Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 133–147

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Proving Resource Consumption of Low-level Programs Using Automated Theorem Provers

Jaroslav Ševčík [1],[2]

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK

Abstract

In this paper we use a program logic and automatic theorem provers to certify resource usage of low-level bytecode programs equipped with annotations describing resource consumption for methods. We have adapted an existing resource counting logic [1] to fit the first-order setting, implemented a verification condition generator, and tested our approach on programs that contain recursion and deal with recursive data structures. We have successfully applied our framework to programs that did not involve any updates to recursive data structures. But mutation is more tricky because of aliasing of heap. We discuss problems related to this and suggest techniques to solve them.

Keywords: Theorem provers, bytecode, resource-counting logic, verification condition generator, Grail

1 Introduction

Recent demand for higher security has spawned interest in more elaborate resource policies than memory safety or termination certification. An important example of such security policy is bounded memory consumption. Although there is a body of work in the area of type systems for resource consumption [11,7,6], there is little work so far on resource consumption certification that would exploit program logics combined with the power of fully automated theorem provers or decision procedures. In our work we explore the logical approach to proving and certifying resource consumption. Now that some automated provers are being equipped with ways to export their proofs we believe that this approach can provide us with explicit proofs of resource consumption, making the approach suitable for usage in proof-carrying code frameworks.

[1] Supported by *Resource Quantification in e-Science Technologies* project (ReQueST) funded by the UK EPSRC, grant EP/C537068.

[2] Email: j.sevcik@sms.ed.ac.uk

doi:10.1016/j.entcs.2007.02.065

In this paper we describe our framework and experiments for proving and certifying resource consumption of programs written in a subset of Java bytecode using program logics and automated theorem provers. Our initial motivation was to obtain certificates of memory consumption for low-level programs annotated with memory consumption on the level of methods. The annotations can be directly generated from the results of a static analysis for inferring heap space consumption of programs written in a high-level functional language called Camelot [17,13]. With the current state of the art in automated theorem proving this seems to be an easy way of obtaining the certificates in comparison with compiling into a typed assembly language and then generating the proofs from typing derivations.

Our approach has been to take an existing resource counting logic for a subset of Java bytecode [1] and redesign it to be first-order. This is part of our methodology, in which we experiment with a meta-theory in an interactive proof tool with an expressive logic and then "extract" the theory to an efficient and automatic solution. To express complex shape and size properties we have equipped the logic with the possibility of defining recursive predicates while guaranteeing consistency of the logic. Finally, we have implemented a verification condition generator for programs annotated with assertions on the level of methods and tested the framework on several examples using the Simplify decision procedure [8] as a prover back-end. We were able to prove resource consumption for possibly recursive programs on recursive data structures, which did not involve complicated reasoning about aliasing. This improves on the earlier work of Barthe *et al.* [3], which described a technique for proving memory consumption of simple programs without any recursive data structures. For programs involving updates of complicated data structures it is necessary to establish shape invariants to prove the resource properties. As general reasoning about reachability and aliasing is hard [10,12] we suggest an approximation of reachability and simple verification condition transformations that allow us to prove shape properties of in-place list reversal algorithm.

This paper is organised as follows. First, we will outline the language, the program logic and our axiomatisation in Sect. 2. Then, in Sect. 3, we will describe our verification condition generator and demonstrate a successful application to reasoning about running time in Sect. 4. In Sect. 5 we will discuss the problems arising when reasoning about programs with heap aliasing and suggest possible solutions to the difficulties. Sect. 6 concludes.

2 Language and Program Logic

Our work stems from the Mobile Resource Guarantees [17,20] project that investigated application of proof-carrying code technology to certifying resource consumption of programs compiled from a high-level functional language with explicit memory management to a subset of Java bytecode. This subset can be viewed as a simple functional language, named Grail.

In our work we have re-used the low-level Grail language and developed a first order variant of the program logic introduced in [1]. We will overview the language and the program logic in the following subsections.

2.1 Grail Language

Grail is a simple functional ML-like language. Just like in Java, a Grail program consists of classes, and the classes contain fields and methods. As opposed to Java, Grail method definitions consist of a trivial method body and local function definitions, which are needed to express looping constructs. Compared with full-blown functional languages, Grail is restrictive — among other limitations it does not support any polymorphism or higher order functions. All function calls must be tail-recursive, functions and function calls in one method must have the same actual and formal parameter list. These restrictions allow to compile a function call as a simple jump instruction in the Java bytecode, where the parameters correspond to local variables in current stack frame. This implies that the Java operand stack must be empty at each branch point, which trivially guarantees proper typing of the operand stack as required by the Java bytecode verifier. For a detailed discussion on Grail design refer to [1].

