Towards certificate generation for linear heap consumption

Lennart Beringer[1], Martin Hofmann[2], Alberto Momigliano[1], Olha Shkaravska[2]

[1] Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland;
{lenb,amomigl1}@inf.ed.ac.uk

[2] Institut für Informatik, Ludwig-Maximilians-Universität München, Oettingenstraße 67, 80538 München;
{mhofmann,shkaravska}@informatik.uni-muenchen.de

Abstract. We present a program logic for verifying the heap consumption of programs written in an abstract representation of the Java virtual machine language. The logic is defined by an expansion into the more general program logic presented in [2], but concrete program program verification may be performed purely on the abstract level, without recourse to the base logic. Format and interpretation of assertions closely match the interpretation of [9]'s type system for functional programs where the consumption of heap space is bounded by a linear function on the input size. The derived proof rules enable us to replay typing derivations in a theorem prover, given assertions about individual methods. The resulting verification conditions are of limited complexity, and may be discharged without significant programmer intervention.

1 Introduction

For the effective use of mobile code, certifying the resource consumption of a program is of great concern. The Mobile Resource Guarantees (MRG) project [20] is developing Proof-Carrying Code (PCC) technology [17] to endow mobile code with certificates of bounded resource consumption. These certificates are generated by a compiler which, in addition to translating high-level programs into machine code, derives formal proofs based on programmer annotations and program analysis. The specific architecture developed by MRG considers programs written in the high-level language *Camelot* [15], an ML-like functional language with type systems for reasoning about resource consumption. The compilation targets the Java bytecode language (JVML) and proceeds in two steps. The first step transforms Camelot programs into an abstract form of JVML called *Grail* [4] by performing well-known functional program manipulations such as monomorphisation and let-normalisation. Datatypes are represented using a single class of objects, while Camelot functions translate into static method invocations [15, 23]. Subsequently, Grail programs may be expanded into virtual machine format in a *reversible* and resource-preserving way. Relying on a bijection between an (impure) functional and an imperative semantics of Grail, this expansion allows us to certify programs at the Grail level while employing Java's classfile mechanism for program transmission: at code consumer side, a transmitted program is simply translated back into Grail before proof checking.

As the basis for reasoning and certificate generation, we employ a VDM-style [10] program logic for Grail where assertions are boolean functions over evaluation environments, pre- and post-heaps, and result values (see below). In [2] we reported on a full formalisation of the operational semantics and the program logic in the theorem prover Isabelle/HOL and proved the soundness and (relative) completeness of the logic with respect to an interpretation of partial correctness, extending previous work on formalisations of Hoare-style program logics for imperative languages [8, 11, 18].

While this formalisation was useful for obtaining the desired meta-logical properties, it proved less suited for concrete program verification. In particular, the complexity of side conditions and the necessity to instantiate existential quantifiers and perform case-splits on the form of data values made automatic verification difficult to achieve.

In this paper we therefore present a derived logic which is defined on top of [2]'s logic (henceforth dubbed the *core* or *base* logic) and is more directly related to Camelot's compile time analysis. Indeed, by encoding a Grail-level interpretation of the type system of Hofmann and Jost for heap space consumption [9], we obtain a smooth transition from program analysis to program verification. Certificate generation amounts to providing the correct invariants and proving auxiliary lemmas which are needed to discharge side conditions occurring from a verification condition generator. The complexity of the side conditions in the new logic is significantly lower than that of side conditions in the base logic, making automatic verification more feasible.

The encoding of the type system is achieved by introducing a fixed form of assertions whose definition in terms of the base logic corresponds to the soundness condition of [9]. During program verification, however, this definition

does not need to be unfolded – we give a *derived* VDM proof system where judgements only mention assertions of the restricted form, and the only side conditions are arithmetic (in-) equalities, context lookup operations and updates. The only time when the definition of restricted assertions is unfolded is thus the proof of the derived rules. We have performed such proofs for the rules presented in this paper, and are currently working on extending them to a more general setting (see Section 7). Continuing our earlier work on formalisation, the whole development presented in this paper has been formalised in Isabelle/HOL: each proof rule given in this paper is in fact a lemma which has been formally proven. Any proof for a concrete program using the logic of derived assertions may thus conceptually be expanded into a proof in the base logic, and thus (via the soundness result) ultimately into a statement about the operational semantics. Indeed, the aim to produce machine-checkable proofs was one of the reasons for formalising the development in a general-purpose theorem prover instead of using more specialised (but at the time still unmechanised) logics such as Separation Logic [19].

Outline The remainder of this document is structured as follows. We first summarise some main components of the MRG architecture, including the two language levels, the type system of [9] and the memory management strategy implemented in the Camelot compiler. We then define the syntax and the semantics of derived assertions in Section 3 and give some proof rules and auxiliary lemmas in Section 4. Finally, we outline the application of the logic to example programs in Sections 5 and 6, before discussing extensions and future and related work in Section 7.

2 Components of the MRG architecture

2.1 Grail

The Grail representation of code retains the object and method structure of Java bytecode and represents method bodies as sets 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. The main characteristic of Grail is its dual identity: its (impure) call-by-value functional semantics may be 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, we require that actual arguments in function calls coincide syntactically with the formal parameters of the function definitions. Together with Administrative-Normal-Form (ANF)-style normalisation of let-expressions, 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 [4]. The formal syntax of expressions

$$
\begin{aligned}
e \in \mathit{expr} ::= {} & \texttt{null} \mid \texttt{int}\ i \mid \texttt{var}\ x \mid \texttt{prim}\ \mathit{op}\ x\ x \mid \texttt{new}\ c\ [t_1 := x_1, \ldots, t_n := x_n] \mid \\
& x.t \mid x.t{:=}x \mid c \diamond t \mid c \diamond t{:=}x \mid \texttt{let}\ x{=}e\ \texttt{in}\ e \mid e\ ;\ e \mid \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{call}\ f \mid c.m(\bar{a}) \\
a \in \mathit{args} ::= {} & \texttt{var}\ x \mid \mathsf{null} \mid i
\end{aligned}
$$

is defined over mutually disjoint sets of method names, class names, function names (i.e. labels of basic blocks), (static) field names and variables, ranged over by m, c, f, t, and x, respectively. In the grammar, i ranges over integers and 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. Heap references are either null or of the form $\mathsf{Ref}\ l$ where $l \in \mathcal{L}$ is a location (represented by a natural number). Formal parameters of method invocations may be integer or object variables, where *self* is a reserved variable name. Actual arguments are sequences of variable names or immediate values.

