Chapter 1

A Resource-aware Program Logic for a JVM-like Language

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

Abstract: We present a resource-aware program logic for a JVM-like language and prove its soundness and completeness. We first define Grail, an abstraction over a subset of the JVM bytecode language to facilitate formalisation while retaining a close correspondence to JVM's cost model. For Grail we then define an operational semantics, and on top of that a VDM-style program logic that additionally tracks resource consumption such as execution time and heap allocation. Finally, we prove soundness and completeness of this program logic, with respect to the operational semantics. All formalisations and proofs have been done in the Isabelle theorem prover.

1.1 INTRODUCTION

In the context of distributed systems, security issues are of great concern. For example a provider of computation resources might only make these resources available to programs that do not exceed certain limits on execution time or heap consumption. With the emergence of Grid technology, that aims to connect such providers on a global scale to provide transparent access to computational resources, such guarantees are increasingly sought after.

In our project we aim to use proof-carrying-code (PCC) technology [Nec97]

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

[2]Institut für Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany; {mhofmann,hwloidl}@informatik.uni-muenchen.de

to endow mobile code with proofs of bounded resource consumption. Thus, a service provider can easily check that a given resource policy is adhered to, and based on this rigorous proof allow execution of the code. The feasibility of this approach relies on the observation that, while it is difficult to produce a proof of certain program properties, it is far less time consuming to check this property. Furthermore, in our context we are interested in resource properties, rather than more general correctness properties, which are harder to verify.

In this paper we focus on the design and implementation of a resource-aware program logic for a slight abstraction of JVM bytecode. In Section 1.2 we define Grail [BMS03] as an abstraction over a subset of the JVM bytecode language. We model objects and dynamic method call, but no class hierarchies at the moment. In Section 1.3 we define a big-step, operational semantics for this first-order functional language. Because of syntactic restrictions on Grail it can also be read as an imperative language of assignments. The imperative reading of Grail is isomorphic to a subset of the Java Virtual Machine Language (JVML), which is why we sometimes call the logic for Grail a "bytecode" logic. Moreover, the resource usage notions which are formalised in our logic are related to a cost model for the imperative execution of Grail on a typical Java Virtual Machine [Ber02].

The infrastructure built in our project, requires that the logic is implemented and can be used to automatically check resource properties. Therefore we have used the theorem prover Isabelle/HOL right from the start of designing the logic. This enabled us to explore various design decisions for the program logic, and we summarise the most important ones in Section 1.5. The use of an automated theorem prover also helped to prove soundness and completeness for the program logic. Discussions of these core technical results are given in Sections 1.4.5 and 1.4.6. Since the program logic and its implementation is part of the trusted code base of the PCC infrastructure, it is essential for the overall security of the systems to have such results available.

1.2 GVM: GRAIL VIRTUAL MACHINE

This section defines the syntax of the language as formalised, the basic components of its semantics and the structure of cost components.

1.2.1 Syntax

We assume mutually disjoint sets of method names, class names, function names (i.e. labels of basic blocks), field names and variables, ranged over by m, c, f, fld, and x, respectively. The category V of values (ranged over by v) comprises integers i, references r and the special value $\perp$ (which stands for the absence of a value). References are either null (written null) or of the form $Ref\ l$ where l is a location (represented by a natural number).

Expressions

Conceptually, Grail expressions are partitioned into two sets.

- simple expressions are the basic unit of execution and directly compute values. They are built from operators, constants, and previously computed values (names), and correspond to primitive sequences of bytecode instructions. An expression may have a side effect, but there are no directly nested expressions. We model untyped expressions.
- compound expressions are built using let expressions. A function body is a let expression. A function ends with tail calls to other functions or methods.

Proofs in the theorem prover are somewhat simplified if we combine both types of expressions in a single phrase class:

$$\begin{aligned} e \in \mathit{expr} \quad ::= \quad & \texttt{Null} \mid \texttt{Int}\ i \mid \texttt{Var}\ x \mid \texttt{Primop}\ p\ x\ x \mid \\ & \texttt{New}\ c\ [(\mathit{fld}_1, x_1), \ldots, (\mathit{fld}_n, x_n)] \mid \texttt{GetF}\ x\ \mathit{fld} \mid \texttt{PutF}\ x\ \mathit{fld}\ x \mid \\ & x \cdot m(x) \mid c.m(x) \mid \texttt{Let}\ x\ e\ e \mid \texttt{Letv}\ e\ e \mid \texttt{If}\ x\ e\ e \mid \texttt{Call}\ f \end{aligned}$$

Here p ranges over primitive operations of type $V \Rightarrow V \Rightarrow V$ such as arithmetic operations and comparison operators. As a compromise between a completely untyped formalisation and the formalisation of a full type system, we encode some simple typing conditions in the syntax. For example, the binding $\texttt{Let}\ x\ e_1\ e_2$ is used if the evaluation of e_1 returns an integer or reference value on top of the JVM stack while $\texttt{Letv}\ e_1\ e_2$ represents purely sequential composition, used for example if e_1 is a PutF expression. Object creation (instruction New) includes initialisation of the object fields according to the argument lists: the content of variable x_i is stored in field fld_i. Function calls (Call) follow the Grail calling convention (i.e. correspond to immediate jumps) and do not carry arguments. The instructions $x \cdot m(x)$ and $c.m(x)$ represent virtual and static method invocation.