In the syntax we use v for values, a for variables or values, *top* for test operators (comparisons), *bop* for binary operators (addition, etc.), t for base types and e for expressions. Note that class and field definitions are omitted from the syntax as their definition is completely standard.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
v & ::= & \texttt{null} \mid i & \mathit{top} & ::= & \texttt{=} \mid \texttt{<>} \mid \texttt{<=} \mid \texttt{>=} \\
a & ::= & v \mid x & \mathit{bop} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \\
t & ::= & \texttt{Int} \mid \texttt{Loc} & & &
\end{array}$$

$$\begin{array}{rcl}
e & ::= & a \mid \mathit{bop}\ a\ a \mid \texttt{new}\ C \mid x.f_t \mid x.f_t\ :=\ a \mid C.m(\bar{a}) \mid \texttt{call}\ f \mid x.m(\bar{a}) \\
& & \mid \texttt{let val}\ x = e\ \texttt{in}\ e \mid \texttt{let val}\ () = e\ \texttt{in}\ e \mid \texttt{if}\ a\ \mathit{top}\ a\ \texttt{then}\ e\ \texttt{else}\ e
\end{array}$$

For a running example of a Grail program see Fig. 1, which shows an implementation of a class for single-linked lists with one method for determining the length of the list. Note that to pass different values of `m` and `acc` to the `rev_aux` from the `loop` function it is necessary to declare variables of the same name and use them as parameters of the call of `rev_aux` so that the parameter list for each function call is syntactically identical.

2.2 Operational Semantics

To formalise the language Grail we use a big-step operational semantics. The judgements of the operational semantics are in the form $E \vdash h, e \Downarrow h', v, \langle c, s\rangle$. The judgement says that expression e in environment E and heap h evaluates to value v and heap h' consuming time c and heap space s.

We model heaps as finite maps of type

$$\texttt{Loc}\ \rightarrow_{\mathrm{fin}} \texttt{Cls}\ \times (\texttt{IntFld}\ \rightarrow \texttt{Int}\ ) \times (\texttt{LocFld}\ \rightarrow \texttt{Int}\ )$$

```
class List {
 field private List tail
 field private int head

 method public static int len(List l) =
  let val m = l
      val acc = 0

      fun rev_aux (List m, int acc) =
       if m = null[List] then acc
       else loop(m, acc)

      fun  loop (List m, int acc) =
       let val  m = getfield m <List List.tail>
           val acc = add acc 1
       in rev_aux(m, acc) end
  in rev_aux(m, acc) end
}
```

```
      0:   aload_0; astore_1
      2:   iconst_0; astore_2
      4:   goto 7 // the method's "body"
      7:   aload_1; aconst_null
      8:   if_acmpeq 15
      12:  goto 17
      15:  aload_2; ireturn // then branch
      17:  aload_2
      18:  getfield <tail>; astore_1
      22:  iload_2; iconst_1; iadd; istore_2
      26:  goto 7
   // <- see label 4
```

Fig. 1. Class List implementing a method returning length of the list.

i.e. mapping locations to objects, where an object consists of a class name and a map from integer-type field names to integers and location-type field names to locations.

We also define some basic operations on heaps — $\mathtt{getf}_{\mathtt{Int}}$ and $\mathtt{getf}_{\mathtt{Loc}}$ retrieve field values given a heap, a location and a field name, $\mathtt{putf}_{\mathtt{Int}}$ and $\mathtt{putf}_{\mathtt{Loc}}$ store a value into an object's field in a given heap, $\mathtt{freshloc}(h)$ returns a location l such that $l \notin \mathrm{dom}(h)$. `alloc` updates a heap h with a new object of a given a class at the location $\mathtt{freshloc}(h)$ and `typeof` retrieves class name from a given heap and location. The types of the operations are summarised below.

$$
\begin{aligned}
\mathtt{getf}_t &: \mathtt{Heap} \times \mathtt{Loc} \times t\mathtt{Fld} \to t \\
\mathtt{putf}_t &: \mathtt{Heap} \times \mathtt{Loc} \times t\mathtt{Fld} \times t \to \mathtt{Heap} \\
\mathtt{freshloc} &: \mathtt{Heap} \to \mathtt{Loc} \\
\mathtt{alloc} &: \mathtt{Heap} \times \mathtt{Cls} \to \mathtt{Heap} \\
\mathtt{typeof} &: \mathtt{Heap} \times \mathtt{Loc} \to \mathtt{Cls}
\end{aligned}
$$

where $t \in \{\mathtt{Int}, \mathtt{Loc}\}$.

To illustrate the rules of the operational semantics we show the rule of the operational semantics[3] for an invocation of method m of an object pointed to by x with parameters $\bar{a}$.

$$
\frac{E\langle x\rangle = l \qquad \mathrm{typeof}(h, l) = C \qquad \{\mathit{this} \mapsto x; \overline{x_i \mapsto a_i}\} \vdash h, \mathrm{body}_{C,m} \Downarrow h_1, v, \langle c, s\rangle}{E \vdash h, x.m(\bar{a}) \Downarrow h_1, v, \langle (c+3), s\rangle}
$$

The first two assumptions of the rule require that the value of x in environment E is a reference to location l pointing to an object of type C in heap h. Then the rule states that evaluating $x.m(\bar{a})$ in heap h and environment E results in the same value v and heap h_1 as evaluating the body of method $C.m$ in heap h in an environment,

[3] The readers interested in all the rules of Grail may refer to [25, Appendix A].

which is constructed as a mapping from formal parameters to the actual parameters and mapping the variable *this* to value of x in the caller's environment. Moreover, it consumes the same amount of space s and takes 3 more units of time.

2.3 Grail Program Logic

We have based our program logic on the original program logic for Grail [1]. However, we could not use it verbatim since it uses higher-order logic as the language of assertions. The main change in our logic is replacing the explicit environment in the assertions of the original logic by substitutions in our logic.

