A Program Logic for Resource Verification

David Aspinall[1], Lennart Beringer[1], Martin Hofmann[2], Hans-Wolfgang Loidl[2], Alberto Momigliano[1]

[1] Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland; {da,lenb,amomigl1}@inf.ed.ac.uk
[2] Institut für Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany; {mhofmann,hwloidl}@informatik.uni-muenchen.de

Abstract. We present a program logic for reasoning about resource consumption of programs written in Grail, an abstract fragment of the Java Virtual Machine Language. Serving as the target logic of a certifying compiler, the logic exploits Grail's dual nature of combining a functional interpretation with object-oriented features and a cost model for the JVM. We present the resource-aware operational semantics of Grail, the program logic, and prove soundness and completeness. All of the work described has been formalised in the theorem prover Isabelle/HOL, which provides us with an implementation of the logic as well as confidence in the results. We conclude with examples of using the logic for proving resource bounds on code resulting from compiling high-level functional programs.

1 Introduction

For the effective use of mobile code, resource consumption is of great concern. A user who downloads an application program onto his mobile phone wants to know that the memory requirement of executing the program does not exceed the memory space available on the phone. Likewise, concerns occur in Grid computing where service providers want to know that user programs adhere to negotiated resource policies and users want to be sure that their program will not be terminated abruptly by the scheduler due to violations of some resource constraints.

The Mobile Resource Guarantees (MRG) project [27] is developing Proof-Carrying Code (PCC) technology [23] to endow mobile code with *certificates* of bounded resource consumption. Certificates in the PCC sense contain proof-theoretic evidence. A service provider can check a certificate to see that a given resource policy will be adhered to before admitting the code to run. The feasibility of the PCC approach relies on the observation that, while it may be difficult to produce a formal proof of a certain program property, it should be easy to check such a proof. Furthermore, resource properties are in many cases easier to verify than general correctness properties.

Following the PCC paradigm the code producer uses a combination of program annotations and analysis to construct a machine proof that a resource policy is met. The proof is expressed in a specialized program logic for the language in which the code is transmitted. In the MRG project, this target language is Grail [4], an abstract representation of (a subset of) the Java Virtual Machine Language (JVML). Certificate generation is performed by a certifying compiler, e.g. [7], which transforms programs written in MRG's high-level functional language Camelot into Grail [17]. Certificates are based on Camelot-level type systems for reasoning about resource consumption of functional programs [3, 11, 12]. For example, the Camelot program

```
let rev l acc = match l with Nil@d -> acc
                            | Cons(h,t)@d -> rev t (Cons(h,acc)@d)
```

for reversing a list does not consume heap space. In the match statement, the annotation `@` names the heap cell inhabited by the value, so that it can be reused when constructing new list nodes in the body. Restrictions on the usage of such annotations are subject of the type system [3, 11] and we have an automatic inference of such annotations for Camelot [12]. Indeed, we will prove later that the Grail code emitted for `rev` by our compiler does not allocate memory.

Contributions: We introduce a resource-aware program logic for Grail in which the certificates are expressed (Sections 2 and 3). The presentation of the logic follows the approach of the Vienna Development Method (VDM), a variation of Hoare-style program logic where assertions may refer to initial as well as to final states [14]. In our case, pre- and post-conditions are combined into single assertions ranging over pre-and post-heap, the environment in which the Grail expression is evaluated, the result value, and a component for the consumption of temporal and spatial resources. We discuss the meta-theoretic properties of soundness and (relative) completeness of the logic with respect to the functional operational semantics of Grail, based on a full formalisation in the theorem prover Isabelle/HOL. Since the program logic and its implementation are part of the trusted code base of the PCC infrastructure, it is essential for the overall security of the system to have such results available. Our formalisation builds upon previous work on embedding program logics in theorem provers, in particular that of Kleymann [15] and Nipkow [24] (see Section 5 for details). In contrast to that, our logic features a semantics that combines object-oriented aspects with a functional-style big-step evaluation relation, and includes a treatment of resource consumption that is related to a cost model for the execution of Grail on a virtual machine platform. The logic is tailored so that it can be proven sound and complete while at the same time it can be refined to be used for PCC-oriented program verification. This has influenced the departure from the more traditional Hoare format, where the need of auxiliary variables to propagate intermediate results from pre- to post-assertions is a serious issue w.r.t. automation. As a main technical result, we give a novel treatment of rules for mutually recursive procedures and adaptation that do not need separate judgements or a very complex variation of the consequence rule, but are elegantly proven admissible. Our focus on using Grail as an intermediate language, namely as the target of Camelot compilation, also motivates the decision not to provide a full treatment of object-oriented features such as inheritance and overriding. The expressiveness of our logic is demonstrated by verifying in Isabelle/HOL some resource properties of heap-manipulating Grail programs that were obtained by compiling Camelot programs (Section 4).

2 Grail

The Grail language [4] was designed as a compromise between raw bytecode and low-level functional languages, and serves as the target of the Camelot compilation. While the object and method structure of bytecode is retained, each method body consists of a set of mutually tail-recursive first-order functions. The syntax comprises instructions for object creation and manipulation, method invocation and primitive operations such as integer arithmetic, as well as let-bindings to combine program fragments. In the context of the Camelot compiler, static methods are of particular interest. Using a whole-program compilation approach, all datatypes are implemented by a single Grail class, the so-called "diamond" class [11], and functions over these datatypes result in distinct static methods operating on objects of this class [17]. The main characteristic of Grail is its dual identity: its (impure) call-by-value functional semantics is shown to coincide with an imperative interpretation of the expansion of Grail programs into the Java Virtual Machine Language, provided that some mild syntactic conditions are met. In particular, these require that actual arguments in function calls coincide syntactically with the formal parameters of the function definitions. This allows function calls to be interpreted as immediate jump instructions since register shuffling at basic block boundaries is performed by the calling code rather than being built into the function application rule. Consequently, the consumption of resources at virtual machine level may be expressed in a functional semantics for Grail: the expansion into JVML does not require register allocation or the insertion of gluing code.

We give an operational semantics and a program logic for a functional interpretation of Grail, where it is assumed (though not explicitly enforced) that expressions are in Administrative-Normal-Form, that is all intermediate values are explicitly named.

