trivially satisfied by \texttt{eval} that explicitly invokes the targets of the suspensions. (We assume a closed world in which the targets of all closures belong to the program.) However, ensuring the former requirement is more involved. An issue here is that checking the preconditions of suspension at indirect call sites could dramatically increase the specification burden. To understand the problem, consider the function \texttt{rotate} defined in Fig. 1. It forces the suspension \( f \) passed as an argument (line 13 of Fig. 1), which say is a suspension of the form \( \text{rotate}(f',r',a') \). To ensure the precondition of \texttt{rotate} for this indirect call, we require the path condition to the indirect call to imply the predicate: \( r'.size == ... && \text{isConcrete}(f', \text{st}) \). This requirement propagates to the precondition of \texttt{rotate}, which in turn introduces further preconditions to \texttt{rotate} by the above reasoning. In essence, we have to replicate the property for the entire chain of suspensions reachable from \( f \) (forced or otherwise). In the model, this corresponds to the sequence of \texttt{Rotate} ADT instances: \( f, f.f, f.f.f.. \).

To circumvent this problem, we check the preconditions of the suspensions at \textit{creation time}, and assume them at the invocation time (as shown in line 24 of Fig. 2). This is sound only if the preconditions are \textit{monotonic} with respect to the state: \( \forall s, s', s \subseteq s' \land \text{pre}(s) \Rightarrow \text{pre}(s') \). With this strategy, checking the preconditions of \texttt{rotate} requires only two properties to hold: (a) \texttt{isConcrete} is monotonic (the other part of the precondition is state-independent), and (b) the precondition holds for the arguments of the suspension created at line 16. That is, \( r1.size == ... && \text{isConcrete}(\text{tail}, \text{nst}) \) holds. We encode these in the model as additional functions that need to be proved. Note that requiring preconditions to be monotonic with respect to the state is not a limitation, as they can always be pushed to the postconditions, e.g. as \( \text{pre} \Rightarrow \text{post} \). However, in that case the additional specification overhead cannot be avoided.

\textbf{Resource Instrumentation.} We follow the approach of Madhavan and Kuncak [28] to instrument the expressions of the model with resource usage. For every expression \( e \), the instrumented expression (denoted \([ e ]_R\)) is a pair, where the first part computes the value of \( e \), and the second part the resource \( R \) consumed by \( e \), which is a function (\(#\)) of the resources consumed by the sub-expressions of \( e \). For instance, \([ v_1 \text{ op } v_2 ]_R = (v_1 \text{ op } v_2, \#(\text{cost}(v_1), \text{cost}(v_2), \text{cost}(\text{op})))\), where \text{cost} is a user-configurable cost model. In the case of \textit{time}, \# is plus, whereas in the case of \textit{stack} (stack memory usage) and \textit{depth} \([9]\), \# is \text{max}. In general, for function calls \([ f(var) ]_R = (f'(var)).1, \#(f'(var)).2, \text{cost}(\text{call})\), where \( f' \) is the instrumented version of \( f \). We specially define the instrumentation of \texttt{eval} as: \([ \text{eval}(cl, st) ]_R = (\text{eval'}(cl, st)).1, \text{if}(cl \in st) \text{cost}(\text{lookup}) \text{ else } \#(\text{eval'}(cl, st)).2, \text{cost}(\text{call})\), in order to account for the effect of caching.

\textbf{VC Generation and Solving.} We use the publicly available Leon verifier \([6, 36, 37]\) to verify the resource-instrumented model, exploiting its efficient unfolding algorithm for handling recursive function. As an optimization, we check the non-resource properties independently on the uninstrumented program, and assume them while checking the instrumented program. To decide the unfolded VCs we
use Z3 and CVC4 SMT solvers (with the recent extensions for sets) in portfolio mode. To make verification easier for users, we implemented a new induction tactic in Leon that would infer the induction schema to use based on the recursive functions called in the body of a function. It can be invoked using \texttt{@traceInduct} annotation.