The judgements of the program logic are in the form $\Gamma \rhd e : \phi$, where Γ is a set of assertions of the form $e : \phi$, e is a Grail expression, and ϕ is a first-order sorted logic assertion. The variables in e may not contain free variables h, c, s or v or any of their primed/indexed counterparts. The sorts of the logic are heaps (`Heap`), integers (`Int`), heap locations (`Loc`), field names for integers (`IntFld`), field names for locations (`LocFld`) and class names (`Cls`). Each free variable of ϕ may be either:

- a free variable of the expression e of the appropriate sort, i.e. of sort `Int` if it is a variable of type `int`, or of sort `Loc` otherwise;
- h or h' of sort `Heap`, denoting the initial and final heaps;
- c or s of sort `Int`, denoting the time and space consumption;
- v if the expression e is not of type `void`. The sort of the variable v must be `Int` if e is of type `int`, `Loc` otherwise, i.e. when the type of e is a reference.

For an expression e and a formula ϕ such that $\mathrm{fv}(\phi) \subseteq \mathrm{fv}(e) \cup \{h, h', v, c, s\}$ the informal meaning of $e : \phi$ is the following — for all h, h', v, c, s and all assignments to the free variables of e if e is executed in the heap h and results into the value v and the heap h' consuming time c and space s then ϕ holds. Note that our logic is a logic of partial correctness — it does not describe non-terminating executions. The formal definition of validity follows.

Definition 2.1 (Validity) Assertion A is valid for e, written $\models e : A$, if for all environments E, heaps h, h', values v of the appropriate base type and integers c, s

$$E \vdash h, e \Downarrow h', v, \langle c, s\rangle \qquad \text{implies} \qquad A[\bar{E}\langle\bar{x}\rangle/\bar{x}]$$

where $A[\bar{E}\langle\bar{x}\rangle/\bar{x}]$ is formula A with all free occurrences of each free variable x_i of e replaced by the value $E\langle x_i\rangle$ from the environment E.

We illustrate the logic on several interesting rules now [4]. The first rule describes the effect of allocating a new object:

$$\frac{}{\Gamma \rhd \texttt{new}\ C : v = \mathrm{freshloc}(h) \wedge h' = \mathrm{alloc}(h, C) \wedge c = 3 \wedge s = 1}$$

[4] The remaining inference rules for the logic are in [25, Appendix B]. The logic uses built-in predicates putf, getf, freshloc, alloc and typeof, which are axiomatised in [25, Appendix C].

The resulting value of the allocation will be a fresh location in the heap, the new heap will be the same as the original, except the new object allocated at the fresh location. The allocation also consumes 3 units of time and 1 unit of heap space — for simplicity we only count number of objects.

The most complicated rule is the one for virtual method invocation.

$$\frac{\forall C \in \texttt{Cls}.\, \Gamma \cup \{(x \cdot m(\bar{a}), A[\bar{a}/\bar{p}, x/self])\} \rhd body_{C,m} : A[\overline{pars_{C,m}}/\bar{p}, c+3/c] \qquad \mathrm{fv}(A) \subseteq \{p_1, \ldots, p_{|\bar{a}|}, h, h', v, c, s\}}{\Gamma \rhd x \cdot m(\bar{a}) : A[\bar{a}/\bar{p}, x/self]}$$

The rule says that if we can come up with a specification A of a method invocation $x \cdot m(\bar{p})$ such that using the specification for recursive calls we can prove that every method body m obeys A with resources decreased by the amount consumed by the invocation itself then our method invocation indeed fulfils the specification A, provided that we replace $\bar{p}$ with the actual parameters. Note that we need to prove the assertion for every method body with name m since we don't know the runtime type of the callee. We have proved soundness of the logic.

Theorem 2.2 *(Soundness) If* $\Gamma \rhd e : A$ *then* $\Gamma \models e : A$.

Proof. Following a modified version of proof from [1], for details refer to [25, Appendix D]. □

3 Verification Condition Generator for Grail

We have extended the Grail language to accommodate assertions on the method and function level and changed the existing Grail compiler to parse the annotated Grail programs and generate verification conditions in the Simplify format [8]. The verification conditions state that the assertion inferred by the rules of our program logic imply the specified assertion for each method body and function body. Moreover, the verification condition generator supports definition of well-founded predicates described below in Sect. 3.0.1 and generates axioms for the predicates. We also implement a simple procedure for instantiation of existential quantifiers that transforms all subformulae of the form $\exists x. x = t \wedge A$ to $A[t/x]$, provided that x is not free in t. This eliminates most quantifiers introduced by the rule for `let`[5].

In the translation to obligations we delete the sorts for heaps, locations and integers. For the sorts for fields and classes we use the standard translation from sorted to unsorted logics using predicates for sorts. Some non-theorems[6] in the sorted logic can become theorems when the sorts are deleted. However, Bouillaguet

[5] However, note that this technique can blow up the size of the formula exponentially, e.g. the generated specification of the program `let val` $\texttt{x}_1$ `= x*x ... val` $\texttt{x}_n$ `=` $\texttt{x}_{n-1}$`*`$\texttt{x}_{n-1}$ `in` $\texttt{x}_n$ `end` will result into an assertion of size $\Omega(2^n)$.