Methods and programs

For simplicity we restrict our attention to methods with a single formal argument, which is always called *param*. In virtual methods, the variable name *self* represents the pointer to the parent object. We assume that all method declarations employ distinct names for identifying inner basic blocks (Grail functions). A program may thus be represented by a table *FT* mapping function identifiers to expressions, and a table *MT* associating the initial basic block of each method to its parent class and method identifier.

1.2.2 Semantic components

The machine model which forms the basis of our semantics consists of a heap, a store for local variables and a class file environment.

Heap

The *heap* is a map from locations to objects. Conceptually, objects consist of a class name together with a mapping of field names to values. In our formalisation, we follow an approach originally due to Burstall which has recently been employed by [Nip02], where each object is treated as a separate "mini"-heap. The heap is split into two components: a total function from locations and field names to values, and a partial function from locations to class names. This object lookup heap ("*oheap*") is used to determine the runtime class of objects during virtual method invocation, but its domain also indicates the size of the heap, which we wish to reason about in the program logics. The rules of the dynamic semantics guarantee that field access operations involve only objects which are located in the current domain of the heap.

Environments

Variables which are local to a method invocation are kept in an environment $E \in$ *env* which maps variables to values. We use $E\langle x \rangle$ to denote the lookup operation and $E\langle x := v \rangle$ to denote an update. Similar to the heap the store is represented as a total function, with a silent assumption that well-defined method bodies only access variables which have previously been assigned a value.

Class file environment

The class file environment is modelled in the context of our formalisation, by abstract total functions *MT* and *FT*, which map class names $\times$ method names and function names to expressions, respectively.

1.2.3 Resource components

Resource consumption is modelled in the operational semantics by resource tuples

$$p = \langle clock_p\ callc_p\ invkc_p\ invkdpth_p \rangle.$$

The four components range over $\mathbb{N}$ and represent the following costs:

- the *clock* represents an abstract instruction counter. In the operational semantics, each Grail instruction is associated the number of clock-ticks corresponding to the number of JVM instructions to which it expands.
- as an example for how one might refine the abstract instruction counter, *callc* counts the number of function calls (i.e. jump instructions). In combination with *invkc* it may also be used to formally verify properties of Grail-level optimisations such as the replacement of method (tail) recursion by function recursion for static methods.
- a second refinement of the instruction count, *invkc*, monitors the number of method invocations. It would be easy to extend this to consider separate counts for each method identifier, class identifier or method declaration.

- *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.

The following two operators combine resource tuples:

$$\begin{aligned} p \oplus q &= \langle (clock_p + clock_q)\ (callc_p + callc_q)\ (invkc_p + invkc_q)\ (invkdpth_p + invkdpth_q) \rangle \\ q \smile q &= \langle (clock_p + clock_q)\ (callc_p + callc_q)\ (invkc_p + invkc_q)\ (max(invkdpth_p)(invkdpth_q)) \rangle \end{aligned}$$

1.3 OPERATIONAL SEMANTICS

This section describes the operational semantics to which the program logic refers. The semantics is a big-step evaluation relation based on the functional interpretation of Grail. Indeed, a big-step semantics suffices for a program logic which (like ours) concentrates on partial correctness and is presented in VDM style.

1.3.1 Rules

The operational semantics is based on the functional view of Grail, with judgements of the form

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

Such a statement is to be read as "in variable environment E and starting with a heap h, code e evaluates to the value v, yielding the heap h' and consuming p resources." The rules are as follows

$$\frac{}{E \vdash h, \texttt{Null} \Downarrow (h, \mathsf{null}, \langle 1\,0\,0\,0 \rangle)} \quad (\textsc{null})$$

$$\frac{}{E \vdash h, \texttt{Int}\ i \Downarrow (h, i, \langle 1\,0\,0\,0 \rangle)} \quad (\textsc{int})$$

$$\frac{}{E \vdash h, \texttt{Var}\ x \Downarrow (h, E\langle x \rangle, \langle 1\,0\,0\,0 \rangle)} \quad (\textsc{var})$$