4 Case Studies and Results

We implemented the data structures and algorithms shown in Fig. 6 in our language (an extension of purely functional Scala), and verified the correctness of the benchmarks using our system. All metrics presented in this section were obtained on a machine with a 8 core, 3.6 GHz, intel i7 processor, having 32GB RAM, running Ubuntu operating system. In Fig. 6 we summarize the results on two main aspects of our system: (a) expressibility of complex programs and specifications, and (b) the performance of verification, which is indicative of the efficiency of the encoding.

![Fig. 6. Benchmarks comprising of \sim 2K lines of Scala code (excluding comments), 183 functions, with \sim 700 lines of specifications to be proven, and 52 time bounds.](image)

The column \texttt{loc} shows the lines of code per benchmark excluding comments. The columns \texttt{Prop.} and \texttt{Hint} show the number lines of specifications in the benchmarks classified loosely into essential properties (\texttt{Prop.}), and proof hints (\texttt{hint}), to provide insights into the level of specifications. The former denotes the high-level facts that are needed to ensure the resource bounds, e.g. predicates like \texttt{isConcrete} of Fig. 1 or data structure invariants like \texttt{valid} of Fig. 8, or axioms like \texttt{monotonicity}. Proof hints denote specifications added to enable the underlying verifier and SMT solvers prove the properties. Mostly, they are either manual instantiations of axioms, or encoding of induction tactic, or annotations like \texttt{@inline} and \texttt{@invstate}. As indicated by the figure, the specifications
were mostly high-level. But, in complex benchmarks like Conqueue there was considerable overheads. The column bound shows the number of time bounds that were verified, a sample of which is shown in the column sample. The column Time shows the total time taken to verify the model. The column VC shows the total number of VCs generated during the verification process. Each VC checks a specific condition like whether a precondition of a suspension holds at a creation site, and often has many recursive functions, which are unfolded and fed to SMT solvers: CVC4 \[3\] and Z3 \[13\]. The remaining columns summarize the number of VCs that were decided first by one solver over the other (column \(S\)), and the average time it took (column time/S). (The solvers were run in portfolio mode). Observe that even though Z3 solves more VCs faster than CVC4, the ones in which CVC4 was more effective were much harder as shown by the long solving time (column time/S). These were VCs that required deep reasoning over sets. The Fig. 6 shows that all benchmarks were verified within hundred seconds, which is significant considering the complexity of the benchmarks. Below we describe the benchmarks used in the study in detail.

**Streams.** The benchmark is a collection of lazy operations over streams such as zipWith, concat, which are constant time operations, and a selection sort that returns the minimum in linear time. This benchmark also includes the examples of the related work of Vasconcelos et al. \[38\], except the Hamming program as it uses a cyclic stream. However, we verify a memoized implementation of the Hamming problem of identical complexity.

**Real-time Queue & Deque.** These data structures use the same idea as a simple immutable queue that has an amortized, constant running time for non-persistent usage. They use two streams: front and rear. The elements could be added to or removed from one or both the ends. At any instance, if the streams have unequal number of elements beyond a threshold factor, the elements are transferred between the streams to restore balance (similar to balancing a tree), which in the case of real-time queue is the rotate function of Fig. 1. But, the costly balance operation is performed incrementally, alongside other operations. For this purpose, the data structures are augmented with (one or two) schedules which are references to suspensions, possibly nested deep inside the streams, that correspond to the next step of an unfinished balance operation. During every operation, the schedules are forced so as to perform one additional step of the balance operation. A complex invariant ensures that all pending balance operations complete before the next balance operation begins. The resulting data structure has a constant running time, in the worst case, for all operations even under persistent usage. A complete implementation of the real-time queue data structure is shown in the Appendix A. Below we show a sketch of the definition of the deque data structure. Below, \(f\) and \(r\) are front and rear streams with respective schedules \(sf\) and \(sr\). firstUneval(s) returns the first unevaluated node of the stream \(s\), and is defined in Fig. 8.

```scala
case class Queue[T](f, lenf, sf, r, lenr, sr) {
  ...
  firstUneval(f) == firstUneval(sf) && lenf == (f.size).size && ...
  lenf <= 2*lenr + 1 && (sf)*.size <= min(2*lenr-lenf+2, 2*lenf-lenr+2) && ...
}
```
Queue: $0 \rightarrow S^1(1, 1 \rightarrow 0 \rightarrow S^2(1, 0 \rightarrow \cdots))$
Schedule: $S^1 \rightarrow S^2 \rightarrow \cdots$