Syntax The syntax of Grail expressions makes use of mutually disjoint sets of integers, $\mathcal{M}$ of method names, $\mathcal{C}$ of class names, $\mathcal{F}$ of function names (i.e. labels of basic blocks), $\mathcal{T}$ of (virtual or static) field names and $\mathcal{X}$ of variables, ranged over by i, m, c, f, t, and x, respectively. We also introduce *self* as a reserved variable. In the following grammar, *op* denotes a primitive operation of type $\mathcal{V} \Rightarrow \mathcal{V} \Rightarrow \mathcal{V}$ such as an arithmetic operation or a comparison operator. Here $\mathcal{V}$ is the semantic category of values (ranged over by v), comprising integers, references r, and the special symbol $\perp$, which stands for the absence of a value. Boolean values are represented as integers. Heap references are either null or of the form $\mathsf{Ref}\, l$ where l is a location (represented by a natural number). Formal parameters of method invocations may be integer or object variables. Actual arguments are sequences of variable names or immediate values – complex expressions which may occur as arguments in Camelot functions are eliminated during the compilation process.

$$a \in args ::= \texttt{var}\ x \mid \mathsf{null} \mid i$$

$$\begin{aligned} e \in expr ::= {} & \texttt{null} \mid \texttt{int}\ i \mid \texttt{var}\ x \mid \texttt{prim}\ op\ x\ x \mid \texttt{new}\ c\ \boxed{t_i := x_i} \mid x.t \mid x.t{:=}x \mid c \diamond t{:=}x \mid \\ & c \diamond t \mid \texttt{let}\ x{=}e\ \texttt{in}\ e \mid e;e \mid \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{call}\ f \mid x \cdot m(\overline{a}) \mid c \diamond m(\overline{a}) \end{aligned}$$

Expressions represent basic blocks and are built from operators, constants, and previously computed values (names). Expressions such as $x.t{:=}y$ (putfield) correspond to primitive sequences of bytecode instructions that may, as a side effect, alter the heap or frame stack. Similarly, $c \diamond t$ and $c \diamond t{:=}y$ denote static field lookup and assignment, which are needed in Camelot's memory management. The binding `let` $x = e_1$ `in` e_2 is used if the evaluation of e_1 returns an integer or reference value on top of the JVM stack while $e_1;e_2$ represents purely sequential composition, used for example if e_1 is a field update $x.t{:=}y$. Object creation includes the initialisation of the object fields according to the argument list: the content of variable x_i is stored in field t_i. Function calls (`call`) follow the Grail calling convention (i.e. correspond to immediate jumps) and do not carry arguments. The instructions $x \cdot m(\overline{a})$ and $c \diamond m(\overline{a})$ represent virtual (instance) and static method invocation. Although a formal type and class system may be imposed on Grail programs, our program logic abstracts from these restrictions; heap and class file environment are total functions on field and method names, respectively.

We assume that all method declarations employ distinct names for identifying inner basic blocks. A program is represented by a table *FT* mapping each function identifier to a list of (distinct) variables (the formal parameters) and an expression, and a table *MT* associating the formal method parameters (again a list of distinct variables) and the initial basic block to class names and method identifiers.

Dynamic Semantics The machine model is based on semantic domains $\mathcal{H}$ of heaps, $\mathcal{E}$ of environments (maps from variables to values) and $\mathcal{R}$ of resource components. A heap h maps locations to objects, where an object comprises a class name and a mapping of field names to values. In our formalisation, we follow an approach inspired by Burstall, where the heap is split into several components: a total function from field names to locations to values, and a partial function from locations to class names. In addition, we also introduce a total map for modelling static (reference) fields, mapping class names and field names to references.

Variables which are local to a method invocation are kept in an environment E that corresponds to the local store of the JVM. Environments are represented as total functions, with the silent assumption that well-defined method bodies only access variables which have previously been assigned a value. We use $E\langle x\rangle$ to denote the lookup operation and $E\langle x := v\rangle$ to denote an update. Since the operational semantics uses environments to represent the local store of method frames, no explicit frame stack is needed. The height of the stack is mentioned as part of the resource component.

Resource consumption is modelled by resource tuples $p \in \mathcal{R}$, where

$$p = \langle clock \quad callc \quad invkc \quad invkdpth\rangle.$$

The four components range over $\mathbb{N}$ and represent the following costs. The *clock* represents a global abstract instruction counter. The *callc* and *invkc* components are more refined, i.e. they count the number of function calls (jump instructions) and method invocations. We can easily count other types of instructions, but we chose these initially as interesting cases: for example they may be used to formally verify Grail-level optimisations such as the replacement of method (tail) recursion by function recursion. Finally, *invkdpth* models the maximal invocation depth, i.e. the maximal height of the frame stack throughout an execution. From this, the maximal frame stack height may be approximated by considering the maximal size of single frames. The size of the heap is not monitored explicitly in the resource components, since it can be deduced from the representation of the object heap as $|dom(h)|$.

The operational semantics and the program logic make use of two operators on resources, $p \oplus q$ and $p \smile q$. In the first three components, both operators perform pointwise addition, as all instruction counts behave additionally during program composition. In the fourth component, the operator $\oplus$ again adds the respective components of p and q, while $\smile$ takes their maximum. By employing $\smile$ in the rules for let-bindings we can thus model the release of frame stacks after the execution of method invocations.

The semantics is a big-step evaluation relation based on the functional interpretation of Grail, with judgements of the form

$$E \vdash h, e \Downarrow (h', v, p).$$

Such a statement reads "in variable environment E and initial heap h, code e evaluates to the value v, yielding the heap h' and consuming p resources."

$$\frac{}{E \vdash h, \texttt{null} \Downarrow (h, \mathsf{null}, \langle 1\ 0\ 0\ 0 \rangle)}\ (\text{NULL}) \qquad \frac{}{E \vdash h, \texttt{int}\ i \Downarrow (h, i, \langle 1\ 0\ 0\ 0 \rangle)}\ (\text{INT})$$

$$\frac{}{E \vdash h, \texttt{var}\ x \Downarrow (h, E\langle x \rangle, \langle 1\ 0\ 0\ 0 \rangle)}\ (\text{VAR})$$

$$\frac{}{E \vdash h, \texttt{prim}\ op\ x\ y \Downarrow (h, op\ (E\langle x \rangle)\ (E\langle y \rangle), \langle 3\ 0\ 0\ 0 \rangle)}\ (\text{PRIM})$$

$$\frac{E\langle x \rangle = \mathsf{Ref}\ l}{E \vdash h, x.t \Downarrow (h, h(l).t, \langle 2\ 0\ 0\ 0 \rangle)}\ (\text{GETF}) \qquad \frac{E\langle x \rangle = \mathsf{Ref}\ l}{E \vdash h, x.t{:=}y \Downarrow (h[l.t \mapsto E\langle y \rangle], \bot, \langle 3\ 0\ 0\ 0 \rangle)}\ (\text{PUTF})$$

$$\frac{}{E \vdash h, c \diamond t \Downarrow (h, h(c).t, \langle 2\ 0\ 0\ 0 \rangle)}\ (\text{GFST}) \qquad \frac{}{E \vdash h, c \diamond t{:=}y \Downarrow (h[c.t \mapsto E\langle y \rangle], \bot, \langle 3\ 0\ 0\ 0 \rangle)}\ (\text{PFST})$$

$$\frac{l = \mathit{freshloc}(h)}{E \vdash h, \texttt{new}\ c\ [\overline{t_i := x_i}] \Downarrow (h[l \mapsto (c, \{\overline{t_i := E\langle x_i \rangle}\})], \mathsf{Ref}\ l, \langle (n+1)\ 0\ 0\ 0 \rangle)}\ (\text{NEW})$$