$$\frac{}{E \vdash h, \texttt{Primop}\ p\ x\ y \Downarrow (h, p\ (E\langle x \rangle)\ (E\langle y \rangle), \langle 3\,0\,0\,0 \rangle)} \quad (\textsc{primop})$$

$$\frac{E\langle x \rangle = Ref\ l}{E \vdash h, \texttt{GetF}\ x\ fld \Downarrow (h, h(l).fld, \langle 2\,0\,0\,0 \rangle)} \quad (\textsc{getf})$$

$$\frac{E\langle x \rangle = Ref\ l}{E \vdash h, \texttt{PutF}\ x\ fld\ y \Downarrow (h[l.fld \mapsto E\langle y \rangle], \bot, \langle 3\,0\,0\,0 \rangle)} \quad (\textsc{putf})$$

$$\frac{freshloc(l, h)}{E \vdash h, \texttt{New}\ c\ fldvals \Downarrow (newObj\ h\ l\ E\ c\ fldvals, Ref\ l, \langle 1\,0\,0\,0 \rangle)} \quad (\textsc{new})$$

$$\frac{E\langle x \rangle = true \quad E \vdash h, e_1 \Downarrow (h_1, v, p)}{E \vdash h, \texttt{If}\ x\ e_1\ e_2 \Downarrow (h_1, v, \langle 2\,0\,0\,0 \rangle \smile p)} \quad (\textsc{iftrue})$$

$$\frac{E\langle x \rangle = false \quad E \vdash h, e_2 \Downarrow (h_1, v, p)}{E \vdash h, \texttt{If}\ x\ e_1\ e_2 \Downarrow (h_1, v, \langle 2\,0\,0\,0 \rangle \smile p)} \quad (\textsc{iffalse})$$

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

$$\frac{E \vdash h, e_1 \Downarrow (h_1, \bot, p) \quad E \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, \texttt{Letv}\ e_1\ e_2 \Downarrow (h_2, v, p \smile q)} \quad (\textsc{letv})$$

$$\frac{E \vdash h, FT\ f \Downarrow (h_1, v, p)}{E \vdash h, \texttt{Call}\ f \Downarrow (h_1, v, \langle 1\,1\,0\,0\rangle \smile p)} \quad (\textsc{call})$$

$$\frac{[self := \mathsf{null}, param := E\langle y\rangle] \vdash h, MT\ c\ m \Downarrow (h_1, v, p)}{E \vdash h, c.m(y) \Downarrow (h_1, v, \langle 3\,0\,1\,1\rangle \oplus p)} \quad (\textsc{invokestatic})$$

$$\frac{\begin{array}{cc} E\langle x\rangle = Ref\ l & h.oheap(l) = Some(c) \\ [self := Ref\ l, param := E\langle y\rangle] \vdash h, MT\ c\ m \Downarrow (h_1, v, p) & \end{array}}{E \vdash h, x \cdot m(y) \Downarrow (h_1, v, \langle 5\,0\,1\,1\rangle \oplus p)} \quad (\textsc{invoke})$$

1.3.2 Discussion of rules

In rule GETF, the notation $h(l).fld$ represents the value of field fld in the object at heap location l, while in rule PUTF the notation $h[l.fld \mapsto v]$ denotes the corresponding update operation. In rule NEW, the condition $freshloc(l,h)$ expresses the fact that l is a location outside the domain of h, while $newObj\ h\ l\ E\ c\ [(fld_1, x_1), \ldots, (fld_n, x_n)]$ represents the heap which agrees with h on all locations different from l and maps l to an object of class c, with the field entries $fld_i := E(x_i)$. We silently assume a static semantics which ensures well-typedness of object creation and initialisation (including class membership), field access operations and method invocations.

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 the rules for conditionals, we charge for pushing the value $E\langle x\rangle$ onto the stack, with an additional tick charged for evaluating the branch condition and performing the appropriate jump. The difference in the costs between LET and LETV arises from the fact that the latter one is purely sequential composition. The implicit typing convention ensures that the check $w \neq \bot$ always succeeds in LET, and that the value returned by e_1 in LETV is indeed $\bot$. Both rules combine the component costs p and q using the operator $\smile$: method invocations in e_1 and e_2 are not nested inside each other, hence the maximal invocation depth is the maximum of the individual invocation depths[3]. In rule CALL, the Grail convention that functions calls amount to immediate jumps explains why the call merely invokes the execution of the function body of f. We charge for one anonymous instruction, and also explicitly for the execution of a jump. In rule INVOKESTATIC, the body of method $c.m$ is executed in an environment which represents a fresh frame - the essentially only binding is that of the

[3]The usage of $\smile$ in rules IFTRUE, IFFALSE and CALL is arbitrary - using $\oplus$ would give the same result as the invocation depth component of the increment is 0.