Expressions represent basic blocks and are built from operators, constants, and previously computed values (names). Expressions correspond to primitive sequences of bytecode instructions that may, as a side effect, alter the heap. For example, $x.t$ and $x.t{:=}y$ represent (non-static) `getfield` and `putfield` instructions, while $c \diamond t$ and $c \diamond t{:=}y$ denote their static counterparts. The binding $\texttt{let}\ x{=}e_1\ \texttt{in}\ e_2$ 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 instruction $c.m(\bar{a})$ represents 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 *funtable* mapping function identifiers to an expression and a list of formal arguments, and a table *methtable* associating method parameters and initial basic blocks to class names and method identifiers. The formal basis of the program logic is an operational semantics that models the functional interpretation of Grail and is expressed as a big-step evaluation relation $E \vdash h, e \Downarrow h', v$. For expression e, such a judgement relates an (initial) variable environment $E \in \mathcal{E}$ and an initial heap $h \in \mathcal{H}$ to a final heap $h' \in \mathcal{H}$ and the result value $v \in \mathcal{V}$.

2.2 The core program logic

In our program logic [2], judgements take the form $G \rhd e : P$ where e is a Grail expression, G a VDM context used for storing verification assumptions for recursive methods and functions, and P an assertion, i.e. a predicate (in the meta-logic) over semantic components that relates the initial and final heaps, the initial environment, and the result value: $P : \mathcal{E} \to \mathcal{H} \to \mathcal{H} \to \mathcal{V} \to \mathcal{B}$, where $\mathcal{B}$ is the set of booleans. Satisfaction of a specification P by program e is denoted by $\models e : P$ and asserts that $E \vdash h, e \Downarrow h', v$ implies $P\,E\,h\,h'\,v$. While the rules for defining the operational semantics are omitted, the rules defining the program logic are given in Appendix A. In [2] we proved the soundness and (relative) completeness of the program logic with respect this (partial) interpretation, i.e. the statement $\emptyset \rhd e : P \iff \models e : P$.

2.3 Analysis and compilation of Camelot programs

The high-level programming language used in the MRG project, Camelot, is an ML-like first-order functional language with polymorphism and algebraic datatypes. In the type system of [9], the syntax of types includes annotations indicating the amount of free heap required for each occurrence of a constructor for that type. For example, the datatype

```
type iList = Nil | Cons of int * iList
```

where a `Nil` value is represented by the `nil` pointer is given a type $\mathtt{L}(k)$ where k is an annotation indicating how much free heap space is required for each occurrence of a `Cons`-node. A list of length N thus comes equipped with $N * k$ free heap cells. The typing $\Gamma, n \vdash e : T, m$ of an expression e not only gives a (conservative) estimate n of the amount of free heap space required for evaluating e, but also contains an (again conservative) estimate m of the amount of heap left over after the evaluation. The latter annotation is measured relative to the size of the result value, not the (input) free variables. Thus, typing judgements may be combined in a compositional way, as shown in the rule for `let`-expressions

$$\frac{\Gamma_1, n \vdash e_1 : A, k \quad \Gamma_2, x : A, k, \vdash e_2 : B, m}{\Gamma_1, \Gamma_2, n \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B, m}$$

The typing rule associated to a constructor stipulates that enough space be provided for allocating the data, including the annotation value. For example, the rule for the constructor `Cons` is

$$\frac{n \geq 1 + k + m}{\Gamma, h : \mathtt{int}, t : \mathtt{L}(k), n \vdash \mathtt{Cons}(h,t) : \mathtt{L}(k), m}$$

During pattern-matching, the heap space inhabited by the value may either be reclaimed (in which case the cell is destroyed and may not be referred to in the continuation of the program) or be retained:

$$\frac{\Gamma, n \vdash e_1 : A, m \qquad \Gamma, h : \mathtt{int}, t : \mathtt{L}(k), n + 1 + k \vdash e_2 : A, m}{\Gamma, x : \mathtt{L}(k), n \vdash \mathtt{match}\ x\ \mathtt{with}\ \ \mathtt{Nil} \Rightarrow e_1 |\ \mathtt{Cons}(h,t)@\_ \Rightarrow e_2 : A, m}$$

$$\frac{\Gamma, n \vdash e_1 : A, m \qquad \Gamma, h : \mathtt{int}, t : \mathtt{L}(k), n + k \vdash e_2 : A, m}{\Gamma, x : \mathtt{L}(k), n \vdash \mathtt{match}\ x\ \mathtt{with}\ \ \mathtt{Nil} \Rightarrow e_1 |\ \mathtt{Cons}(h,t) \Rightarrow e_2 : A, m}$$

The program annotation $@\_$ indicates the destructive nature of the pattern match in the first rule, as used in the following example, insertion sort:

```
let ins a l = match l with
                Nil -> Cons(a,Nil)
              | Cons(x,t)@_ -> if a < x then Cons(a,Cons(x,t))
                                        else Cons(x, ins a t)
let sort l =  match l with Nil -> Nil
                         | Cons(a,t)@_ -> ins a (sort t)
```

As a result, this program may be given a type which indicates that no space is consumed during the evaluation:

```
ins : 1, int -> L(0) -> L(0), 0
sort : 0, L(0) -> L(0), 0.
```

In function `ins`, a single additional cell is needed: if the list is empty, we need to construct one node, while in the case where `l` is non-empty, either two `Cons` cells are needed immediately, or one immediately and one in the recursive call to `ins`, but one cell is also gained during the destructive pattern match. Function `sort` does not consume heap space since the one cell needed when calling `ins` has been gained during the destructive pattern match.