$$\frac{E\langle x \rangle = \mathsf{true} \quad E \vdash h, e_1 \Downarrow (h_1, v, p)}{E \vdash h, \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow (h_1, v, \langle 2\ 0\ 0\ 0 \rangle \oplus p)}\ (\text{IFTRUE})$$

$$\frac{E\langle x \rangle = \mathsf{false} \quad E \vdash h, e_2 \Downarrow (h_1, v, p)}{E \vdash h, \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow (h_1, v, \langle 2\ 0\ 0\ 0 \rangle \oplus p)}\ (\text{IFFALSE})$$

$$\frac{E \vdash h, e_1 \Downarrow (h_1, w, p) \quad w \neq \bot \quad E\langle x := w \rangle \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, \texttt{let}\ x{=}e_1\ \texttt{in}\ e_2 \Downarrow (h_2, v, \langle 1\ 0\ 0\ 0 \rangle \oplus (p \smile q))}\ (\text{LET})$$

$$\frac{E \vdash h, e_1 \Downarrow (h_1, \bot, p) \quad E \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, e_1 ; e_2 \Downarrow (h_2, v, p \smile q)}\ (\text{COMP})$$

$$\frac{E \vdash h, \mathit{snd}(FT\ f) \Downarrow (h_1, v, p)}{E \vdash h, \texttt{call}\ f \Downarrow (h_1, v, \langle 1\ 1\ 0\ 0 \rangle \oplus p)}\ (\text{CALL})$$

$$\frac{(\mathit{newframe}\ \mathsf{null}\ \mathit{fst}(MT\ c\ m)\ \overline{a}\ E) \vdash h, \mathit{snd}(MT\ c\ m) \Downarrow (h_1, v, p)}{E \vdash h, c \diamond m(\overline{a}) \Downarrow (h_1, v, \langle (2 + |\ \overline{a}\ |)\ 0\ 1\ 1 \rangle \oplus p)}\ (\text{SINV})$$

$$\frac{\mathit{classOf}\ E\ h\ x\ c \quad (\mathit{newframe}\, E\langle x\rangle\, \mathit{fst}(MT\ c\ m)\ \overline{a}\ E) \vdash h, \mathit{snd}(MT\ c\ m) \Downarrow (h_1, v, p)}{E \vdash h, x \cdot m(\overline{a}) \Downarrow (h_1, v, \langle (4+ \mid \overline{a} \mid)\ 0\ 1\ 1\rangle \oplus p)} \quad (\text{VINV})$$

In rule GETF, the notation $h(l).t$ represents the value of field t in the object at heap location l, while in rule PUTF the notation $h[l.t \mapsto v]$ denotes the corresponding update operation. Similarly for static fields. In rule NEW, the function *freshloc*(h) returns a fresh location outside the domain of h, and $h[l \mapsto (c, \{\overline{t_i := E\langle x_i\rangle}\})]$ represents the heap that agrees with h on all locations different from l and maps l to an object of class c, with field entries $t_i := E\langle x_i\rangle$. In the rules CALL, SINV and VINV the lookup functions *FT* and *MT* are used to obtain function and method bodies from names. These are here implemented as static tables, though they could be used to model a class hierarchy. In particular *MT* has type $\mathcal{C} \Rightarrow \mathcal{M} \Rightarrow (\mathcal{X}\ \mathit{list} \times \mathit{expr})$, where the parameter passing in method invocations is modelled by accessing the parameter values from the caller's environment. Each method invocation allocates a new frame on the frame stack, where the function *newframe* creates the appropriate environment, given a reference to the invoking object, the formal parameters and the actual arguments. The environment contains bindings for the self object and the method parameters. If we invoke a static method we set the self variable to null, otherwise to the current object.

The resource tuples in the operational semantics abstractly characterise resource consumption in an unspecified virtual machine; because resources are treated separately, these values could be changed for particular virtual machines. The temporal costs associated to basic instructions reflect the number of bytecode instructions to which the expression expands. For example, the PUTF operation involves two instructions for pushing the object pointer $E\langle x\rangle$ and the new content $E\langle y\rangle$ onto the operand stack, plus one additional instruction for performing the actual field modification. In rule NEW we charge a single clock tick for object creation, and n for field initialisation. The costs for primitive operations may be generalised to a table lookup. In the rules for conditionals, we charge for pushing the value $E\langle x\rangle$ onto the stack, with an additional clock tick for evaluating the branch condition and performing the appropriate jump. In rule CALL, the Grail functional call convention explains why we treat the call as a jump, continuing with the execution of function body. We charge for one anonymous instruction, and also explicitly for the execution of a jump. In rule SINV, the body of method is executed in an environment which represents a fresh frame. The instruction counter is incremented by 2 for pushing and popping the frame and $\mid \overline{a} \mid$ for evaluating the arguments. In addition, both the invocation counter and the invocation depth are incremented by one — the usage of $\oplus$ ensures that the depth correctly represents the nesting depth of frames. Finally, in rule VINV, the predicate *classOf E h x c* first retrieves the dynamic class name c associated to the object pointed to by x. Then, the method body associated to m and c is executed in a fresh environment which contains the reference to $E\langle x\rangle$ in variable *self* and the formal parameters as above. The costs charged arise again by considering the evaluation of $E\langle x\rangle$ and the method arguments, and the pushing and popping of the frame, but we also charge one clock tick for the indirection needed to retrieve the correct method body from the class file.

3 Program logic

The program logic targets the partial correctness of resource bounds such as heap allocation and combines aspects of VDM-style verification [14] and Abadi-Leino's logic for object calculi [1]. Sequents are of the form $\Gamma \rhd e : P$ and relate a Grail expression $e \in expr$ to a specification $P \in \mathcal{A}$ in a context $\Gamma \in \mathcal{G}$ (see definition below). We abbreviate $\emptyset \rhd e : P$ to $\rhd e : P$. We follow the *extensional* approach to the representation of assertions [15], where specifications are predicates (in the meta-logic) over semantic components and can refer to the initial and final heaps of a program expression, the initial environment, the resources consumed and the result value: $\mathcal{A} \equiv \mathcal{E} \to \mathcal{H} \to \mathcal{H} \to \mathcal{V} \to \mathcal{R} \to \mathcal{B}$, where $\mathcal{B}$ is the set of booleans. Satisfaction of a specification P by program e is denoted by $\models e : P$. We interpret a judgement $\models e : \lambda\, E\; h\; h'\; v\; p.\; P\; E\; h\; h'\; v\; p$ to mean that whenever the execution of e for initial heap h and environment E terminates and delivers final heap h', result v and resources p, P is satisfied, that is that $E \vdash h, e \Downarrow (h', v, p)$ implies $P\, E\, h\, h'\, v\, p$.

Similar to assertions in VDM logics, our specifications relate pre- and post-states without auxiliary variables. For example, programs that do not allocate heap space satisfy the assertion $|dom(h)| = |dom(h')|$.