(standard) method parameter to $E\langle y\rangle$. The instruction counter is incremented by 3, for evaluating $E\langle y\rangle$ and for pushing and popping the frame. 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, rule INVOKE first evaluates the object pointer $E\langle x\rangle$ and uses the resulting location to retrieve the object's dynamic class from the heap. 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 value $E\langle y\rangle$ in variable *param*. The costs charged arise by again considering the evaluation of $E\langle x\rangle$ and $E\langle y\rangle$ 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.

1.4 A PROGRAM LOGIC FOR GRAIL

1.4.1 Style of the Program Logic: VDM vs Hoare

In developing a program logic, we consider two different styles: VDM-style [Jon90] and Hoare-style [Hoa69]. The more commonly used Hoare-style is based on triples of the form $\{P\}\ e\ \{Q\}$ stating that if the assertion P is valid before executing the expression e, then the assertion Q is valid after execution. In order to capture intermediate values in the execution of a program, auxiliary variables are used. These variables have to be universally quantified in the formal definition of validity. For example the specification of the exponential function, returning its result in variable v, can be written with auxiliary variable X and Y as follows $\{0 \leq y \ \wedge\ x = X \ \wedge\ y = Y\}\ exp(x,y)\ \{v = X^Y\}$. In contrast, a VDM-style logic uses tuples of the form $e\ :\ Q$ stating that an assertion Q is valid for expression e, where Q can refer to variables in both pre- and post-state of the execution of e. Variables in the pre-state are often written as "hooked" variables, e.g. $\grave{x}$, as in the specification of an exponential function $exp(x,y)\ : \{0 \leq \grave{y} \implies v = \grave{x}^{\grave{y}}\}$.

The main advantage of a VDM-style program logic is the absence of auxiliary variables in the assertions, which are used in a Hoare-style to propagate values from the pre- to the post-condition. This requires a rather intricate rule of adaptation in a Hoare-style logic, which is elaborated by Kleymann [Kle99] and used by Oheimb [von01] in a program logic for a Java subset. To avoid such complications we prefer a *VDM style*, which gives direct access to both the pre- and the post-state of the computation.

1.4.2 Type Definitions

In modelling assertions we use a *shallow embedding*, which defines an assertion as a predicate over the state of the computation. In contrast, a deep embedding would define assertions as a separate data type, and has the advantage of being easier to manipulate, but the disadvantage of being less flexible.

In our setting, a VDM-style assertion (specification) is a set, usually written as set comprehension with the predicate inside, which can refer to the pre- and the post-state of a program expression, as well as the resources consumed.

A GVM state consists of a store (environment) of local variables (of type *env*) and the heap (of type *heap*). As can be seen from the operational semantics, only the heap is modified in the evaluation of an expression, returning a value (of type *val*) and a resource tuple (of type *rcount*). Thus, the overall type of a VDM-style assertion is:

$$vdmassertion = env \times heap \times heap \times val \times rcount$$

With this type, the informal statement "assertion P is fulfilled in pre-state (E, h), post-state (E, hh) with result value v and resource consumption p" is written formally as set membership, i.e. $(E,h,hh,v,p) \in P$. Similarly, a program expression e satisfies an assertion, written as $e : P$, iff every (terminating) execution of e is allowed in P, i.e. $\forall E\ h\ hh\ v\ p.\quad E \vdash h, e \Downarrow (hh,v,p) \implies (E,h,hh,v,p) \in P$. The requirement that this holds for every derivable statement $e : P$ is the soundness criterion for our logic, considered further in Section 1.4.5.

For example, suppose for the expression `Let` n (`GetF` x *count*) `Var` n, we would like to specify that the result value is the value of the *count* field of object x in the heap, which has to be an instance of class *Foo*. This is written using the set comprehension:

$$\{(E,h,hh,v,p) \mid E\langle x\rangle = Ref\ l \wedge h.oheap(l) = Some\ Foo \implies v = h(l).count\}$$

Sometimes we will want to prove statements under assumptions. The *VDM context* has the role of storing assumptions that program expressions meet specifications. It has the following type:

$$vdmcontext = \{expr \times vdmassertion\}$$

The rules for CALL, INVOKE and INVOKESTATIC are the only ones which extend the context of assumptions.