The process of type inference first builds a skeleton typing derivation where the numerical values $n, m, \ldots$ are interpreted as (rational) variables. The side conditions over these variables are then fed into a solver for linear-programming problems [9]. At the Camelot level, the soundness of the type system with respect to memory consumption relies on a notion of *benign sharing* which ensures that deallocated cells cannot be accessed in the code following the deallocation primitive. As a by-product, this condition also ensures functional correctness.

The Grail code emitted for a program such as insertion sort contains a hand-crafted memory manager, independent from the JVM garbage collection and implemented as a freelist, pointed to by the static field FLIST. The `@_` directive results in the cell against which the pattern match is performed being inserted into the freelist, using the method `free` with parameter *node* and body

$$\texttt{let f}=\texttt{D}\diamond\mathsf{FLIST}\ \texttt{in node.}\mathsf{NEXT}:=\texttt{f ; D}\diamond\mathsf{FLIST}:=\texttt{node}$$

The application of a datatype constructor translates into the invocation of method `make`, which draws an entry from the freelist if that is non-empty, and allocates a new object otherwise:

$$\mathit{methtable} \equiv \begin{bmatrix} \texttt{D.alloc}() \mapsto \texttt{let f}=\texttt{D}\diamond\mathsf{FLIST}\ \texttt{in let b}=\texttt{prim}\ \mathit{isNull}\ \texttt{f f in} \\ \qquad\qquad \texttt{if b then new D}\ [\,]\ \texttt{else let tl}=\texttt{f.}\mathsf{NEXT}\ \texttt{in D}\diamond\mathsf{FLIST}:=\texttt{tl ; f} \\ \texttt{D.fill}(\texttt{x},\texttt{tag},\texttt{v},\texttt{w}) \mapsto \texttt{x.}\mathsf{TAG}:=\texttt{tag ; x.}\mathsf{HD}:=\texttt{v ; x.}\mathsf{TL}:=\texttt{w ; x} \\ \texttt{D.make}(\texttt{tag},\texttt{v},\texttt{w}) \mapsto \texttt{let x}=\texttt{D.alloc}()\ \texttt{in D.fill}(\texttt{x},\texttt{tag},\texttt{v},\texttt{w}) \end{bmatrix}$$

Thus, the Grail-level interpretation of a typing judgement $\Gamma, n \vdash e : T, m$ says that no object allocation takes place during the evaluation of *e*, provided the initial freelist is of length (at least) *n*. The task of our (derived) program logic is to verify this property for the Grail code emitted by the compiler [1].

3 Format and interpretation of assertions

For the purpose of the following three sections, derived assertions are defined over the types

$$T \in \mathbf{T} ::= \mathbf{1} \mid \mathbf{I} \mid \mathbf{L}(n)$$

which represent, respectively, the unit (void) type, the integer type, and the type of lists where each occurrence of the `Cons` constructor is equipped with $n \in \mathbb{N}$ additional free heap cells and the `Nil` constructor does not consume space. Assertions take the form

$$[\![U, n, \Gamma \blacktriangleright T, m]\!]$$

with the following components:

[1] In addition to the code mentioned, our compiler also emits code for creating the initial freelist. At the top-level, an appropriate soundness criterion would thus claim that the execution of the whole program (wrapper code and application code) does not consume more memory than claimed by the annotations. While we have indeed verified such statements formally, our discussion concentrates on the correctness of inner functions, using the interpretation where an initial freelist is assumed to exist.

$n, m \in \mathbb{N}$ represent the numerical results from the analysis which relate the initial and final length of the freelist. While Camelot's resource inference permits rational weights, our formalisation is restricted to naturals.
Γ (a partial map from program variables to types) represents the typing context in which an expression may be typed.
U (a finite set of program variables) is used to restrict the variables to which a statement may refer.
T indicates the return type.

The proof rules (given in the following section) are defined such that the verification of a program infers the set U during the verification condition generation. In effect, the restricted context $\Gamma\lfloor_U$ amounts to the minimal context in with an expression e may be typed.

The semantic definition of an assertion $[\![U, n, \Gamma \blacktriangleright T, m]\!]$ yields a VDM assertion in the base logic, i.e. a boolean function over the semantic components environment, pre-heap, post-heap and return value. We first define some auxiliary predicates. For each type T, we define a representation predicate which associates to each value v of type T the heap region R inhabited by v in a heap h, and the number n of free heap cells associated with it. For the above collection of types these predicates $v, h \models_T R, n$ are defined as follows:

$$\frac{LIST(n, r, R, h)}{r, h \models_{\mathbf{L}(k)} R, k * n}\ \text{REGL} \qquad \frac{}{i, h \models_{\mathbf{I}} \emptyset, 0}\ \text{REGI} \qquad \frac{}{v, h \models_{\mathbf{1}} \emptyset, 0}\ \text{REGU}$$

Here, the list predicate $LIST(n, r, R, h)$ is satisfied if reference r in heap h points to a (cycle-free) linked list of length n, whose cells inhabit exactly locations R. The definition directly reflects the layout of data values implemented by the Camelot compiler, and resembles the predicates used in [19].