[6] For example, the existence of heap, i.e. $\exists h : \mathrm{Heap}.\, h = h$ cannot be proved from the axioms of our sorted logic, but it is a theorem in the unsorted one.

et al. [4] have shown that the theorems introduced by the translation are valid in all structures that are models for our axioms if the universes for the sorts of heaps, locations and integers have the same cardinality. This requirement is naturally satisfied in all sensible models of heaps — integers, locations and heaps are expected to be infinite countable sets. Moreover, any proof in the sorted logic is also a proof in the unsorted logic, hence all translated theorems from the sorted logic are theorems in the unsorted logic. In this sense, the translation is sound and complete.

3.0.1 Axiomatisation.

We assume that the underlying prover has some built-in notion of integers. This is realistic assumption for satisfiability modulo theories [2] decision procedures. First-order theorem provers will require some additional approximation of axioms for integers.

The axiomatisation of field names and class names specifies the constants for field/class names to be distinct and each value of sort field/class must be equal to one of the constants. The heap is axiomatised in a way similar to axiomatising arrays, using 12 axioms for all the combinations of our constructors and destructors of heaps and 2 axioms for freshness of *freshloc*. The following three axioms show the behaviours of the combination of *typeof* destructor with *alloc* constructor and the combination of *typeof* with *putf*.

$$\begin{aligned}\mathrm{typeof}(\mathrm{putf}_t(h,l,f,d),k) &= \mathrm{typeof}(h,k)\\ \mathrm{typeof}(\mathrm{alloc}(h,C),\mathrm{freshloc}(h)) &= C\\ k \neq \mathrm{freshloc}(h) \longrightarrow \mathrm{typeof}(\mathrm{alloc}(h,C),k) &= \mathrm{typeof}(h,k)\end{aligned}$$

The axioms are consistent because we can construct a simple model for the axioms, for example the concrete implementation of heaps from [1] is a model for the theory.

3.0.2 Well-founded predicates.

Because first-order provers accept any axioms we need a principled way to capture recursive definitions. The natural way to describe aliasing of recursive data structures like lists or trees is to use reachability. However, the straightforward definition of reachability is recursive and an axiom having the same predicate on both sides of an equivalence can make the theory inconsistent.

We ensure consistency by using well-foundedness of natural numbers – we allow only recursive definitions of new predicates in the form

$$p(\bar{x},n) \equiv (n = 0 \wedge A(\bar{x})) \vee (n > 0 \wedge B(\bar{x},n))$$

where the formula A cannot contain any occurrence of the predicate p and the formula B can only contain occurrences of p in the form $p(\bar{y}, n-1)$, where $\bar{y}$ is a vector of the same size as $\bar{x}$ and n is not quantified in B. We use the natural numbers because there are many decision procedures that can deal with arithmetic efficiently.

It is easy to see that if the original theory had a model, we can extend the model to the theory with this new axiom added. Therefore, the theory with the definition added remains consistent.

This allows for an easy definition of reachability.

$$\begin{aligned}\mathrm{reach}(h, x, y, n) \equiv{}& (n = 0 \wedge x = y) \vee \\ & \vee\ (n > 0 \wedge x \neq \mathrm{null} \wedge \exists f.\, \mathrm{reach}(h, \mathrm{getf}_{\mathrm{Loc}}(h, x, f), y, n - 1))\end{aligned}$$

4 Experiments with Time Consumption

The first results are encouraging — using our approach we are able to prove time consumption of simple recursive programs. In the example below we show a manually annotated version of the method for computing the length of a list from Fig. 1. The program illustrates reasoning about recursive data structures using the well-founded predicates to describe a list of length n. This allows us to assert that the static method `len` for computing the length of the list runs in linear time with respect to the length of the supplied list.

```
field private List tail
field private int head

predicate is_list(heap h, List l, int n) =
(n = 0 && l = null) ||
(n > 0 && typeof(h, l) = <class List> &&
     is_list(h, getfieldL(h, l, <field tail>), sub(n, 1)))
end

assert \forall int n . is_list(\heap, l, n) ->
                       \clock<add(100, mul(n, 100)) in
method public static int len(List l) =
let val m = l    val acc = 0

    assert \forall int n . is_list(\heap, m, n) ->
                           \clock < add(50, mul(n, 100)) in
    fun rev_aux (List m, int acc) =
        if m = null[List] then acc  else loop(m, acc)

    assert !(m = null) -> \forall int n . is_list(\heap, m, n) ->
                                          \clock < mul(n, 100) in
    fun  loop (List m, int acc) =
    let
        val m = getfield m <List List.tail>
        val acc = add acc 1
     in rev_aux(m, acc) end

in rev_aux(m, acc) end
```

The example has one method and two functions, which results into three verification conditions. For an illustration of a verification condition we show the proof obligation for the `loop` method. At first our generator computes a specification of `loop` using the rules of the program logic (we omit space consumption and some trivially eliminated quantifiers from the specification).