Main Data structure Invariants:
(a) There is a zero in the queue before $S^1$.
(b) There is a zero between any $S^i$ and $S^{i+1}$.
(c) Every element between $S^i$ and $S^{i+1}$ in the schedule is forced.
(d) After fully propagating $S^i$, in the resulting queue the first unforced suspension is $S^{i+1}$.

// invariant(a)

```python
def zeroPrecedesLazy(q: $\mathbb{N}$)
    if q.isEvaluated:
        q = match {
            case Spine(Zero(), _, tail) => true
            case Spine(_, _, tail) => zeroPrecedesLazy(tail)
            case Tip() => true
        }
    else false
```

Fig. 7. Some invariants of the lazy numerical representation benchmark. $S^i(c, q)$ is an unevaluated suspension of a function with two arguments: a carry c, and a queue q.

We verify that all the invariants of the queue and deque hold after every operation, and also the (asymptotically) precise running time of all operations. We also fixed a missing corner case of the rotateDrop function shown in Fig 8.4 of [30], which was unraveled by the tool.

**Bottom-up Merge Sort.** This is a lazy, bottom-up implementation of merge sort that allows accessing the $k^{th}$ element of the resulting sorted list in $O(kn)$ time. When $k = o(\log(n))$ we get asymptotic speed up due to lazy evaluation. The algorithm creates (logically) a tree of suspensions, in which each node is a suspension of the function merge applied over the output of the child trees. Forcing the suspension at a node of a tree will transitively force (through calls to merge) the suspensions of its children tree. One tricky aspect in verifying the benchmark was the need to bound such cascading of forces while accessing an element to $O(l.size)$. (The lazy merge function is often erroneously estimated as taking $O(\log(l.size))$ time. [bit.ly/1RIbum0].)

**Numerical Representation.** In this benchmark, we consider the lazy versions of a class of data structures like binomial heap [41] that are based on binary numbers. The data structures define unique structures representing $2^i$, $i \geq 0$. Inserting an element is similar to incrementing a binary number, and, occasionally, could result in rippling of carries. The lazy versions avoid this rippling by suspending the carry propagation. However, to prevent the forces from cascading, the data structures maintain a schedule, which is a stack of references to suspensions that correspond to unfinished carry propagations. Fig. 7 informally shows the invariants of the benchmark, and shows the implementation of an invariant. We prove that the insert operation takes constant time, and also that all invariants are preserved. (All invariants in the benchmark where necessary to prove the time bounds and also inductively imply themselves.) This serves as a generic proof for resource consumption of other instances of this representation like lazy binomial heap. Furthermore, we also formalize a particular instance: Conqueue data structure [32], which is a persistent queue with efficient concat, insert, and split operations, used to implement data parallel operations in Scala. Conqueue uses AVL like balanced trees optimized for Split to represent the digits.
Dynamic Programming Algorithms. We verify the precise resources bounds of dynamic programming algorithms expressed as memoized recursive functions. The Hamming Numbers benchmark computes a sorted list of numbers of the form $2^i3^j5^k$. The Knapsack benchmark computes the optimal way of packing items, each of value $v_i$ and weight $w_i$, into a knapsack of capacity weight that maximizes the total value of the items in the knapsack. (A resource verified implementation of Knapsack is shown in Appendix A.) The weighted scheduling benchmark optimally schedules $n$ jobs with (overlapping) start and finish times so that the total value of the scheduled jobs is maximized. The Packrat Parser benchmark is a memoized implementation of the packrat parser discussed by Ford [15], and uses the same parsing expression grammar as in that work. Below we briefly sketch the proof methodology we adopt for these algorithms, using the Hamming benchmark. Below we show snippets of the benchmark.

```scala
@memoize
def h(n: BigInt): Result = { // returns the nth hamming number
  require(n == 0 || n > 0 && deps(n-1))
  if(n == BigInt(0)) Result(1, 0, 0, 0)
  else { val Result(i, j, k) = h(n-1)
    threeWayMerge(h(i).v * 2, h(j).v * 3, h(k).v * 5, i, j, k)
  }
}

ensuring(res => res.i <= n && ... && time <= 140)
```