Next, we define a predicate $\Gamma, U \models^E_h n, R$ that indicates the amount n of free heap associated to the variables in $\Gamma\lfloor_U$ and the heap region R inhabited by the corresponding data structures. The predicate is defined by induction on U:

$$\frac{}{\Gamma, \emptyset \models^E_h 0, \emptyset}\ \text{SIZE1} \qquad \frac{x \in U \quad E\langle x \rangle, h \models_{\Gamma(x)} R_1, n \quad \Gamma, U \setminus \{x\} \models^E_h m, R_2 \quad R_1 \cap R_2 = \emptyset}{\Gamma, U \models^E_h n + m, R_1 \cup R_2}\ \text{SIZE2}$$

and incorporates a separation condition between different data structures.

The interpretation of an assertion $[\![U, n, \Gamma \blacktriangleright T, m]\!]$ is now defined by

$$\begin{aligned}[\![U, n, \Gamma \blacktriangleright T, m]\!] \equiv\ \lambda E\ h\ h'\ v.\ \forall\, q\ F\ R.\ \ & (\exists\, N\ K.\ freelist(h, F, N)\ \wedge\ \Gamma, U \models^E_h K, R \wedge R \cap F = \emptyset \wedge\ n + K + q \leq N) \\ & \longrightarrow\ (\exists\, R'\ S\ M\ G.\ v, h' \models_T R', S\ \wedge\ freelist(h', G, M)\ \wedge\ R' \cap G = \emptyset\ \wedge \\ & \qquad modified(F \cup R, h, h')\ \wedge\ (R' \cup G) \subseteq (R \cup F)\ \wedge \\ & \qquad m + S + q \leq M\ \wedge\ dom(h) = dom(h'))\end{aligned}$$

and asserts that, whenever

- the initial heap contains a freelist of length N, inhabiting region F,
- the region R inhabited by the data structures $\Gamma\lfloor_U$ is disjoint from the freelist region F, and
- the length N of the freelist is at least the amount K of heap owned by $\Gamma\lfloor_U$, plus the additionally required size n and some constant q,

then there exist numbers M and S, and regions R' and G such that

- the result v (according to the type T) inhabits region R' (in the final heap) and is of size S,
- the final heap contains a freelist of length M inhabiting region G,
- the result and the final freelist do not overlap,
- both G and R' are contained in the initial freelist region F, extended by the regions pointed to (in the initial heap) by the variables in $\Gamma\lfloor_U$,
- locations that are neither part of the freelist nor used by variables from $\Gamma\lfloor_U$ remain unchanged,
- the freelist grows (or shrinks) as predicted by the analysis of [9], i.e. the final length M is at least the size of the result, plus the analysis number m and the offset q, and
- no new objects are allocated.

The first auxiliary predicate, *freelist*, is defined by

$$freelist(h, F, N) \equiv FL(N, h\langle D.\mathsf{FLIST}\rangle, F, h)$$

and expresses that in heap h, the static field $D.\mathsf{FLIST}$ points to a list of length N inhabiting locations F, where $FL(\_, \_, \_, \_)$ is defined analogously to the predicate $LIST(\_, \_, \_, \_)$. The second auxiliary predicate is defined by

$$modified(X, h, h') \equiv \forall l \in dom(h) \setminus X.\ h(l) = h'(l).$$

4 Proof rules

The design of the proof rules was guided by the aim to minimise the complexity of verification conditions that arise from side conditions, and to mirror the high-level typing rules. Indeed, the granularity of the proof rules matches that of the typing system: match statements and constructor applications are verified as single entities, i.e. only the soundness proof of the rules inspects the constituent instructions of the corresponding methods. In order to statically approximate the condition of benign sharing, we adopt a largely (affine) linear context management, which in combination with the separation condition in the predicate $\Gamma, U \models^E_h n, R$ results in a rather strict typing discipline. While this clearly leaves room for future improvement (see Section 7), the system as given suffices for verifying a significant class of example programs. The rules come in three groups.

Syntax-directed rules We first present the rules for the various syntactic constructs of Grail. There are no proof rules for object creation and (virtual or static) field access instructions, since these operations are only performed inside the memory management methods.

$$\frac{m \leq n}{G \rhd \texttt{null} : [\![\emptyset, n, \Gamma \blacktriangleright \mathbf{L}(k), m]\!]}\ \textsc{Null} \qquad \frac{m \leq n}{G \rhd \texttt{int}\ i : [\![\emptyset, n, \Gamma \blacktriangleright \mathbf{I}, m]\!]}\ \textsc{Int}$$

$$\frac{m \leq n}{G \rhd \texttt{var}\ x : [\![\{x\}, n, \Gamma \blacktriangleright \Gamma(x), m]\!]}\ \textsc{Var} \qquad \frac{\{x,y\} \subseteq dom(\Gamma) \quad m \leq n}{G \rhd \texttt{prim}\ op\ x\ y : [\![\{x,y\}, n, \Gamma \blacktriangleright \mathbf{I}, m]\!]}\ \textsc{Prim}$$

$$\frac{G \rhd e_1 : [\![U_1, n, \Gamma \blacktriangleright \mathbf{1}, m]\!] \quad G \rhd e_2 : [\![U_2, m, \Gamma \blacktriangleright T, k]\!] \quad U_1 \cap U_2 = \emptyset}{G \rhd e_1\ ;\ e_2 : [\![U_1 \cup U_2, n, \Gamma \blacktriangleright T, k]\!]}\ \textsc{Comp}$$

$$\frac{G \rhd e_1 : [\![U_1, n, \Gamma \blacktriangleright \mathbf{L}(k), l]\!] \quad G \rhd e_2 : [\![U_2, l, \Gamma, x : \mathbf{L}(k) \blacktriangleright T, m]\!] \quad U_1 \cap (U_2 \setminus \{x\}) = \emptyset}{G \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : [\![U_1 \cup (U_2 \setminus \{x\}), n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{Let}$$

$$\frac{G \rhd e_1 : [\![U_1, n, \Gamma \blacktriangleright T, m]\!] \quad G \rhd e_2 : [\![U_2, n, \Gamma \blacktriangleright T, m]\!]}{G \rhd \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : [\![U_1 \cup U_2, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{If}$$