1.4.3 Program Logic

VDM Rules

The judgement of the program logic

$$G \rhd e : P$$

is read as "under the assumptions G, the Grail code e satisfies the specification P," where G is of type *vdmcontext*, e is of type *expr* and P of type *vdmassertion*. This judgement is defined inductively by the rules given below. First, there are two structural rules, and then one rule for each form of program expression. The rules in fact derive the strongest specification for each expression (this is proven formally in the completeness proof, in Section 1.4.6).

$$\frac{(e,P) \in G}{G \rhd e : P}\ (\text{VAX}) \qquad \frac{G \rhd e : P \quad P \subseteq Q}{G \rhd e : Q}\ (\text{VCONSEQ})$$

$$\frac{}{G \rhd \texttt{Null} : \{(E,h,hh,v,p) \mid hh = h \wedge v = \mathsf{null} \wedge p = \langle 1\,0\,0\,0\rangle\}} \quad \text{(VNULL)}$$

$$\frac{}{G \rhd \texttt{Int}\ i : \{(E,h,hh,v,p) \mid hh = h \wedge v = i \wedge p = \langle 1\,0\,0\,0\rangle\}} \quad \text{(VINT)}$$

$$\frac{}{G \rhd \texttt{Var}\ x : \{(E,h,hh,v,p) \mid hh = h \wedge v = E\langle x\rangle \wedge p = \langle 1\,0\,0\,0\rangle\}} \quad \text{(VVAR)}$$

$$\frac{}{G \rhd \texttt{Primop}\ p\ x\ y : \{(E,h,hh,v,p) \mid \begin{array}{l} v = p\,E\langle x\rangle\,E\langle y\rangle \wedge \\ \quad hh = h \wedge p = \langle 3\,0\,0\,0\rangle\} \end{array}} \quad \text{(VPRIM)}$$

$$\frac{}{G \rhd \texttt{GetF}\ x\ fld : \{(E,h,hh,v,p) \mid \begin{array}{l} \exists l.E\langle x\rangle = Ref\ l \wedge hh = h \wedge \\ \quad v = hh(l).fld \wedge p = \langle 2\,0\,0\,0\rangle\} \end{array}} \quad \text{(VGETF)}$$

$$\frac{}{G \rhd \texttt{PutF}\ x\ fld\ y : \{(E,h,hh,v,p) \mid \exists l.\ \begin{array}{l} E\langle x\rangle = Ref\ l \wedge p = \langle 3\,0\,0\,0\rangle \wedge \\ \quad hh = h[l.fld \mapsto E\langle y\rangle] \wedge v = \bot\} \end{array}} \quad \text{(VPUTF)}$$

$$\frac{}{G \rhd \texttt{New}\ c\ fldvals : \{(E,h,hh,v,p) \mid \exists l.\ \begin{array}{l} l \notin dom\ h \wedge v = Ref\ l \wedge \\ \quad hh = newObj\ h\ l\ E\ c\ fldvals \wedge p = \langle 1\,0\,0\,0\rangle\} \end{array}} \quad \text{(VNEW)}$$

$$\frac{G \rhd e_1 : P_1 \quad G \rhd e_2 : P_2}{G \rhd \texttt{If}\ x\ e_1\ e_2 : \{(E,h,hh,v,p) \mid \exists pp.\ \begin{array}{l} p = pp \oplus \langle 2\,0\,0\,0\rangle \wedge \\ (E\langle x\rangle = true \Longrightarrow (E,h,hh,v,pp) \in P_1) \wedge \\ (E\langle x\rangle = false \Longrightarrow (E,h,hh,v,pp) \in P_2) \wedge \\ (E\langle x\rangle = true \vee E\langle x\rangle = false)\} \end{array}} \quad \text{(VIF)}$$

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

$$\frac{G \rhd e_1 : P_1 \quad G \rhd e_2 : P_2}{G \rhd \texttt{Letv}\ e_1\ e_2 : \{(E,h,hh,v,p) \mid \exists\ p_1\,p_2\,h_1.\ \begin{array}{l} (E,h,h_1,\bot,p_1) \in P_1 \wedge \\ (E,h_1,hh,v,p_2) \in P_2 \wedge \\ p = p_1 \smile p_2\} \end{array}} \quad \text{(VLETV)}$$

$$\frac{G \cup \{(\texttt{Call}\ f,P)\} \rhd (FT\ f) : \{(E,h,hh,v,p) \mid (E,h,hh,v,\langle 1\,1\,0\,0\rangle \oplus p) \in P\}}{G \rhd \texttt{Call}\ f : P} \quad \text{(VCALL)}$$

$$\frac{\forall E'.\quad \begin{array}{l} G \cup \{(c.m(y),P)\} \rhd \\ MT\ c\ m : \{(E,h,hh,v,p) \mid \begin{array}{l} E = [self := \mathsf{null}, param := E'\langle x\rangle] \\ \quad \Longrightarrow (E',h,hh,v,\langle 3\,0\,1\,1\rangle \oplus p) \in P\} \end{array} \end{array}}{G \rhd c.m(y) : P} \quad \text{(VINVOKESTATIC)}$$

$$\frac{\begin{array}{l} \forall E'\,h'\,l\,c.\quad E'\langle x\rangle = Ref\ l \wedge h'.oheap(l) = Some(c) \Longrightarrow \\ \quad G \cup \{x \cdot m(y),P)\} \rhd \\ \quad MT\ c\ m : \{(E,h,hh,v,p) \mid \begin{array}{l} E = [self := E'\langle x\rangle, param := E'\langle y\rangle] \wedge h = h' \\ \quad \Longrightarrow (E',h,hh,v,\langle 5\,0\,1\,1\rangle \oplus p) \in P\} \end{array} \end{array}}{G \rhd x \cdot m(y) : P} \quad \text{(VINVOKE)}$$