Rules: In the program logic, contexts Γ manage assumptions when dealing with (mutually) recursive or externally defined methods. They consist of pairs of expressions and specifications: $\mathcal{G} \equiv expr \times \mathcal{A}$. In addition to rules for each form of program expression there are two logical rules, VAX and VCONSEQ.

$$\frac{(e,P) \in \Gamma}{\Gamma \rhd e : P}\ (\text{VAX}) \qquad \frac{\Gamma \rhd e : P \quad \forall E\, h\, h'\, v\, p.\; P\, E\, h\, h'\, v\, p \longrightarrow Q\, E\, h\, h'\, v\, p}{\Gamma \rhd e : Q}\ (\text{VCONSEQ})$$

$$\frac{}{\Gamma \rhd \texttt{null} : \lambda E\, h\, h'\, v\, p.\, h' = h \wedge v = \mathsf{null} \wedge p = \langle 1\ 0\ 0\ 0 \rangle}\ (\text{VNULL})$$

$$\frac{}{\Gamma \rhd \texttt{int}\ i : \lambda E\, h\, h'\, v\, p.\, h' = h \wedge v = i \wedge p = \langle 1\ 0\ 0\ 0 \rangle}\ (\text{VINT})$$

$$\frac{}{\Gamma \rhd \texttt{var}\ x : \lambda E\, h\, h'\, v\, p.\, h' = h \wedge v = E\langle x \rangle \wedge p = \langle 1\ 0\ 0\ 0 \rangle}\ (\text{VVAR})$$

$$\frac{}{\Gamma \rhd \texttt{prim}\ op\ x\ y : \lambda E\, h\, h'\, v\, p.\; v = op\ E\langle x \rangle\ E\langle y \rangle\ \wedge h' = h \wedge p = \langle 3\ 0\ 0\ 0 \rangle}\ (\text{VPRIM})$$

$$\frac{}{\Gamma \rhd x.t : \lambda E\, h\, h'\, v\, p.\, \exists l.\, E\langle x \rangle = \mathsf{Ref}\, l \wedge h' = h\ \wedge v = h'(l).t\ \wedge p = \langle 2\ 0\ 0\ 0 \rangle}\ (\text{VGETF})$$

$$\frac{}{\begin{array}{r}\Gamma \rhd x.t{:=}y : \lambda E\, h\, h'\, v\, p.\, \exists l.\, E\langle x \rangle = \mathsf{Ref}\, l \wedge p = \langle 3\ 0\ 0\ 0 \rangle\ \wedge \\ h' = h[l.t \mapsto E\langle y \rangle]\ \wedge v = \bot\end{array}}\ (\text{VPUTF})$$

$$\frac{}{\Gamma \rhd c \diamond t : \lambda E\, h\, h'\, v\, p.\, h' = h\ \wedge v = h(c).t\ \wedge p = \langle 2\ 0\ 0\ 0 \rangle}\ (\text{VGETST})$$

$$\frac{}{\Gamma \rhd c \diamond t{:=}y : \lambda E\, h\, h'\, v\, p.\, h' = h[c.t \mapsto E\langle y \rangle]\ \wedge v = \bot \wedge p = \langle 3\ 0\ 0\ 0 \rangle}\ (\text{VPUTST})$$

$$\frac{}{\Gamma \rhd \texttt{new}\ c\ [\overline{t_i := x_i}] : \lambda E\, h\, h'\, v\, p.\, \exists l.\, l = \mathit{freshloc}(h) \wedge p = \langle (n+1)\ 0\ 0\ 0 \rangle \wedge h' = h[l \mapsto (c, \{\overline{t_i := E\langle x_i \rangle}\})] \ \wedge\ v = \mathsf{Ref}\, l} \quad \text{(VNEW)}$$

$$\frac{\Gamma \rhd e_1 : P_1 \quad \Gamma \rhd e_2 : P_2}{\begin{array}{l}\Gamma \rhd \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \lambda E\, h\, h'\, v\, p.\, \exists p'.\, p = p' \oplus \langle 2\ 0\ 0\ 0 \rangle \wedge \\ \qquad (E\langle x \rangle = \mathsf{true} \longrightarrow P_1\, E\, h\, h'\, v\, p') \wedge \\ \qquad (E\langle x \rangle = \mathsf{false} \longrightarrow P_2\, E\, h\, h'\, v\, p') \wedge \\ \qquad (E\langle x \rangle = \mathsf{true} \vee E\langle x \rangle = \mathsf{false})\end{array}} \quad \text{(VIF)}$$

$$\frac{\Gamma \rhd e_1 : P_1 \quad \Gamma \rhd e_2 : P_2}{\begin{array}{l}\Gamma \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \lambda E\, h\, h'\, v\, p.\, \exists\ p_1\ p_2\ h_1\ w.\, (P_1\, E\, h\, h_1\, w\, p_1) \ \wedge\ w \neq \bot \ \wedge \\ \qquad (P_2\, (E\langle x := w \rangle)\, h_1\, h'\, v\, p_2) \ \wedge \\ \qquad p = \langle 1\ 0\ 0\ 0 \rangle \oplus (p_1 \smile p_2)\end{array}} \quad \text{(VLET)}$$

$$\frac{\Gamma \rhd e_1 : P_1 \quad \Gamma \rhd e_2 : P_2}{\begin{array}{l}\Gamma \rhd e_1 ; e_2 : \lambda E\, h\, h'\, v\, p.\, \exists\ p_1\, p_2\, h_1.\, P_1\, E\, h\, h_1 \perp p_1 \ \wedge \\ \qquad P_2\, E\, h_1\, h'\, v\, p_2 \ \wedge\ p = p_1 \smile p_2\end{array}} \quad \text{(VCOMP)}$$

$$\frac{\Gamma \cup \{(\texttt{call}\, f, P)\} \rhd \mathit{snd}(FT\, f) : \lambda E\, h\, h'\, v\, p.\, P\, E\, h\, h'\, v\, \langle 1\ 1\ 0\ 0 \rangle \oplus p}{\Gamma \rhd \texttt{call}\, f : P} \quad \text{(VCALL)}$$

$$\frac{\begin{array}{l}\Gamma \cup \{(c \diamond m(\overline{a}), P)\} \rhd \mathit{snd}(MT\ c\ m) : \lambda E\, h\, h'\, v\, p.\, \forall E'.\ E = (\mathit{newframe}\, \mathsf{null}\, \mathit{fst}(MT\ c\ m)\ \overline{a}\ E') \\ \qquad \longrightarrow P\, E'\, h\, h'\, v\, \langle (2 + |\, \overline{a}\, |)\ 0\ 1\ 1 \rangle \oplus p\end{array}}{\Gamma \rhd c \diamond m(\overline{a}) : P} \quad \text{(VSINV)}$$

$$\frac{\begin{array}{l}\Gamma \cup \{x \cdot m(\overline{a}), P)\} \rhd \\ \mathit{snd}(MT\ c\ m) : \lambda\ E\ h\ h'\ v\ p.\ \forall\ E'.\, (\mathit{classOf}\ E\ h\ x\ c\ \wedge \\ \qquad E = (\mathit{newframe}\, (E'\langle x \rangle)\, \mathit{fst}(MT\ c\ m)\ \overline{a}\ E')) \\ \qquad \longrightarrow\ (E', h, h', v, \langle (4 + |\, \overline{a}\, |)\ 0\ 1\ 1 \rangle \oplus p) \in P\end{array}}{\Gamma \rhd x \cdot m(\overline{a}) : P} \quad \text{(VVINV)}$$