The property `deps` used in the precondition of `h`, asserts that when `h` is invoked with `n`, every call to `h` with arguments smaller than `n` are already cached. (`h(n).isCached` is equivalent to $(h(n)).isEvaluated`). These are possible invocations on which `h` depends. The precondition ensures a constant time bound for `h`. To satisfy the precondition we use a wrapper function that given a `n` invokes `h` in the order that ensures this property: `h(0), · · ·, h(n)`. Fig. 10 of Appendix A shows an example.

5 Related Work

Static Resource Analysis for Lazy Evaluation. Danielsson [12] present a lightweight type-based analysis for verifying time complexity of lazy functional programs, and applies it to implicit queues. As noted in the paper, the approach is limited in handling aliasing of lazy references, which makes them ineffective on our benchmarks. Vasconcelos et al. [38, 35] present a typed-based analysis for inferring bounds on memory allocations of Haskell programs. They evaluated their system on simple functions, which were included in our benchmarks, and hamming numbers (discussed in section 4). A main difference between our approach and the above works is that, our approach is targeted at verifying user-specified bounds, and is demonstrated on complex, real-world programs. Our approach also handles memoization, finds counter-examples, and can be extended to other resources as well.
Static Resource Bounds Analysis. Resource-Aware ML [19, 21] is a type-driven approach for inferring resource bounds of ML programs. Other automated systems for resource bounds inference include Speed [16, 43], and Costa [1]. These approaches do not seem to directly support lazy evaluation or memoization. While fully automated, these approaches target simpler programs and simpler bounds that depend on less complex invariants compared to our approach. Madhavan and Kuncak [28] present an approach that infers resource bounds from user-defined templates for non-lazy functional programs.

Coinductive datatypes. Leino and Moskal [26] use coinduction to verify programs with (possibly) infinite lazy data structures. They do not consider state-dependent or resource properties of such programs. Blanchette et al. [8, 7] 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 Dafny [25], JStar [14], TVLA [10, 34], GRASShopper [31], and Bedrock [11] have been used to verify complex, imperative programs. Automation in our system appears above the one in interactive provers, and we believe it could be further improved using quantifier instantiation, induction, and static analysis [4, 17, 28, 33]. The approaches mostly target a homogeneous, mutable heap. As discussed in section 3.2 using higher-order functions and mutation directly to model suspensions dramatically increases the contract overhead in our benchmarks. In contrast, in this work we consider an almost immutable heap, and devise an efficient representation for supporting limited mutations. We believe that similar hybrid representations can be explored for other forms of restricted mutation such as write-once fields [2], and Unique references. LiquidHaskell [39] supports verification of complex specifications expressed as type refinements. The system checks the simpler state-independent size invariants of the real-time queue data structure shown in Fig. 8 (http://bit.ly/1SeqdZ0). However, it does not allow expressing stateful properties like isConcrete or resources. Leon verification system supports verification of programs with higher-order functions [40], but it does not support lazy evaluation or memoization in its present form.