$$\frac{(G \cup \{\texttt{call}\ f, [\![U, n, \Gamma \blacktriangleright T, m]\!]) \rhd (\mathit{funtable}\ f) : [\![U, n, \Gamma \blacktriangleright T, m]\!]}{G \rhd \texttt{call}\ f : [\![U, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{Call}$$

$$\frac{\begin{array}{l}(G \cup \{(c.M(\overline{a}), [\![U, n, \Gamma \blacktriangleright T, m]\!])\}) \rhd \\ \quad (\mathit{methtable}\ c\ M) : (\lambda\ E\ h\ h'\ v.\ \forall E'.E = \mathit{frame}\ \mathsf{null}\ (\mathit{params}\ c\ M)\ \overline{a}\ E' \longrightarrow [\![U, n, \Gamma \blacktriangleright T, m]\!]\ E'\ h\ h'\ v)\end{array}}{G \rhd c.M(\overline{a}) : [\![U, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{InvS}$$

In the last two rules the VDM context in the hypothesis is extended by entries which allow the verification of the (function or method) bodies to use the recursive assumption. The construction of E in rule InvS corresponds to the creation of a new frame. This treatment of recursive invocations is directly lifted from our base logic and follows previous work on formalised program logics with procedures [8, 18].

Rules for freelist management We have rules for non-destructive and destructive match operations, and for constructor `Cons`.

$$\frac{\Gamma(x) = \mathbf{L}(k) \quad h \neq t \quad x \notin \{h\} \cup (U \setminus \{t\}) \quad G \rhd e : [\![U, n+k, \Gamma, h : \mathbf{I}, t : \mathbf{L}(k) \blacktriangleright T, m]\!]}{G \rhd \texttt{let}\ h = x.\mathsf{HD}\ \texttt{in}\ \texttt{let}\ t = x.\mathsf{TL}\ \texttt{in}\ e : [\![(U \setminus \{h,t\}) \cup \{x\}, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{Match}$$

$$\frac{\Gamma(x) = \mathbf{L}(k) \quad G \rhd e : [\![U, n+k+1, \Gamma, h : \mathbf{I}, t : \mathbf{L}(k) \blacktriangleright T, m]\!] \quad x \notin U \cup \{h,t\}}{G \rhd \texttt{let}\ h = x.\mathsf{HD}\ \texttt{in}\ \texttt{let}\ t = x.\mathsf{TL}\ \texttt{in}\ \mathtt{D.free}(x)\ ;\ e : [\![(U \setminus \{h,t\}) \cup \{x\}, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{DMatch}$$

$$\frac{\Gamma(y) = \mathbf{L}(k) \quad \Gamma(x) = \mathbf{I}}{G \rhd \mathtt{D.make}(1, x, y) : [\![\{x,y\}, m+k+1, \Gamma \blacktriangleright \mathbf{L}(k), m]\!]}\ \textsc{Cons}$$

Treating the freelist management operations atomically reflects the fact that their implementation should be shielded from the program verification. Indeed, the states at intermediate program points of these composite statements do not satisfy formulae of the restricted form – they contain dangling pointers and incompletely built data structures.

Logical rules Finally, we give three structural rules.

$$\frac{G \rhd e : [\![U, n, \Gamma \blacktriangleright T, m]\!] \quad n \leq n' \quad m' \leq m \quad U \subseteq V}{G \rhd e : [\![V, n', \Gamma \blacktriangleright T, m']\!]}\ \textsc{Generalise}$$

$$\frac{G \rhd e : [\![U, n, \Gamma \blacktriangleright T, m]\!]}{G \rhd e : [\![U, n+k, \Gamma \blacktriangleright T, m+k]\!]}\ \textsc{Shift} \qquad \frac{G \rhd e : [\![U, n, \Gamma \blacktriangleright T, m]\!] \quad \forall\, x \in U.\ \Delta(x) = \Gamma(x)}{G \rhd e : [\![U, n, \Delta \blacktriangleright T, m]\!]}\ \textsc{Context}$$

5 Example: verification of Insertion sort

For the Camelot code given in Section 2.3, our compiler emits Grail code which extends the earlier method table to

$$\mathit{methtable} \equiv \begin{bmatrix} \ldots \\ \mathsf{InsSort.ins}(\mathtt{a},\mathtt{l}) \mapsto \mathtt{call\ fIns} \\ \mathsf{InsSort.sort}(\mathtt{l}) \mapsto \mathtt{call\ fSort} \end{bmatrix}$$

with function definitions

```
fun fIns(a,l) = let b=prim isNull l l in if b then call f0_Ins else call f1_Ins
fun f0_Ins(a) = let l=null in D.make(1,a,l)
fun f1_Ins(a,l) = let v3=l.HD in let v2=l.TL in D.free(l) ;
                  let b=prim less a v3 in if b then call f2_Ins else call f3_Ins
fun f2_Ins(a,v2,v3) = let l=D.make(1,v3,v2) in D.make(1,a,l)
fun f3_Ins(a,v2,v3) = let l=InsSort.ins(a,v2) in D.make(1,v3,l)
fun fSort(l) = let b=prim isNull l l in if l then call f0_Sort else call f1_Sort
fun f0_Sort() = null
fun f1_Sort(l) = let v3=l.HD in let v2=l.TL in D.free(l) ;
                 let l=InsSort.sort(v2) in InsSort.ins(v3,l)
```