The axiom rule VAX allows one to use specifications found in the context. The VCONSEQ consequence rule derives an assertion Q that follows from another assertion P. The leaf rules (VNULL to VNEW) directly model the corresponding rules in the operational semantics, with constants for the resource tuples. The VIF rule uses the appropriate assertion based on the boolean value in the variable x. Since the evaluation of the branch condition does not modify the heap we only existentially quantify over the resource tuple p'. In contrast, rule VLET existentially quantifies over the result value w, the heap h_1 resulting from evaluating e_1, and the resources from e_1 and e_2. Apart from the absence of environment update, rule VCOMP is similar to VLET. By relating pre and post conditions in a single assertion we avoid the complications associated to the usual VDM rules for sequencing [14]. However, this makes reasoning about *total* correctness more difficult. The rules for recursive functions and methods involve the context and generalize Hoare's original rule for parameterless recursive procedures. They require one to prove that the function or method body satisfies the required specification (with an updated resource component) under the additional assumption that the assertion holds for further calls or invocations.

Admissible Rules: A context weakening rule is easily seen to be admissible. We can also prove the following cut rule by induction on derivations of $\{(e',P)\} \cup \Delta \rhd e : Q$,

$$\frac{\{(e',P)\} \cup \Delta \rhd e : Q \qquad \Gamma \rhd e' : P \qquad \Gamma \subseteq \Delta}{\Delta \rhd e : Q} \qquad \text{(CUT)}$$

One of the contributions of this paper lies in an innovative approach to mutually recursive procedures and adaptation. In fact, rules VCALL, VSINV and VVINV already cover the case of mutual recursion. So we do not need a separate derivation system for judgements with *sets* of assertions and related set introduction and elimination rules, as for example Nipkow does [24], nor do we need to modify the consequence rule to take care of adaptation. The treatment is based on specification tables for functions and methods. A function specification table *FST* maps each function identifier to an assertion, a virtual method specification table *vMST* maps triples consisting of variable names, method names and (actual) argument lists to assertions, and a static method specification table *sMST* maps triples consisting of class names, method names and (actual) argument lists to assertions. Since the types allow us to disambiguate between the three tables, we use the notation *ST* to refer to their union.

A context Γ is *good* with respect to the specification tables, notation $good_{ST}(\Gamma)$, if all entries $(e,P) \in \Gamma$ satisfy

$$\begin{array}{l}(\exists f.\, e = \texttt{call}\, f \,\wedge\, P = FST\, f \,\wedge\, \Gamma \rhd snd(FT\ f) : Q_0(f)) \,\vee \\ (\,\exists\, c\, m\, \overline{a}.\, e = c \diamond m(\overline{a}) \,\wedge\, P = ST\ c\ m\ \overline{a} \,\wedge\, \forall \overline{b}.\, \Gamma \rhd snd(MT\ c\ m) : Q_1(c,m,\overline{b})\,) \,\vee \\ (\,\exists\, x\, m\, \overline{a}.\, e = x \cdot m(\overline{a}) \,\wedge\, P = ST\ x\ m\ \overline{a} \,\wedge\, \forall\, y\, \overline{b}\, c.\, \Gamma \rhd snd(MT\ c\ m) : Q_2(c,m,\overline{b},y)\,)\end{array}$$

where

$$\begin{array}{rl} Q_0(f) \equiv & \lambda E\, h\, h'\, v\, p.\, (FST\, f)\ E\ h\ h'\ v\ (\langle 1\ 1\ 0\ 0 \rangle \oplus p) \\ Q_1(c,m,\overline{b}) \equiv & \lambda E\, h\, h'\, v\, p\,.\, \forall E'.\ E = (newframe\, \mathsf{null}\, fst(MT\ c\ m)\ \overline{b}\ E') \\ & \longrightarrow ST\ c\ m\ \overline{b} E'\ h\ h'\ v\ (\langle (2+ \mid \overline{b} \mid)\ 0\ 1\ 1 \rangle \oplus p) \\ Q_2(c,m,\overline{b},y) \equiv & \lambda\ E\ h\ h'\ v\ p.\forall\ E'.\ (classOf\ E'\ h\ y\ c \,\wedge \\ & E = (newframe\, (E' \langle y \rangle)\, fst(MT\ c\ m)\ \overline{b}\ E')) \\ & \longrightarrow ST\ y\ m\ \overline{b}\ E'\ h\ h'\ v\ (\langle (4+ \mid \overline{b} \mid)\ 0\ 1\ 1 \rangle \oplus p). \end{array}$$

Using the cut rule, we can prove that *good* contexts are subset-closed.

Lemma 1. $good_{ST}(\Gamma) \longrightarrow good_{ST}(\Gamma - \{(e,P)\})$.

By combining this lemma with another application of CUT, one can prove by induction on the size of Γ the following rule for mutually recursive function calls or method invocations, for the *empty* context,

$$\frac{\Gamma\ \mathit{finite} \quad good_{ST}(\Gamma) \quad (e,P) \in \Gamma}{\rhd e : P} \qquad \text{(MUTREC)}$$

A variant of Lemma 1 also plays an important part in the proof of our adaptation rule. Parameter adaptation is notoriously problematic and has often been coupled with rules of consequence, resulting in fairly complicated rules [15, 24, 25]. Instead, building on the notion of *good*, we can prove (via cut and weakening) the following lemma, which allows one to change the actual parameters from $\overline{b}$ to $\overline{a}$.

Lemma 2. $(good_{ST}(\Gamma) \wedge (c \diamond m(\overline{b}), ST\ c\ m\ \overline{b}) \in \Gamma) \longrightarrow$
$\Gamma - \{(c \diamond m(\overline{b}), ST\ c\ m\ \overline{b})\} \rhd c \diamond m(\overline{a}) : ST\ c\ m\ \overline{a}$

The predicate *good* ensures, that for every pair method invocation/specification over given actual arguments, the context proves that the method body satisfies the same specification over any other arguments, provided the former is updated to reflect the new environment with the appropriate binding for the formal parameters. Since we want to prove specifications in the empty context, the lemma allows one to shrink the context.

From that, adaptation in the empty context follows:

$$\frac{\Gamma\ \mathit{finite} \quad good_{ST}(\Gamma) \quad (c \diamond m(\overline{b}), ST\ c\ m\ \overline{b}) \in \Gamma}{\rhd c \diamond m(\overline{a}) : ST\ c\ m\ \overline{a}} \qquad (\text{ADAPTS})$$

We shall see this rule in action in Section 4. Both, Lemma 2 and rule ADAPTS, have counterparts for virtual methods.