$$\begin{aligned}&\mathrm{Spec}_{\mathtt{loop}}(h, h', v, c, s, m, acc) \equiv \\ &\quad \exists c_1.\, (\exists c_2.\, (\exists c_3.\, (\forall n.\, \mathrm{is\_list}(h, \mathrm{getf}(h, m, \langle \mathrm{tail} \rangle), n) \longrightarrow c_3 < 50 + 100\, n) \wedge \\ &\qquad \wedge\, c_2 = c_3 + 1) \wedge c_1 = 3 + c_2) \wedge c = 2 + c_1 + 2\end{aligned}$$

The verification condition for `loop` then requires this to imply the annotated assertion:

$$\begin{aligned}&\mathrm{Spec}_{\texttt{loop}}(h, h', v, c, s, m, acc) \longrightarrow\\ &\quad\longrightarrow (m \neq \mathrm{null} \longrightarrow (\forall n.\ \mathrm{is\_list}(h, m, n) \longrightarrow c < 100\, n))\end{aligned}$$

for all h, h', v, c, s, m and acc.

The obligations are proved by Simplify in a fraction of a second. It seems likely that this example would extend to other cases where recursion pattern matches the definition of the predicate, such as proving running time of a search in a binary tree of bounded height. We have successfully applied our technique to other simple examples, such as multiplying integers using only addition.

5 Proving Space Consumption Using Free List

The story changes significantly if our aim is to prove memory consumption of programs containing recursive data structures with updates.

In our work we have adopted the approach taken by the Camelot language [17], which is a high level ML-like language with a type system that ensures linear heap space consumption. The essence of the Camelot language type system lies in bounding the minimal length of the free list of memory cells by a weighted sum of cells in input and output data structures for each function, where the weights are encoded into types of the function's parameters and result. The programmer is provided with a facility for freeing memory explicitly. The freed memory is stored in the free list for later usage. The reasoning about space consumption then amounts to reasoning about the length of the free list.

Generally, for reasoning about the length of the free list we need to preserve an important invariant — the free list must remain acyclic. To preserve the acyclicity every call to `move_to_free_list(x)` has to guarantee that the reference `x` is not a member of the free list. In Camelot, a linear type system is used to ensure that each memory cell is freed at most once. In our approach we need to assume such a layout of input parameters of methods that would imply preserving acyclicity of the free list.

We have chosen the in-place list reversal program to test our logic. See Fig. 2 for an implementation of the algorithm in Camelot. The program seems to be just like in Caml, it differs only in the explicit memory management. The `!Nil` construct means that the value `Nil` will be represented by `null` pointer and thus it will not consume any memory. The pattern `Cons(x, t)@_` instructs the compiler to free the memory used by the pattern. The memory will be then reused by the `Cons` that will become the head of the new accumulator. Hence the program does not use any additional memory, which is what the tool designed by Jost [13] infers.

When the Camelot code in Fig. 2 compiles into a Grail bytecode program it is necessary that we preserve the information about the acyclicity of all variables that represent lists in Camelot. If the acyclicity is not preserved then we could get a

```
type ilist = !Nil | Cons of int * ilist

let reverse m acc = match m with
        Nil          -> acc
      | Cons(x, t)@_ -> rev t (Cons(x, acc))
```

Fig. 2. In-place destructive list reversal in Camelot.

```
class List {
  predicate nreach(heap h, List l, List k, int n) =
    (n = 0 && l = k) ||
    (n > 0 && l <> null && typeof(h, l) = <class List> &&
     nreach(h, getfieldL(h, l, <field tail>), k, sub(n, 1))) end

  predicate reach(heap h, List l, List k) =
    \exists int n . nreach(h, l, k, n) end

  predicate acyclic(heap h, List l) =
    \forall List k . k <> null && reach(h, l, k) ->
      ! reach(h, getfieldL(h, k, <field tail>), k) end

  predicate separated(heap h, List l1, List l2) =
    \forall List k . k = null || ! reach(h, l1, k) || ! reach(h, l2, k) end

  field private List tail
  field private int head

  assert acyclic(\heap, l) -> acyclic(\newheap, \result) in
  method public static List reverse(List l) =
    let val acc = null[List]
        val m = l

        assert acyclic(\heap, m) && acyclic(\heap, acc) &&
               separated(\heap, m, acc) -> acyclic(\newheap, \result) in
        fun rev_aux (List m, List acc) =
            if m = null[List] then acc
            else loop(m, acc)

        assert m <> null &&  acyclic(\heap, m) && acyclic(\heap, acc) &&
               separated(\heap, m, acc) -> acyclic(\newheap, \result) in
        fun  loop (List m, List acc) =
        let val h = getfield m <int List.head>
            val t = getfield m <List List.tail>
            val () = putfield m <List List.tail> acc
            val acc = m
            val m = t
         in rev_aux(m, acc) end

    in rev_aux(m, acc) end
}
```

Fig. 3. Annotated list reversal algorithm in Grail.

cycle in the free list if we try to free the list later. Since our reasoning is modular we do not know whether the list will be freed, thus we need to be conservative and guarantee the acyclicity of the result[7]. Moreover, we must ensure separation of m and acc, otherwise `reverse` returns a "list" with a cycle. When we add these assertions to the compiled Grail code we obtain the program in Fig. 3.