The outcome of the compile time type analysis (see Section 2.3) amounts to assertions

$$\begin{aligned} \mathit{Ins\_Spec} &\equiv [\![\{a,l\}, 1, [a : \mathbf{I}, l : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!] \\ \mathit{Sort\_Spec} &\equiv [\![\{l\}, 0, [l : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!] \end{aligned}$$

when formalised from the perspective of the method body: the variable names are the formal method parameters. In order to satisfy the requirements of rule VADAPTS, we collect these assertions in a method specification table

$$\begin{aligned} \mathit{MST} \equiv\ & \lambda\, M\ \overline{b}\ E\ h\ h'\ v.\ \mathit{if}\ M = \mathit{Ins}\ \mathit{then}\ \mathit{Ins\_Spec}(\mathit{frame}\ \mathsf{null}\ ((\mathit{params}\ \mathsf{InsSort}\ \mathit{ins})@[\mathit{self}])\ \overline{b}\ E)\ h\ h'\ v\ \mathit{else} \\ & \mathit{if}\ M = \mathit{Sort}\ \mathit{then}\ \mathit{Sort\_Spec}(\mathit{frame}\ \mathsf{null}\ ((\mathit{params}\ \mathsf{InsSort}\ \mathit{sort})@[\mathit{self}])\ b\ E)\ h\ h'\ v\ \mathit{else} \\ & \mathit{False}, \end{aligned}$$

but we can immediately prove that the entries are in fact of the restricted assertion form:

Lemma 1. *For any x and y, we have*

$$\begin{aligned} \mathit{MST}\ \mathit{ins}\ [x, y, \mathsf{null}] &= [\![\{x,y\}, 1, [x : \mathbf{I}, y : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!] \\ \mathit{MST}\ \mathit{sort}\ [y, \mathsf{null}] &= [\![\{y\}, 0, [y : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!] \end{aligned}$$

Next, we construct a VDM context containing all syntactically occurring method invocations:

$$G_{\mathit{InsSort}} \equiv \left\{ \begin{array}{l} (\mathsf{InsSort.ins}(\mathtt{a},\mathtt{v_2}), \mathit{MST}\ \mathtt{ins}\ [\mathtt{a},\mathtt{v_2},\mathsf{null}]),\ (\mathsf{InsSort.ins}(\mathtt{v_3},\mathtt{l}), \mathit{MST}\ \mathtt{ins}\ [\mathtt{v_3},\mathtt{l},\mathsf{null}]), \\ (\mathsf{InsSort.sort}(\mathtt{v_2}), \mathit{MST}\ \mathtt{sort}\ [\mathtt{v_2},\mathsf{null}]) \end{array} \right\}$$

Each invocation is related to the corresponding entry in the method specification table.

Then, we show that each method body satisfies its specification:

Lemma 2. *We have* $G_{\mathit{InsSort}} \rhd \mathit{methtable}\ \mathsf{InsSort}\ \mathit{ins} : \mathit{Ins\_Spec}$ *and* $G_{\mathit{InsSort}} \rhd \mathit{methtable}\ \mathsf{InsSort}\ \mathit{sort} : \mathit{Sort\_Spec}$.

In both cases the fully automated verification proceeds by unfolding the definitions of the method body and the specification, applying rule GENERALISE and then the syntax-directed rules and the memory management rules until invocations of application methods are hit. Such invocations are verified by applying a specialisation of the general axiom rule VAX weakened according to rule SHIFT and CONTEXT and exploiting Lemma 1.

From Lemma 2 and rule VADAPTS (again see Appendix A or [2]) we finally obtain, again automatically, the correctness of invocations of *ins* and *sort* for *arbitrary* method arguments, in the empty VDM context:

Theorem 1. *We have* $\emptyset \rhd \mathsf{InsSort}.ins(x,y) : MST\ ins\ [x,y,\mathsf{null}]$ *and* $\emptyset \rhd \mathsf{InsSort}.sort(x) : MST\ sort\ [x,\mathsf{null}]$.

6 Improved treatment of merge points

While all Grail functions in the previous example have a single call point, a function may in general be called from more than one place. Indeed, the Camelot compiler avoids code duplication when compiling conditionals with non-trivial branch conditions by introducing merge points. As an example, the compilation of the *siftdown* function

```
let siftdown w t1 t2 =
match t1 with Leaf -> Node(w,Leaf,Leaf)
            | Node(v,t11,t12)@_ ->
              begin
                match t2 with
                 Leaf -> if w < v then Node(w, Node(v,Leaf,Leaf), Leaf)
                                  else Node(v, Node(w,Leaf,Leaf), Leaf)
                | Node(u, t21,t22)@_ ->
                    if w < u & w < v then
                          Node(w, Node(v,t11,t12), Node(u,t21,t22))
                    else if u < w & u < v then
                          Node(u, Node(v,t11,t12), siftdown w t21 t22)
                    else  Node(v, siftdown w t11 t12, Node(u,t21,t22))
              end
```

in an implementation of heapsort yields a call graph with merge points in the two conditionals where the branch condition is a conjunction. Using the proof rules given in Section 4, each call to the first merge point function would cause the corresponding function body to be unfolded once, and each verification of that body would involve three verifications of the inner merge point function. In order to achieve a verification in which each function is verified only once, one could add entries for such merge points in the VDM context G – but then the program analysis would be required to communicate analysis results for functions. An alternative uses the additional VDM rule

$$\frac{G \cup \{(\texttt{call}\ f, Q)\} \rhd e : P \quad provable(G \cup \{(\texttt{call}\ f, Q)\}, G)}{G \rhd e : P}\ \textsc{CutCall}$$