Soundness We first define the *validity* of an assertion for a given program expression in a given context. In order to deal with soundness of function calls and method invocations we additionally parameterise the operational semantics by a natural number acting as the height of the evaluation [10, 15, 24].

Definition 1. *(Validity) Specification P is* valid *for e, written* $\models_n e : P$, *if*

$$(m \le n \ \wedge E \vdash h, e \Downarrow_m (h', v, p)) \longrightarrow P\ E\ h\ h'\ v\ p.$$

We define $\models e : P$ as $\forall n. \models_n e : P$. Note that the counter n restricts the set of pre- and post-states for which P has to be fulfilled. It is easy to show that this bound, occurring negatively in the validity formula, can be weakened, i.e. $(m < n \wedge \models_n e : P) \longrightarrow \models_m e : P$. Validity is generalised to contexts as follows:

Definition 2. *(Context Validity) Context* Γ *is* valid, *written* $\vDash_n \Gamma$, *if* $\models_n e : P$ *holds for all* $(e, P) \in \Gamma$. *Assertion P is* valid for e in context Γ, *denoted* $\Gamma \models_n e : P$, *if* $\vDash_n \Gamma$ *implies* $\models_n e : P$.

The soundness theorem follows from a stronger result expressing the soundness property for contextual, relativised validity.

Theorem 1. *(Soundness)* $\Gamma \rhd e : P \longrightarrow \forall n. \Gamma \models_n e : P$.

Completeness The program logic may be proven complete relative to the ambient logic (here HOL) using the notion of *strongest specifications*, similar to *most general triples* in Hoare-style verification.

Definition 3. *(Strongest Specification) The* strongest specification *of expression e is*

$$SSpec(e) = \lambda\ E\ h\ h'\ v\ p.\ E \vdash h, e \Downarrow (h', v, p).$$

It is not difficult to prove that strongest specifications are valid, i.e. $\models e : SSpec(e)$, and further that they are stronger than any other valid specification, that is $(\models e : P \wedge\ SSpec(e)\ E\ h\ h'\ v\ p) \longrightarrow P\ E\ h\ h'\ v\ p$.

The overall proof idea of completeness follows [10, 24]: we first prove a lemma that allows one to relate *any* expression e to $SSpec(e)$ in a context Γ, provided that Γ in turn relates each function or method call to its strongest specification.

Lemma 3. $\begin{pmatrix} \forall\, f.\ \Gamma \rhd \texttt{call}\, f : SSpec(\texttt{call}\, f) \wedge \\ \forall\, c\ m\ \overline{a}.\ \Gamma \rhd c \diamond m(\overline{a}) : SSpec(c \diamond m(\overline{a})) \wedge \\ \forall\, x\ m\ \overline{a}.\ \Gamma \rhd x \cdot m(\overline{a}) : SSpec(x \cdot m(\overline{a})) \end{pmatrix} \longrightarrow \Gamma \rhd e : SSpec(e)$

The proof of this lemma proceeds by induction on the structure of e. Next, we define a specific context, $\widehat{\Gamma}$, containing exactly the strongest specifications for all function calls and method invocations.

$$\widehat{\Gamma} \equiv \{(e,P) \mid P = SSpec(e)\ \wedge\ \begin{pmatrix} (\exists\, f.\ e = \texttt{call}\, f)\ \vee\ (\exists\, c\ m\ \overline{a}.\ e = c \diamond m(\overline{a}))\ \vee \\ (\exists\, x\ m\ \overline{a}.\ e = x \cdot m(\overline{a})) \end{pmatrix}\}.$$

We also define specification tables that associate the strongest assertions to all calls and invocations:

$$\widehat{ST} \equiv (\lambda f.\ SSpec(\texttt{call}\, f)) \cup (\lambda\ c\ m\ \overline{a}.\ SSpec(c \diamond m(\overline{a}))) \cup (\lambda\ x\ m\ \overline{a}.\ SSpec(x \cdot m(\overline{a}))).$$

Next, we show that $\widehat{\Gamma}$ is *good* with respect to these tables:

Lemma 4. $good_{\widehat{ST}}(\widehat{\Gamma})$.

On the other hand, combining a variant of rule CUT and MUTREC with Lemma 3 yields

Lemma 5. *If $good_{ST}(\widehat{\Gamma})$ and $\widehat{\Gamma}$ finite, then $\rhd e : SSpec(e)$ holds for all e,*

for arbitrary specification tables ST. Finally, combining Lemmas 4 and 5 and rule VCONSEQ yields

Theorem 2. *(Completeness) If $\widehat{\Gamma}$ finite and $\models e : P$ then $\rhd e : P$.*

The finiteness condition merely represents a constraint on the syntactic categories of function and method names. It is fulfilled for any concrete program.

4 Examples

In this section we give examples of proving resource properties of Grail programs working on integer lists. We first discuss how lists are modelled in our formalisation and then consider in-place list reversal and doubling elements in a list as example programs. The Grail code in this section corresponds to the Isabelle-output of the Camelot compiler.

During the compilation, heap-allocated data structures arise from algebraic datatypes in Camelot. Values of the type `ilist = Nil | Cons of int * ilist` are represented as a linked list of objects of the diamond class. Each node contains fields HD, TL and TAG, where TAG indicates the constructor (`Nil` or `Cons`) used to create the cell. Since our verification targets the consumption of resources rather than full correctness we use a representation predicate that ensures that a portion of the heap represents a list structure without considering the data contained in individual cells. The predicate takes the form $h,l \models_X n$, to be read as "starting at location l the sub-heap of h given by the set X of locations contains a list of length n". It is defined inductively, using the additional notation $class_h(l)$ to refer to the class of the object located at l in heap h.

$$
\begin{array}{l}
(class_h(l) = \mathsf{ILIST} \wedge h(l).\mathsf{TAG} = 0) \longrightarrow h,l \models_{\{l\}} 0 \\
\begin{pmatrix} class_h(l) = \mathsf{ILIST} \wedge h(l).\mathsf{TAG} = 1 \wedge \\ h(l).\mathsf{TL} = \mathsf{Ref}\, r \wedge l \notin X \wedge h,r \models_X n \end{pmatrix} \longrightarrow h,l \models_{X\cup\{l\}} n+1
\end{array}
$$

Similar predicates have been used by Reynolds in separation logic [26]. Notice that in the second case the reference l has to be distinct from all previously used locations X.