Running our generator on the code will provide three verification conditions — the ones for the `reverse` method and the `rev_aux` function are trivial and they are easily proved by a theorem prover. However, the verification condition for the `loop` function needs to describe the effects of the heap cell mutation. Below is the

[7] These conditions are necessary, but they are not sufficient for modular reasoning. To make the specification usable we would need to establish frame condition as well — everything that is not reachable from m remains unchanged. We omit the frame condition as its treatment is not very different from the acyclicity.

inferred specification of `loop`.

$$\begin{aligned}
&\mathrm{Spec}_{\texttt{loop}}(h, h', v, c, s, m, acc) \equiv\\
&\quad \exists h_0.\, h_0 = \mathrm{putf}(h, m, \langle\mathrm{tail}\rangle, acc) \wedge\\
&\qquad (\mathrm{acyclic}(h_0, \mathrm{getf}(h, m, \langle\mathrm{head}\rangle)) \wedge \mathrm{acyclic}(h_0, m) \wedge\\
&\qquad\ \mathrm{separated}(h_0, \mathrm{getf}(h, m, \langle\mathrm{head}\rangle), m) \longrightarrow \mathrm{acyclic}(h', v))
\end{aligned}$$

The verification condition for `loop` requires that

$$\begin{aligned}
&\mathrm{Spec}_{\texttt{loop}}(h, h', v, c, s, m, acc) \longrightarrow\\
&m \neq \mathrm{null} \wedge \mathrm{acyclic}(h, m) \wedge \mathrm{acyclic}(h, acc) \wedge\\
&\mathrm{separated}(h, m, acc) \longrightarrow \mathrm{acyclic}(h', v)
\end{aligned}$$

for all h, h', v, c, s, m and acc.

As is well-known we need induction even for proving simple statements like that a change of a memory location x not contained in a list l does not affect acyclicity of the list l. In our language of assertions this translates to $\mathrm{acyclic}(h, l) \wedge \neg\mathrm{reach}(h, l, x) \longrightarrow \mathrm{acyclic}(\mathrm{putf}(h, x, f, v), l)$. Note that this happens for a simple operation like *cons h l* — we allocate a memory cell that is not in l and then update its head to h and its tail to l. Therefore, it is not surprising that the proof of the verification condition for `loop` needs to establish that the new m and acc remain acyclic and separated in the mutated heap using induction. This will then guarantee acyclicity of the result by the specification of `rev_aux`.

Resolution based theorem provers such as SPASS [27] or Otter [18] have no knowledge of integers and cannot instantiate induction, so they obviously fail to establish the obligation — it is not even provable in their logics. Although decision procedures have built-in knowledge of arithmetic they are much less powerful if we combine arithmetic with quantifiers, uninterpreted functions and uninterpreted predicates. As a result, neither Simplify [8] nor Harvey [9] can prove the obligations as they are generated.

In the following we will describe simplifications that we have performed by hand to make the obligations provable using these theorem provers. The first two steps could be automated easily, the last one still needs more work, but we believe it's promising future research. Observe that the alias change happens during the *putf* statement and the property we need to prove our obligation is

$$\triangleright\ m.tail := acc\ :$$

$$\mathrm{acyclic}(h, m) \wedge \mathrm{acyclic}(h, acc) \wedge \mathrm{separated}(h, m, acc)$$

$$\longrightarrow \mathrm{acyclic}(h', m) \wedge \mathrm{acyclic}(h', \mathrm{getf}(h, m, \langle\mathrm{tail}\rangle))$$

$$\wedge \mathrm{separated}(h', m, \mathrm{getf}(h, m, \langle\mathrm{tail}\rangle))$$

In the following we have concentrated on proving this property.

Replace inductive definitions with instantiation of induction.

None of the theorem provers used in our experiments is capable of instantiating induction automatically — all the provers that we have used accept a finite number of axioms, hence the induction has to be instantiated explicitly for them. Proving several programs by hand suggested the following useful instantiations.

$$\texttt{reach}(h,k,l) \equiv k = l \lor (k \neq \texttt{null} \land \texttt{reach}(h, \texttt{getf}(h,k,\texttt{tail}), l)) \tag{1}$$

$$\texttt{reach}(h,x,y) \land \texttt{reach}(h,y,z) \longrightarrow \texttt{reach}(h,x,z) \tag{2}$$

$$\begin{gathered}(\forall k.\texttt{getf}(h,k,\texttt{tail}) = \texttt{getf}(h',k,\texttt{tail}) \lor k = l) \land \texttt{reach}(h,a,b) \\ \longrightarrow \texttt{reach}(h',a,b) \lor (\texttt{reach}(h',a,l) \land \texttt{reach}(h', \texttt{getf}(h,l,\texttt{tail}), b)))\end{gathered} \tag{3}$$

$$\begin{gathered}(\forall k.\texttt{getf}(h,k,\texttt{tail}) = \texttt{getf}(h',k,\texttt{tail}) \lor k = l) \land \neg\texttt{reach}(h,a,b) \\ \longrightarrow \neg\texttt{reach}(h',a,b) \lor \texttt{reach}(h',a,l)\end{gathered} \tag{4}$$

Equation (1) defines reachability in a standard way, (2) postulates transitivity of reachability, (3) and (4) axiomatise behaviour of reachability after heap change. Using these instantiations as axioms the verification condition becomes provable in first order logic.

Remove the sort for field names and unfold non-recursive predicates.