Discussion of the Rules

The axiom rule VAX allows one to use specifications found in the context. The VCONSEQ consequence rule allows one to derive an assertion Q that follows from another derivable assertion P. Because of our encoding of assertions as sets, implication on assertions is written as subset inclusion.

The leaf rules (VNULL to VNEW) directly model the corresponding rules in the operational semantics, with constants for the resource tuples. The only rules modifying the heap are VPUTF and VNEW. The former adds a new mapping to the heap, the latter allocates a new object using the auxiliary function *newObj*. The VIF rule uses the appropriate assertion based on the boolean value in the variable x. Since the header of the conditional is just a variable, the heap cannot be modified by this step and therefore only existential quantification over the resource tuple pp is needed. In the VLET rule, however, existential quantification over the result value w and result heap h_1 of evaluating the let header is needed, as well as a quantification over the resource tuples from let header and body. Combining the resource tuples on top level, rather than forwarding the consumed resources from header to body, has the advantage of minimising dependencies between the states in the execution. This, together with the separation of the components of the state, facilitates the use of the simplification machinery provided by Isabelle.

The rules for recursive functions and methods are the most interesting, involving the context G. The VCALL rule is similar to Hoare's original rule for parameter-less recursive procedures. In a backward reasoning style it requires to prove the assertion under consideration, for the body of the function, under the additional assumption that this assertion holds for the entire function, which is captured in the context. Additionally, we have to modify the resource tuple when analysing the body. We use a global table FT to map function names to their corresponding bodies.

The VINVOKESTATIC and VINVOKE rules have the same overall structure. The former can directly extract the method body from the class file by accessing the global table MT, indexed with class name and method name. The latter has to look up the class name in the heap. Therefore, an additional quantification over all possible class names c is required. For both rules the environment, containing the special variables *self* and *param*, is initialised appropriately.

1.4.4 Additional Admissible Rules of the Logic

Additionally to the above rules, we may use further rules in the verification.

Context weakening

The weakening rule

$$\frac{G \rhd e : P}{G \cup D \rhd e : P} \qquad \text{(WEAK)}$$

is proven by induction on the derivations of $G \rhd e : P$.

Cut rules

These rules allow to substitute derivations for context assumptions. Rule

$$\frac{\{(ee,P)\}\cup D\rhd e:Q \qquad G\rhd ee:P \qquad G\subseteq D}{D\rhd e:Q} \qquad \text{(CCUT)}$$

is proven by induction on derivations of $\{(ee,P)\}\cup D\rhd e:Q$, rule

$$\frac{G\rhd e:P \quad D\subseteq G \quad \mathit{CallInvContext}(G) \quad D\,\mathit{proves}\,G}{D\rhd e:P} \qquad \text{(CUT2)}$$

by induction on derivations of $G\rhd e:P$. The premises of CUT2 are defined by

$$\begin{array}{l}\mathit{CallInvContext}(G) = \\ \forall e\,Q.(e,Q)\in G \longrightarrow ((\exists f.\ e=\texttt{Call}\,f)\vee(\exists x\,m\,y.\ e=x\cdot m(y))\vee(\exists c\,m\,y.\ e=c.m(y)))\end{array}$$

and

$$\begin{array}{rcl} D\,\mathit{proves}\,G & = & ((\forall f\,Q.\ (\texttt{Call}\ f,Q)\in G\longrightarrow D\rhd \texttt{Call}\ f:Q)\wedge \\ & & (\forall x\,m\,y\,Q.\ (x\cdot m(y),Q)\in G\longrightarrow D\rhd x\cdot m(y):Q)\wedge \\ & & (\forall c\,m\,y\,Q.\ (c.m(y),Q)\in G\longrightarrow D\rhd c.m(y):Q))\end{array}$$

Mutual recursion

Unlike Nipkow's mechanism for mutually recursive procedures [Nip02], we do not define a separate derivation system for judgements with sets of VDM assertions on both sides, but instead give a single rule for mutual recursion. We assume the existence of a specification table *Spec* which maps each function identifier to a VDM assertion, and another similar table *MSpec* associating assertions to pairs of class names and method names.