Here the property $provable(D,G)$ holds if G and D are finite and for all e and Q, $(e,Q) \in D$ implies $G \rhd e : Q$. This rule follows immediately from the rule VCUT given in Appendix A, and allows the VDM context to be extended by further entries. Applying this rule in the (immediate) dominator of a merge point forces all calls to the merge point function to use the same specification Q. In the case of the special assertions $[\![U,n,\Gamma \blacktriangleright T,m]\!]$, we now apply rule

$$\frac{(\texttt{call}\ f, [\![U,n,\Gamma|_{params(f)} \blacktriangleright T,m]\!]) \in G \quad U \subseteq params(f)}{G \rhd \texttt{call}\ f : [\![U,n,\Gamma \blacktriangleright T,m]\!]}\ \textsc{CallMP}$$

when calling a merge point function, while other calls use a simplified call rule

$$\frac{G \rhd funtable\, f : [\![U,n,\Gamma \blacktriangleright T,m]\!]}{G \rhd \texttt{call}\ f : [\![U,n,\Gamma \blacktriangleright T,m]\!]}\ \textsc{Jump}$$

that unfolds the body without inserting a recursive assumption to the context. Notice that both hypotheses of rule CALLMP are side conditions. In effect, an application of this rule instantiates the unknown assertion Q from rule

CUTCALL to $[\![U, n, \Gamma\downarrow_{params(f)} \blacktriangleright T, m]\!]$. The verification that the *body* of f satisfies this specification is performed once, in the proof of the predicate *provable*.

Using these rules we have indeed verified the implementation of heapsort alluded to above, visiting each function exactly once. To that end, we extended the syntax of types to

$$T \in \mathbf{T} ::= \ldots \mid \mathbf{T}(n) \mid \mathbf{R}(n_1, T, n_2)$$

where $\mathbf{T}(n)$ represents binary trees where each inner node consumes n heap cells and $\mathbf{R}(n_1, T, n_2)$ denotes an option type with contents of type $\texttt{int} \times T$, with annotations for the two constructors `None` and `Some`. In addition, we introduced the appropriate representation predicates and derived proof rules for constructor applications and (destructive and non-destructive) pattern matches, similar to those given in Section 4. The proof of

$$\emptyset \rhd \mathsf{HpSort}.sort(x) : [\![\{x\}, 0, [x : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!]$$

uses a verification of the *siftdown* method that visits each function body exactly once.

7 Conclusion

Because the MRG project aims to verify the consumption of a variety of resources, we have employed a general purpose logic as the basis of our formalisation. Our work is thus best compared to other work on mechanical or at least formal verification of pointer programs using variants of traditional (general purpose) Hoare logic. The first, though flawed, automated verification of pointer programs is [21] (see also [14] and [13]), where a model of the store is incorporated in the assertion logic. More recent is the verification of several algorithms, including list manipulating programs and the Schorr-Waite graph-marking algorithm, by Bornat [6] using the Jape system. This approach employs a Hoare logic for a while-language with components that are semantically modelled as pointer-indexed arrays. Separation conditions are expressed as predicates on (object) pointers.

Mehta and Nipkow [16] employ the same semantic model of the heap for reasoning about pointer programs in higher-order logics; the way in which datatypes are represented on the heap is very similar to ours. This effort extends earlier work by Nipkow et al. [18] on formalised proofs in HOL of soundness and (relative) completeness properties of program logics with respect to operational semantics. Again, an interactive proof in ISAR of the Schorr-Waite algorithm is given as an example. Tang [22] formalises the logic of Abadi and Leino [1] and implements a verification condition generator which includes a type inference algorithm.

An alternative to the usage of general purpose logics are specialised ones such as Separation Logic [19]. Indeed, the primitives of Separation Logic appear well suited to express the mutual separation of data structures, and their separation from the freelist, more succinctly. On the other hand, the VDM-style of our logic allows us to relate pre- and post states without the usage of auxiliary variables. Furthermore, properties such as the heap preservation in predicate *modifies* are more intensional than is usually the case in (Hoare-style) Separation Logic. Non-trivial applications of Separation Logic to date include Yang's treatment of the Schorr-Waite algorithm [24], the partial correctness of Cheney's copying garbage collector [5] and the verification of graph algorithms with aliasing and internal sharing [7].

In this paper, we have described a logic for derived assertions that allows the results of [9]'s analysis to be verified in Grail's bytecode logic. We have presented the logic for some specific datatypes and future work will aim to extend the development to algebraic datatypes in general. While the region calculation can be adapted relatively easily, the proofs of the memory management rules (destructive and non-destructive matches, constructors) appear more problematic. Another limitation of our system is the linear typing discipline, which we will aim to replace by more generous sharing and separation systems by transferring high-level type systems such as Konečný's [12] system for layered sharing and Aspinall and Hofmann's work on usage aspects [3] to the bytecode level.

Comparing the verification of the example programs with the verification of similar programs in the core bytecode logic demonstrates the general benefit of a proof system of derived assertions, concerning both the proof complexity and the potential for automation. Indeed, while verification in the bytecode logic appears to depend on the machinery of a general purpose theorem prover and manual intervention, a logic of derived assertions may be implementable in a stand alone prover with access to special solvers for arithmetic equalities and set containments.

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 members of the MRG project for the numerous discussions on program logics and proof generation.

References

1. M. Abadi and R. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214, pages 682–696. Springer-Verlag, New York, N.Y., 1997.
2. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, 2004. To appear.
3. D. Aspinall and M. Hofmann. Another type system for in-place update. In *Proccedings of the European Symposium On Programming*. Springer LNCS, 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. L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, 2004.
6. R. Bornat. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
7. R. Bornat, C. Calcagno, and P. W. O'Hearn. Local reasoning, separation and aliasing. In *Proceedings of SPACE 2004*, 2004.
8. M. Hofmann. Semantik und Verifikation. Lecture Notes, WS 97/98 1998. TU Darmstadt.
9. M. Hofmann 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*, 2003.
10. C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
11. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.
12. M. Konečný. Functional in-place update with datatype sharing. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications*. Springer LNCS, 2003.
13. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.
14. D. C. Luckham and N. Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, Oct. 1979.
15. K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware Functional Programming for the JVM. In S. Gilmore, editor, *Proceedings of the Fourth Symposium on Trends in Functional Programming*, 2003.
16. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003.
17. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
18. T. Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *Computer Science Logic (CSL 2002)*, LNCS 2471, pages 103–119, 2002.
19. 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.
20. D. Sannella and M. Hofmann. Mobile Resource Guarantees. EU Project IST-2001-33149, 2002–2004. http://groups/inf.ed.ac.uk/mrg/.
21. N. Suzuki. *Automatic verification of programs with complex data structures*. PhD thesis, Stanford University, 1976.
22. F. Tang. *Towards feasible, machine assisted verification of object-oriented programs*. PhD thesis, School of Informatics, University of Edinburgh, 2002.
23. N. Wolverson and K. MacKenzie. O'Camelot: Adding Objects to a Resource Aware Functional Language. In S. Gilmore, editor, *Proceedings of the Fourth Symposium on Trends in Functional Programming*, 2003.
24. H. Yang. *Local reasoning for stateful programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2001.