References

A Sample Implementations and Specifications of Benchmarks

Okasaki’s Real-Time Queue. Fig. 8 shows an implementation of the Okasaki’s real time queue data structure [30] in our syntax. It uses the datatype Stream, and the functions rotate, and isConcrete defined in Fig. 4. As shown in
Fig. 8, the real time queue data structure has three components: a lazy stream \( f \) denoting the front of the queue, a list \( r \) denoting the rear of the queue, and a lazy stream \( s \) denoting the schedule. We define the data structure invariants using the boolean-valued function \( \text{valid} \). Every public queue operation, namely \( \text{enqueue} \) and \( \text{dequeue} \), require that the \( \text{valid} \) property holds for the input queue, and also ensures that the property holds for the output queue (see the definitions of the functions in Fig. 8).

Consider the property \( \text{firstUneval}(f) == \text{firstUneval}(s) \) that relates the schedule and the front streams that is a part of the definition of \( \text{valid} \). The definition of \( \text{firstUneval} \) is shown in Fig. 8. It returns the first node in the stream that has not been forced. This property states that the unevaluated nodes of \( f \) and \( s \) are equal. In addition to this, the data structure also maintains the invariant that the size of the front is greater than the size rear, and that the size of the schedule is equal to the difference between the sizes of the front and the rear. These are succinctly captured by the second predicate of the function \( \text{valid} \). The specification of the \( \text{firstUneval} \) function asserts a few interesting properties of the function that are needed for verification.

The data structure uses the same idea as a simple immutable queue that uses two lists, namely front and rear, that has a constant, amortized running time for ephemeral (i.e., non-persistent) usage. The elements are enqueued to the rear list and dequeued from the front list. Once in a while, when there are very few or no elements in the front list, the dequeue operation would reverse the rear and append it to the front. This is captured by the \( \text{rotate} \) function of Fig. 1. The real time queue data structure uses a similar strategy, but it exploits lazy evaluation to perform the costly rotate operation incrementally, alongside the enqueue and dequeue operations. It thus achieves constant running time, in the worst case, for all operations even under persistent usage. For this purpose, it augments the queue with a \( \text{schedule} \) which is a reference to a suspension that corresponds to the next step of an unfinished rotate operation. The rotate operation itself is performed lazily: every call to \( \text{rotate} \) constructs the first element of the result, and returns a stream whose tail is a suspended recursive call (see Fig. 1).

During every enqueue and dequeue operation, if the schedule is non-empty, the head of the schedule is forced (line 32 of the function \( \text{createQ} \)). This corresponds to performing one step of the rotate operation. On the other hand, if the schedule is empty, which implies that there are no pending rotate operations, a new rotate operation is initiated (lines 34 to 35). Hence, whenever a rotate operation is initiated every node of the argument \( f \) is forced. This is asserted by the \( \text{isConcrete}(f) \) predicate used in the precondition of the \( \text{rotate} \) function, which is critical for proving the \( \mathcal{O}(1) \) time bound of \( \text{rotate} \) (as discussed in 2). Our system verifies the complete program shown in Fig. 8 and Fig. 1.

**Memoized Knapsack Program.** Figures 9 and 10 show the verified implementation of the \( \text{Knapsack} \) program. No additional specifications or hints except those shown in the figures were required to prove the benchmark.
sealed abstract class Stream[T] {
  def isEmpty: Boolean = { this match {
    case SNil() => true
    case _ => false }
  }
  def isCons: Boolean = { this match {
    case SCons(_, _) => true
    case _ => false }
  }
  def size: BigInt = { this match {
    case SNil() => BigInt(0)
    case SCons(x, t) => 1 + (t.size
  } ) ensuring (_ >= 0)
}

case class SCons[T](x: T, tail: $[Stream[T]]) extends Stream[T]

case class SNil[T](): extends Stream[T]

case class Queue[T](f: $[Stream[T]], r: List[T], s: $[Stream[T]]) {
  def isEmpty = f == SNil()
  def valid = firstUneval(f) == firstUneval(s) && (s.size == f.size - r.size
}
  def firstUneval[T](l: $[Stream[T]]): $[Stream[T]] = {
    if (l.isEvaluated) { l match {
      case SCons(_, tail) => firstUneval(tail)
      case _ => l
    } } else l
  } ensuring (res => ((res.size.isEmpty ==) isConcrete(l)) &&
    (res.value match {
      case SCons(_, tail) => firstUneval(l) == firstUneval(tail)
      case _ => true
    } ))
}