A VDM context G is called *consistent* with the specification tables if

- $(\texttt{Call}\,f,\ P)\in G$ implies

  $$P=\mathit{Spec}\ f\ \wedge\ G\rhd \mathit{FT}\ f:\{(E,h,hh,v,p)\mid(E,h,hh,v,\langle 1\,1\,0\,0\rangle\frown p)\in P\},$$

- $(c.m(y),P)\in G$ implies

  $$\begin{array}{l}P=\mathit{MSpec}\ c\ m\wedge \\ \forall E'.\ G\rhd \mathit{MT}\ c\ m:\{(E,h,hh,v,p).\ E=[\mathit{self}:=\mathsf{null},\mathit{param}:=E'\langle y\rangle] \\ \qquad\qquad\qquad\qquad\longrightarrow(E',h,hh,v,\langle 3\,0\,1\,1\rangle\oplus p)\in P\},\end{array}$$

- and $(x\cdot m(y),P)\in G$ implies

  $$\begin{array}{l}(\exists E'\,h'\,l\,c.\ E'\langle x\rangle=\mathit{Ref}\ l\wedge h'.\mathit{oheap}(l)=c\wedge P=\mathit{MSpec}\,c\,m)\ \wedge \\ (\forall E'\,h'\,l\,c.\ \ (E'\langle x\rangle=\mathit{Ref}\ l\wedge h'.\mathit{oheap}(l)=c)\longrightarrow \\ \qquad G\rhd \mathit{MT}\ c\ m:\{(E,h,hh,v,p).\ E=[\mathit{self}:=E'\langle x\rangle,\mathit{param}:=E'\langle y\rangle]\wedge h=h' \\ \qquad\qquad\qquad\qquad\longrightarrow(E',h,hh,v,\langle 5\,0\,1\,1\rangle\oplus p)\in P\}).\end{array}$$

The rule for mutually recursive function calls or method invocations

$$\frac{G\,finite \quad CallInvContext(G) \quad G\,consistent \quad (e,P) \in G}{\emptyset \rhd e : P} \text{ (MUTREC)}$$

is proven by induction on the size of G, using CCUT and the following lemma.

Lemma 1.1. *If G is consistent, $(e,Q) \in G$ and $CallInvContext(G)$ holds then $G - (e,Q)$ is consistent.*

The restriction to finite contexts rule MUTREC is fulfilled for any practical program finitely many function symbols, and method invocations.

For illustration purposes we give the specialised rule for two (possibly mutually recursive) functions:

$$\frac{\begin{array}{c} i \in \{1,2\} \\ G = \{(\texttt{Call}\ f_1, P_1), (\texttt{Call}\ f_2, P_2)\} \\ G \rhd FT\ f_1 : \{(E,h,hh,v,p).(E,h,hh,v,\langle 1\,1\,0\,0\rangle \frown p) \in P_1\} \\ G \rhd FT\ f_2 : \{(E,h,hh,v,p).(E,h,hh,v,\langle 1\,1\,0\,0\rangle \frown p) \in P_2\} \end{array}}{\emptyset \rhd \texttt{Call}\ f_i : P_i} \text{ (FUNREC2)}$$

Notice that the rules in particular allow one to derive statements about function calls and method invocations in the empty context.

1.4.5 Soundness

In this section we prove soundness for the program logic in Section 1.4.3.

Validity

We first have to define the *validity* of an assertion for a given program expression, in a given context. Using standard techniques to prove soundness of function calls and method invocations, we additionally parameterise the notion of validity by a natural number acting as step counter in the evaluation. Note, that this counter will only be used to do induction over when we prove soundness. It is not used for capturing resources, since we need a more detailed cost metric for the latter, and use the resource tuples for that.

Definition 1.1. *(Validity) Let Q be in vdmassertion and e in expr e. Q is valid for e iff*

$$\models_n e : Q \equiv (\forall m.\, m \leq n \Longrightarrow (\forall E\, h\, hh\, v\, p.\, E \vdash h, e \Downarrow_m (hh, v, p) \Longrightarrow (E,h,hh,v,p) \in Q))$$

This definition uses a variant of the operational semantics, that adds the same counter but is otherwise equivalent to the semantics in Section 1.3.

This definition is related to the counter-less definition of validity in Section 1.4.2 like this

Lemma 1.2. *Let e be in expr, Q in vdmassertion and n in* $\mathbb{N}$*, then*

$$\forall n.\ \models_n e : Q \implies \models e : Q$$

Proof. By unfolding definitions of validity and using the operational semantics.

Note that the counter n restricts the set of pre- and post-states for which Q has to be fulfilled, i.e. that have to be contained in Q. Because of this negative occurrence of the counter in the validity formula, we have the following lemma, allowing us to weaken this counter.