In-place reversal: Returning to our motivating example from the introduction, the following Grail code is produced for the method *rev* in class ILIST with formal parameters $[l, acc]$:

$$
\begin{array}{l}
\texttt{let}\ tag = l.\mathsf{TAG}\ \texttt{in}\quad \texttt{let}\ b = \texttt{prim}\ iszero\ tag\ tag\ \texttt{in} \\
\texttt{if}\ b\ \texttt{then}\ \texttt{var}\ acc \\
\quad\quad \texttt{else}\ \texttt{let}\ h = l.\mathsf{HD}\ \texttt{in}\ \ \texttt{let}\ t = l.\mathsf{TL}\ \texttt{in}\ \ \texttt{let}\ one = \texttt{int}\ 1\ \texttt{in} \\
\quad\quad\quad\quad l.\mathsf{TAG}{:=}one; l.\mathsf{HD}{:=}h; l.\mathsf{TL}{:=}acc; \mathsf{ILIST} \diamond rev([t,l])
\end{array}
$$

We constrain the specification tables to contain the entry

$$
\begin{array}{l}
ST\ \mathsf{ILIST}\ rev\ z\ E\ h\ h'\ v\ p = \\
\forall\, n\ a\ X\ m\ b\ Y.\ \begin{pmatrix} (eval\ E\ z = [\mathsf{Ref}\, a, \mathsf{Ref}\, b] \wedge h,a \models_X n \wedge h,b \models_Y m \wedge X \cap Y = \emptyset) \\ \longrightarrow |dom(h)| = |dom(h')| \wedge p = \langle (29n+13)\ 0\ (n+1)\ (n+1) \rangle \end{pmatrix}
\end{array}
$$

If the first method argument points initially to a list of length n, and the second argument points to some other (disjoint) list, any terminating execution of *rev* returns a heap of the same size as the initial heap, and the number of instructions and function calls (jump instructions) depend linearly on n. The function *eval* implements the evaluation of methods arguments and is part of the *newframe* construction. We aim to prove the property

$$\rhd\, \mathsf{ILIST} \diamond rev([x,y]) : ST\ \mathsf{ILIST}\ rev\ [x,y] \tag{1}$$

which states that an invocation of *rev* with (arbitrary) arguments x and y satisfies its specification. The generic structure of a proof of such a resource predicate first applies the rule ADAPTS. The required context Γ contains one entry for each method invocation that occurs in the method body, pairing each such call with its specification:

$$\Gamma \equiv \{(\mathsf{ILIST} \diamond rev([t,l]), ST\ \mathsf{ILIST}\ rev\ [t,l])\}.$$

As the main lemma we then prove that Γ is *good* with respect to the specification tables:

$$good_{ST}(\Gamma).$$

The proof of this statement proceeds by first applying the VDM rules VSINV and VCONSEQ, and then the other syntax-directed rules according to the program text, closing the recursion by an invocation of VAX. This first phase can be seen as a classical VCG over the program logic rules. Two side conditions remain, requiring us to show that both branches satisfy the specification — the verification condition of the recursion case amounts to a loop invariant. Both side conditions can be discharged by unfolding the definition of $h,l \models_X n$ and instantiating some quantifiers.

Where do the polynomials in the specification come from? Currently, we have left those values indeterminate and have them generated during a proof. In a later phase of the project, the Camelot compiler will generate certificates for such resource properties based on high-level program analysis similar to [12]'s type system for heap space consumption. The syntactic form of *rev* would allow a tail-call optimisation, where the recursive method invocation is transformed into a recursive function call satisfying the Grail calling convention.

Doubling a list: Consider the following code for doubling the elements of a list.

```
let double l = match l with
                 Nil@d -> Nil@d
               | Cons(h,t)@d -> Cons(h,Cons(h,double t)@d)
```

Remember the usage of `@` indicates that heap cells which are freed during a match may be reused later — but only once [11] — so the outer application of `Cons` will require the allocation of fresh memory. Since the recursion occurs in non-tail position, it cannot be replaced by a simple function recursion and the resulting Grail code contains a static method $\mathsf{ILIST} \diamond \mathit{double}(l)$ with body

$$\begin{array}{l}
\texttt{let}\ x = l.\mathsf{TAG}\ \texttt{in}\ \ \texttt{let}\ b = \texttt{prim}\ \mathit{iszero}\ x\ x\ \texttt{in} \\
\texttt{if}\ b\ \texttt{then}\ \texttt{let}\ \mathit{zero} = \texttt{int}\ 0\ \texttt{in}\ \ l.\mathsf{TAG}{:=}\mathit{zero};\texttt{var}\ l \\
\quad\ \texttt{else}\ \texttt{let}\ x = l.\mathsf{HD}\ \texttt{in}\ \ \texttt{let}\ t = l.\mathsf{TL}\ \texttt{in}\ \ \texttt{let}\ y = \texttt{var}\ l\ \texttt{in} \\
\qquad\quad \texttt{let}\ z = \mathsf{ILIST} \diamond \mathit{double}([t])\ \texttt{in}\ \ \texttt{let}\ \mathit{one} = \texttt{int}\ 1\ \texttt{in} \\
\qquad\quad y.\mathsf{TAG}{:=}\mathit{one};y.\mathsf{HD}{:=}x;y.\mathsf{TL}{:=}z;\texttt{let}\ l = \texttt{var}\ y\ \texttt{in} \\
\qquad\quad \texttt{new}\ \mathsf{ILIST}\ [(\mathsf{TAG},\mathit{one}),(\mathsf{HD},x),(\mathsf{TL},l)]
\end{array}$$

The specification has the same general structure as before, but now asserts that the heap grows by n many objects, that no function calls occur, and that both the number and the nesting depth of method invocations are linear in n.

$$\mathit{ST}\ \mathsf{ILIST}\ \mathit{double}\ z\ E\ h\ h'\ v\ p = \forall\, n\ a\ X.\ \left(\begin{array}{l} (\mathit{eval}\ E\ z\ =\ [\mathsf{Ref}\, a]\ \wedge\ h,a \models_X n) \\ \longrightarrow\ |\mathit{dom}(h')| = |\mathit{dom}(h)| + n\ \wedge \\ \qquad p = \langle (35n+18)\ 0\ (n+1)\ (n+1) \rangle \end{array}\right)$$

We prove the following resource property for an arbitrary x:

$$\rhd\, \mathsf{ILIST} \diamond \mathit{double}([x]) : \mathit{ST}\ \mathsf{ILIST}\ \mathit{double}\ [x]$$

The proof has the same overall structure as the previous one, where the auxiliary lemma now reads

$$\mathit{good}_{ST}(\{\ (\mathsf{ILIST} \diamond \mathit{double}([t]), \mathit{ST}\ \mathsf{ILIST}\ \mathit{double}\ [t]\ \})\,.$$

5 Related Work

Most closely related to our work on the meta-theoretical side are Nipkow's implementation of Hoare logic in Isabelle/HOL [24], the Java-light logic by von Oheimb [29], Kleymann's thesis [15], and Hofmann's [10] work on completeness of program logics. The logic by Nipkow in [24] is for a while-language with parameterless functions, with proofs of soundness and completeness. Several techniques we use in our treatment are inspired by this work, such as modelling of the heap via mini-heaps. However, we have made progress on the treatment of mutual recursion and adaptation. Several options for formalising either VDM or Hoare-style program logics have been explored by Kleymann [15]. In particular this work demonstrates how to formalise an adaptation rule that permits to modify auxiliary variables. The techniques used in our completeness proof are based on those by one of the authors in [10].