val rot = $(rotate(f, r, SNil()))
  Queue(rot, Nil(), rot)
}

val res = res.valid && time < = 60
}

val res = res.valid && time < = 120
}

Fig. 8. Complete implementation and specification of Okasaki's real-time queue data structure. isConcrete, and rotate are defined in Fig. [1] No additional specifications or hints were provided to verify this benchmark.
sealed abstract class IList { // a list of pairs of weights and values  
  def size: BigInt = {  
    this match {  
      case Cons(_, tail) => 1 + tail.size  
      case Nil() => BigInt(0)  
    }  
  }  
  ensuring(_ >= 0)  
}  

case class Cons(x: (BigInt, BigInt), tail: IList) extends IList  
  case class Nil() extends IList

val deps(i: BigInt, items: IList): Boolean = {  
  require(i >= 0)  
  knapsack(i, items).isCached && (if (i <= 0) true else deps(i - 1, items))  
}

@invstate
val maxValue(items: IList, w: BigInt, currList: IList): BigInt = {  
  require((w == 0 || (w > 0 & & (if (i <= 0) true else deps(i - 1, items))  
    // lemma inst begin  
    (currList match {  
      case Cons((wi, _), _) =>  
        if (wi <= w & & wi > 0) depsLem(w - wi, w - 1, items)  
        else true  
      case Nil() => true  
    })  
    // inst end  
    currList match {  
      case Cons((wi, vi), tail) =>  
        val oldMax = maxValue(items, w, tail)  
        if (wi <= w & & wi > 0) {  
          val choiceVal = vi + knapsack(w - wi, items)  
          if (choiceVal >= oldMax) choiceVal  
          else oldMax  
        }  
      case Nil() => BigInt(0)  
    }  
  )  
  ensuring(time <= 40 + currList.size + 20)
}

Fig. 9. A top-down memoized algorithm for Knapsack. The remaining part of the code is shown in Fig. 10. No additional specifications or hints were provided to verify the program.
@memoize

```java
def knapsackMem(w: BigInt, items: IList): BigInt = {
    require(w >= 0 && (w == 0 || deps(w - 1, items)))
    if (w == 0) BigInt(0)
    else maxValue(items, w, items)
} ensuring(time <= 40*items.size + 25)

val boundOnItems = 10

def knapsack(i: BigInt, w: BigInt, items: IList): IList = {
    require(w >= i && (i == 0 || i > 0 && deps(i - 1, items)))
    val ri = invoke(i, items)
    if (i == w) Cons((i,ri), Nil())
    else Cons((i,ri), knapSack(i + 1, w, items))
} ensuring(items.size <= boundOnItems ==> time <= 500 + (w - i + 1))

def invoke(i: BigInt, items: IList) = {
    require(i == 0 || (i > 0 && deps(i - 1, items)))
    knapSackMem(i, items)
} ensuring(i == 0 || deps(i, items)) == deps(i - 1, items) && //lemma inst
time <= 40*items.size + 40)

// deps is monotonic
@traceInduct
def depsMono(i: BigInt, items: IList, st1: Set[Mem[BigInt]], st2: Set[Mem[BigInt]]) = {
    require(i > 0)
    (st1.subsetOf(st2) && (deps(i, items) withSt st1)) ==> (deps(i, items) withSt st2)
} holds

// forall. x, x <= y && deps(y) => deps(x)
@traceInduct
def depsLem(x: BigInt, y: BigInt, items: IList) = {
    require(x >= 0 && y >= 0)
    (x <= y && deps(y, items)) ==> deps(x, items)
} holds
```

Fig. 10. The remaining part of the implementation of Knapsack shown in Fig. 9. No additional specification or hints were provided to verify the program.