In the logic, there is the sort of fields, that has only finite number of values. This introduces additional axioms and increases arity of the `getf` and `putf` predicate, which appears to increase the search space for the provers. Moreover, axiomatising non-recursive definitions of predicates, such as the ones of *separated* and *acyclic* in Fig. 3, makes the definitions available for both folding and unfolding, although we only need to unfold them. By replacing fields by specialised versions of `getf` and `putf` function symbols and by unfolding the non-recursive predicates we can make the obligations for list reversal become provable for SPASS[8]; however, relatively long times for proofs even for this simple example suggest that this is close to the limits of SPASS. Note that for resource counting it is highly desirable to be able to handle aliasing with the decision procedures because of the built-in arithmetic, which is lacking in the traditional automated theorem (SPASS, Otter and others), where it must be axiomatised separately, typically using Robinson arithmetic and instantiating induction for important properties — commutativity, associativity and others.

Eliminate the explicit heap variable.

From the counterexamples provided by Simplify it appears that it has difficulties with reasoning about the explicit heaps. Since our obligation for *putf* only contains heaps h and h', we can replace the predicates *reach* and *getf* with two versions without the explicit heap representing the predicates in h and the other one in h'. We will also need to introduce axioms to describe the relationship of the predicates

[8] See [25, Appendix D.1] for the transformed obligation.

in h and h'. In our case of obligation for *putf* predicate we can use the array-like select-store axioms for *putf* and *getf* to describe relationship of the two versions of *getf*. To capture the *reach* predicates we will use the axiom for effects of *putf* on reachability. The verification condition transformed this way[9] is provable by both Simplify and Harvey. Although we have not tested the approach on other verification conditions, we believe we could handle more complex obligations by eliminating all existentially quantified explicit heaps using a similar approach we have used for eliminating h and h'.

6 Conclusion

In our work we have applied logical methods to proving resource properties of low-level code. Using program logics and automated reasoning tools for proving resource consumption of programs seems to be practical for programs that do not use mutable data structures, but the direct approach attempted here fails in their presence. The reason for that is the inability of current theorem provers to use induction. However, even if we provide instances of the induction scheme as axioms to the provers the obligations for simple programs the obligations still appear to be too difficult for the provers. We have experimented with transformations of the generated verification conditions to make them provable by first order theorem provers. Using several simple transformations we were able to prove shape properties of the in-place list reversal algorithm, which is necessary for proving its memory consumption. Despite this progress it is hard to scale to more complicated examples.

A similar approach to ours was used in [3] to prove memory consumption of simple programs with no recursive data structures involved. As opposed to our technique, they have used ghost variables to describe memory usage in a Hoare-like program logic for bytecode [5]. Their work does not discuss treatment of programs involving recursive data structures and aliasing at all.

There are several other alternatives to our approach. Nguyen, David, Qin and Chin [22] apply separation logic to prove size and shape properties using a custom decision procedure based on folding and unfolding of recursive predicates, designed similarly to our well-founded predicates. By the usage of folding/unfolding in separation logic they neatly avoid the necessity of proving frame conditions using induction.

To handle shape analysis using logical tools [16] employs first-order theorem provers in combination with heuristics for instantiation of an induction scheme. Similar work of Lahiri and Qadeer [15] uses a decision procedure and several fixed instantiations of an induction scheme to prove shape properties of possibly cyclic data structures. We improve on their work by considering resources, their induction scheme is expressible using our well-founded predicates and the induction scheme for natural numbers. Interesting results in shape analysis were obtained by using limited, but decidable logics [28,21]. It is still an open question how to combine such logic with resources. It was suggested to us that Kuncak and Rinard [14] combine

[9] The transformed verification condition is in [25, Appendix D.2].

several logical approaches, including interfacing a first order theorem prover [4], to prove full correctness of data structure implementations. Although their tool Jahob cannot prove resource consumption, it would be interesting to use their techniques to reason about resources.

Meng and Paulson [19] utilise first-order theorem provers to prove typed theorems. This might seem to solve our problems instantly. However, their translation does not address the most difficult problem – the instantiation of induction schemes. Moreover, the authors admit that the significant number of axioms generated by the translation causes serious problems to current generation of first-order provers. We believe that specialised translation achieves significantly better results than this generic translation.

Another alternative is using type systems, such as [7,6,11] to guarantee the resource consumption. The type-theoretic approach suffers from similar problems as the logical one as it must describe aliasing of the heap as well, e.g. using [26]. The difference is in the price paid – the cost of using logical methods is undecidability whereas type systems for resources must limit the language considerably to be able to preserve aliasing invariants. Our experience shows that the cost of using theorem provers might still be too high in comparison to programming in a constrained language. This also applies to proof-carrying code setting — it appears to be easier to produce proofs from typing derivations than to reconstruct them from annotations on method level, at least with the present state of fully automated verification. We believe that shape-related problems, such as the ones presented in our work, should be part of the benchmarks for theorem provers and decision procedures [24,23] to encourage further development of the prover technology.

References

[1] Aspinall, D., L. Beringer, M. Hofmann, H.-W. Loidl and A. Momigliano, *A program logic for resource verification*, in: *Proceedings of 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, Lecture Notes in Computer Science **3223** (2004), pp. 34–49.