The program logic for Java-light by von Oheimb [29] is encoded in Isabelle/HOL and proven sound and complete. It covers more object-oriented features, but works on a higher level than our logic for a bytecode language and does not cover resources. Moreover, it is hardly suitable for concrete program verification.

With respect to other relevant program logics, de Boer [8] presents a sound and complete Hoare-style logic for an sequential object-oriented language with inheritance and subtyping. In contrast to our approach, the proof system employs a specific assertion language for object structures, whose WP calculus is heavily based on syntactical substitutions. Recently a tool supporting the verification of annotated programs (flowcharts) yielding verification conditions to be solved in HOL has been produced [5]. This also extends to multi-threaded Java [2].

Abadi and Leino combine a program logic for an object-oriented language with a type system [1, 16]. The language supports sub-classing and recursive object types and attaches specifications as well as types to expressions. In contrast to our logics, it uses a global store model, with the possibility of storing pointers to arbitrary methods in objects. As a result of this design decision this logic is incomplete. An implementation of this logic and a verification condition generator are described in [28].

Several projects aim at developing program logics for subsets of Java, mainly as tools for program development. Müller and Poetzsch-Heffter present a sound Hoare-style logic for a Java subset [22]. Their language covers class and interface types with subtyping and inheritance, as well as dynamic and static binding, and aliasing via object references, see also the Jive tool [20]. As part of the LOOP project, Huisman and Jacobs [13] present an extension of a Hoare logic that includes means for reasoning about abrupt termination and side-effects, encoded in the PVS theorem prover. Krakatoa [18] is a tool for verifying JML-annotated Java programs that acts as front-end to the Why system [9], using Coq to model the semantics and conduct the proofs. Why produces proof-obligations for programs in imperative-functional style via an interpretation in a type theory of effects and monads. Similarly, the target of the JACK environment [6] are verification conditions for the B system from JML annotations, though much effort is invested in making the system usable by Java programmers. We also mention [19], which embeds a Hoare logic in HOL, following previous work by Mike Gordon, to reason about pointer programs in a simple while-language. As an example, the authors provide an interactive proof in ISAR of the correctness of the Schorr-Waite algorithm.

Finally, [21] proves properties of the JVM in ACL2 directly from invariants and the operational semantics, that is without resorting to a VCG.

6 Conclusions

This paper has presented a resource-aware program logic for Grail, together with proofs of soundness and completeness. Our logic is unique in combining reasoning about resources for a general object-oriented language with completeness results for this logic. Grail is an abstraction over the JVM bytecode language which can be given a semi-functional semantics. We have developed admissible rules to work with mutually recursive methods, including parameter adaptation. While the logic already covers dynamic method invocation, we left a formalisation of the class hierarchy for future research. The logic has been encoded in the Isabelle/HOL theorem prover, and the formalisation of the soudness and completeness proofs provide additional confidence in the results. We demonstrated the usability of the logic by giving some examples, where we proved concrete resource bounds on space and time. These example programs have been generated by the Camelot compiler, indicating that the logic is sufficiently expressive to serve as the target logic in our proof-carrying-code infrastructure. In order to mechanise the verification of concrete programs, we are currently defining more specialised logics for various resources. These logics are defined in terms of the logic presented in this paper and thus inherit crucial properties such as soundness.

Acknowledgements This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing. We would like to thank all the MRG members as well as Tobias Nipkow and his group for discussions about formalising program logics.

References

1. M. Abadi and R. Leino. A Logic of Object-Oriented Programs. In *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 682–696. Springer, 1997.
2. E. Abraham-Mumm, F. S. de Boer, W. P. de Roever, , and M. Steffen. A tool-supported proof system for mutlithreaded Java. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *FMCO 2002: Formal Methods for Component Objects, Proceedings*, LNCS. Springer, 2003.
3. D. Aspinall and M. Hofmann. Another Type System for In-Place Update. In *ESOP'02 — European Symposium on Programming*, volume 2305 of *LNCS*, pages 36–52. Springer, 2002.
4. L. Beringer, K. MacKenzie, and I. Stark. Grail: a Functional Form for Imperative Mobile Code. *Electronic Notes in Theoretical Computer Science*, 85(1), 2003.
5. F. d. Boer and C. Pierik. Computer-aided specification and verification of annotated object-oriented programs. In B. Jacobs and A. Rensink, editors, *FMOODS 2002*, volume 209 of *IFIP Conference Proceedings*, pages 163–177. Kluwer, 2002.
6. L. Burdy and A. Requet. Jack: Java applet correctness kit. In *4th Gemplus Developer Conference,*, 2002.

7. C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A Certifying Compiler for Java. In *PLDI'00 — Conference on Programming Language Design and Implementation*, pages 95–107. ACM Press, 2000.
8. F. de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures*, volume 1578 of *LNCS*, pages 135–149. Springer, 1999.
9. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003. http://www.lri.fr/ filliatr/ftp/publis/why-tool.ps.gz.
10. M. Hofmann. Semantik und Verifikation. Lecture Notes, WS 97/98 1998. TU Darmstadt.
11. M. Hofmann. A Type System for Bounded Space and Functional In-place Update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
12. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*, pages 185–197, New Orleans, LA, USA, Jan. 2003. ACM Press.
13. M. Huisman and B. Jacobs. Java Program Verfication via a Hoare Logic with Abrupt Termination. In *FASE'00 — Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*, pages 284–303. Springer, 2000.
14. C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
15. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.
16. R. Leino. Recursive Object Types in a Logic of Object-oriented Programs. *Nordic Journal of Computing*, 5(4):330–360, 1998.
17. K. MacKenzie and N. Wolverson. Camelot and Grail: Compiling a Resource-aware Functional Language for the Java Virtual Machine. In *TFP'03, Symposium on Trends in Functional Languages*, Edinburgh, Sep. 11–12, 2003.
18. C. Marche, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58:89, January 2004.
19. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.
20. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`, 2000.
21. J. S. Moore. Proving theorems about Java and the JVM with ACL2. *NATO Science Series Sub Series III Computer and Systems Sciences*, 191:227–290, 2003.
22. P. Müller and A. Poetzsch-Heffter. A Programming Logic for Sequential Java. In *ESOP'99 — European Symposium on Programming*, volume 1576 of *LNCS*, pages 162–176, 1999.
23. G. Necula. Proof-carrying Code. In *POPL'97 — Symposium on Principles of Programming Languages*, pages 106–116, Paris, France, January 15–17, 1997. ACM Press.
24. T. Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.
25. C. Pierik and F. d. Boer. A rule of adaptation for OO. Technical Report UU-CS-2003-032, Utrecht University, 2004.
26. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02 — Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 22–25, 2002.
27. D. Sannella and M. Hofmann. Mobile Resource Guarantees. EU OpenFET Project, 2002. http://www.dcs.ed.ac.uk/home/mrg/.
28. F. Tang. *Towards feasible, machine assisted verification of object-oriented programs*. PhD thesis, School of Informatics, University of Edinburgh, 2002.
29. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.