A Proof rules of the core program logic

We summarise the proof rules of the program logic presented in [2]. Judgements there contain a further component for reasoning about other resources. This has been deleted here for purposes of readability. Furthermore, we have simplified the rule VADAPTS and omitted the rules for virtual method invocation.

$$\frac{(e,P)\in\Gamma}{\Gamma\rhd e:P}\ \textsc{Vax} \qquad \frac{\Gamma\rhd e:P \quad \forall E\,h\,h'\,v.\ P\,E\,h\,h'\,v\longrightarrow Q\,E\,h\,h'\,v}{\Gamma\rhd e:Q}\ \textsc{Vconseq}$$

$$\frac{}{\Gamma\rhd \texttt{null}:\lambda E\,h\,h'\,v.\,h'=h\wedge v=\mathsf{null}}\ \textsc{Vnull} \qquad \frac{}{\Gamma\rhd \texttt{int}\ i:\lambda E\,h\,h'\,v.\,h'=h\wedge v=i}\ \textsc{Vint}$$

$$\frac{}{\Gamma\rhd \texttt{var}\ x:\lambda E\,h\,h'\,v.\,h'=h\wedge v=E\langle x\rangle}\ \textsc{Vvar}$$

$$\frac{}{\Gamma\rhd \texttt{prim}\ op\ x\ y:\lambda E\,h\,h'\,v.\,v=op\,E\langle x\rangle\ E\langle y\rangle\ \wedge h'=h}\ \textsc{Vprim}$$

$$\frac{}{\Gamma\rhd x.t:\lambda E\,h\,h'\,v.\,\exists l.E\langle x\rangle=Ref\ l\wedge h'=h\ \wedge v=h'(l).t}\ \textsc{Vgetf}$$

$$\frac{}{\Gamma\rhd x.t:=y:\lambda E\,h\,h'\,v.\,\exists l.E\langle x\rangle=Ref\ l\wedge h'=h[l.t\mapsto E\langle y\rangle]\wedge v=\bot}\ \textsc{Vputf}$$

$$\frac{}{\Gamma\rhd c\diamond t:\lambda E\,h\,h'\,v.\,h'=h\ \wedge v=h(c).t}\ \textsc{Vgetst}$$

$$\frac{}{\Gamma\rhd c\diamond t:=y:\lambda E\,h\,h'\,v.\,h'=h[c.t\mapsto E\langle y\rangle]\ \wedge v=\bot}\ \textsc{Vputst}$$

$$\frac{}{\Gamma\rhd \texttt{new}\ c\ [t_1:=x_1,\ldots,t_n:=x_n]:\lambda E\,h\,h'\,v.\,\exists l.freshloc(l,h)\wedge v=Ref\ l\ \wedge\ h'=h[l\mapsto(c,\{t_i:=E\langle x_i\rangle\})]}\ \textsc{Vnew}$$

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

$$\frac{\Gamma\rhd e_1:P_1 \quad \Gamma\rhd e_2:P_2}{\Gamma\rhd \texttt{let}\ x=e_1\ \texttt{in}\ e_2:\lambda E\,h\,h'\,v.\,\exists\,h_1\ w.(P_1\,E\,h\,h_1\,w)\ \wedge\ w\neq\bot\ \wedge\ (P_2\,(E\langle x:=w\rangle)\,h_1\,h'\,v)}\ \textsc{Vlet}$$

$$\frac{\Gamma\rhd e_1:P_1 \quad \Gamma\rhd e_2:P_2}{\Gamma\rhd e_1\,;e_2:\lambda E\,h\,h'\,v.\,\exists\,h_1.P_1\,E\,h\,h_1\,\bot\ \wedge P_2\,E\,h_1\,h'\,v}\ \textsc{Vcomp}$$

$$\frac{\Gamma\cup\{(\texttt{call}\ f,P)\}\rhd funtable\ f:\lambda E\,h\,h'\,v.P\,E\,h\,h'\,v}{\Gamma\rhd \texttt{call}\ f:P}\ \textsc{Vcall}$$

$$\frac{\Gamma\cup\{(c.m(\bar a),P)\}\rhd methtable\ c\ m:\lambda E\,h\,h'\,v.\forall\,E'.\ E=frame\ \mathsf{null}\ (params\ c\ m)\ \bar a\ E'\longrightarrow P\,E'\,h\,h'\,v}{\Gamma\rhd c.m(\bar a):P}\ \textsc{VinvS}$$

$$\frac{finite(D) \quad D\rhd e:P \quad G\subseteq D \quad provable(D,G)}{G\rhd e:P}\ \textsc{Vcut}$$

$$\frac{goodContext\ MST\ G \quad finite(G) \quad (c.m(y),MST\ m\ (y@[\mathsf{null}]))\in G}{\emptyset\rhd c.m(z):MST\ m\ (z@[\mathsf{null}])}\ \textsc{VadaptS}$$

The *goodContext* property requires that whenever a method invocation is associated to its specification table entry in G, the method body satisfies the specification for any arguments passed to the body via the formal parameters. See the slightly more general definition in [2].