[2] Barrett, C., L. de Moura and A. Stump, *Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005)*, Journal of Automated Reasoning (2006).

[3] Barthe, G., M. Pavlova and G. Schneider, *Precise analysis of memory consumption using program logics*, in: *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods* (2005), pp. 86–95.

[4] Bouillaguet, C., V. Kuncak, T. Wies, K. Zee and M. Rinard, *Using first-order theorem provers in the Jahob data structure verification system*, in: B. Cook and A. Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007, Nice, January 14-16)*, Lecture Notes in Computer Science **4349** (2007).

[5] Burdy, L. and M. Pavlova, *Java bytecode specification and verification*, in: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (2006), pp. 1835–1839.

[6] Chin, W.-N., H. H. Nguyen, S. Qin and M. C. Rinard, *Memory usage verification for OO programs.*, in: C. Hankin and I. Siveroni, editors, *SAS*, Lecture Notes in Computer Science **3672** (2005), pp. 70–86.

[7] Crary, K. and S. Weirich, *Resource bound certification*, in: *Proceedings of the Symposium on Principles of Programming Languages*, Boston, Massachusetts, 2000, pp. 184–198.

[8] Detlefs, D., G. Nelson and J. B. Saxe, *Simplify: a theorem prover for program checking*, J. ACM **52** (2005), pp. 365–473.

[9] Fontaine, P., “Techniques for verification of concurrent systems with invariants,” Ph.D. thesis, Institut Montefiore, Universit de Lige, Belgium (2004). URL http://haRVey.loria.fr

[10] Grädel, E., M. Otto and E. Rosen, *Undecidability results on two-variable logics*, in: *STACS ’97: Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science* (1997), pp. 249–260.

[11] Hofmann, M. and S. Jost, *Static prediction of heap space usage for first-order functional programs*, in: *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices **38** (2003), pp. 185–197.

[12] Immerman, N., A. Rabinovich, T. Reps, M. Sagiv and G. Yorsh, *The boundary between decidability and undecidability for transitive-closure logics*, in: J. Marcinkowski and A. Tarlecki, editors, *18th International Workshop CSL 2004*, Lecture Notes in Computer Science **3210** (2004), pp. 160–174.

[13] Jost, S., `lfd_infer`*: an implementation of a static inference on heap space usage*, in: *SPACE 2004: Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2004.

[14] Kuncak, V. and M. Rinard, *An overview of the Jahob analysis system: Project goals and current status*, in: *NSF Next Generation Software Workshop*, 2006.

[15] Lahiri, S. K. and S. Qadeer, *Verifying properties of well-founded linked lists*, in: *POPL ’06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2006), pp. 115–126.

[16] Lev-Ami, T., N. Immerman, T. W. Reps, S. Sagiv, S. Srivastava and G. Yorsh, *Simulating reachability using first-order logic with applications to verification of linked data structures.*, in: R. Nieuwenhuis, editor, *CADE*, Lecture Notes in Computer Science **3632** (2005), pp. 99–115.

[17] MacKenzie, K. and N. Wolverson, *Camelot and Grail: resource-aware functional programming on the JVM*, in: *Trends in Functional Programming, Volume 4*, Trends in Functional Programming **4** (2004), pp. 29–46.

[18] McCune, W. W., “OTTER 3.0 Reference Manual and Guide,” (1994).
URL citeseer.ist.psu.edu/mccune94otter.html

[19] Meng, J. and L. C. Paulson, *Experiments on supporting interactive proof using resolution.*, in: D. A. Basin and M. Rusinowitch, editors, *IJCAR*, Lecture Notes in Computer Science **3097** (2004), pp. 372–384.

[20] *Mobile Resource Guarantees web* (2004), http://groups.inf.ed.ac.uk/mrg.

[21] Møller, A. and M. I. Schwartzbach, *The pointer assertion logic engine*, in: *PLDI ’01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (2001), pp. 221–231.

[22] Nguyen, H. H., C. David, S. Qin and W.-N. Chin, *Automated verification of shape and size properties via separation logic*, in: B. Cook and A. Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007, Nice, January 14-16)*, Lecture Notes in Computer Science **4349** (2007).

[23] Ranise, S. and C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, www.SMT-LIB.org (2006).

[24] Sutcliffe, G. and C. Suttner, *The TPTP problem library*, J. Autom. Reason. **21** (1998), pp. 177–203.

[25] Ševčík, J., *Proving resource consumption of low-level programs using automated theorem provers* (2007), full version of this paper. URL http://homepages.inf.ed.ac.uk/s0566973/papers/fospacefull.pdf

[26] Walker, D. and J. G. Morrisett, *Alias types for recursive data structures*, in: *TIC ’00: Selected papers from the Third International Workshop on Types in Compilation* (2001), pp. 177–206.

[27] Weidenbach, C., U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt and D. Topič, *SPASS version 2.0*, in: A. Voronkov, editor, *Automated deduction, CADE-18 : 18th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence **2392** (2002), pp. 275–279.

[28] Yorsh, G., A. M. Rabinovich, M. Sagiv, A. Meyer and A. Bouajjani, *A logic of reachable patterns in linked data-structures.*, in: L. Aceto and A. Ingólfsdóttir, editors, *FoSSaCS*, Lecture Notes in Computer Science **3921** (2006), pp. 94–110.