Lemma 1.3. *Let e be in expr, Q in vdmassertion and* m, n *in* $\mathbb{N}$*, then*

$$m < n \wedge \models_n e : Q \implies \models_m e : Q$$

Proof. By unfolding the definition of validity and simplification.

The validity of an entire context is defined as the conjunction over the validity of its components.

Definition 1.2. *Let G be in vdmcontext, e in expr and Q in vdmassertion. The context G is valid iff*

$$\models G \ \equiv\ \forall (e, Q) \in G.\ \models e : Q$$

We generalise the above notion of validity, to one with a context as follows.

Definition 1.3. *Let G be in vdmcontext, e in expr and Q in vdmassertion. The assertion Q is valid for e in context G iff*

$$G \models e : Q \ \equiv\ \models G \implies \models e : Q$$

Again, for both notions we also have definitions that are relativised over the counter used in the definition of validity. We omit these obvious variants of the definitions above.

Soundness Theorem

The main soundness theorem states that validity follows from derivability in an empty context and is formalised as follows.

Theorem 1.1. *(Soundness) Let e be in expr and P in vdmassertion, then*

$$\triangleright\, e : P \implies \models e : P$$

Proof. This theorem follows directly from Lemma 1.4 by instantiating G with $\emptyset$.

Lemma 1.4. *Let G be in vdmcontext, e in expr and P in vdmassertion, then*

$$G \triangleright e : P \implies G \models e : P$$

Proof. This theorem follows directly from Lemma 1.5 by unfolding the definition of relativised validity and simplifying.

The following stronger lemma expresses the same soundness property over contextual, relativised validity. It is the main theorem left to prove.

Lemma 1.5. *Let G be in vdmcontext, e in expr and P in vdmassertion, then*

$$G \rhd e : P \implies \forall n.\ G \models_n e : P$$

Proof. The proof proceeds by structural induction over the program expression *e*. The main steps in the individual cases are as follows:

- All leaf cases follow directly from unfolding the definition of validity and applying the rules of the operational semantics.
- The `If` case additionally applies a variant of *InductLemma* in both the true and the false case.
- The `let` case additionally applies a variant of *InductLemma* and basic arithmetic over naturals and max.
- The `call` case uses induction over the counter of the relativised validity. In the induction step it uses *InductLemma* and the rules of the operational semantics, as well as some auxiliary lemmas on contexts.
- The invoke cases also use induction over the counter of the relativised validity, together with a modified *InductLemma* and the same auxiliary lemmas.

This proof relies on one more lemma for performing the induction.

Lemma 1.6. *(InductLemma)*

$$\begin{array}{l}\forall e\ P\ G.\ (e,P) \in G \implies \\ \quad \forall n.\ \models_n G \implies \\ \quad\quad \forall m.\ m \leq n \implies \\ \quad\quad\quad \forall E\ h\ hh\ v\ p.\ E \vdash h, e \Downarrow m(hh,v,p) \implies (E,h,hh,v,p) \in P\end{array}$$

Proof. This lemma is proven by unfolding the definition of relativised and context validity, followed by simplification.

1.4.6 Completeness

The VDM-style program logic may be proven complete relative to the ambient logic using *strongest specifications*, similar to most general triples in Hoare-style verification

Definition 1.4. *Let e be in expr. The* strongest specification *of e is*

$$SSpec(e) = \{(E,h,hh,v,p).E \vdash h, e \Downarrow (hh,v,p)\}.$$

It is not difficult to prove that strongest specifications are valid

Lemma 1.7. $\models e : SSpec(e)$

and are stronger than (i.e. contained by) any other valid specification

Lemma 1.8. $\models e : P \Longrightarrow SSpec(e) \subseteq P$.

The overall proof idea of completeness is that of [Hof98] and [Nip02]: we first prove a lemma,

Lemma 1.9.

$$\begin{array}{rcl} \forall f.\ G \rhd \texttt{Call}\ f : SSpec(\texttt{Call}\ f) & & \\ \wedge\ \forall c\, m\, y.\ G \rhd c.m(y) : SSpec(c.m(y)) & \Longrightarrow & G \rhd e : SSpec(e) \\ \wedge\ \forall x\, m\, y.\ G \rhd x \cdot m(y) : SSpec(x \cdot m(y)) & & \end{array}$$

which allows one to relate *any* expression e to its strongest specification $SSpec(e)$ in a context G, provided that G in turn relates each function or method call to its strongest specification. The proof of this lemma proceeds by induction on the structure of e.

Next, we define a specific context, *StrongG*, which contains exactly the strongest specifications for all function calls and method invocations.

Definition 1.5.

$$\begin{array}{rl} StrongG = & \{(e,P) \mid \exists f.\ e = \texttt{Call}\ f\ \wedge\ P = SSpec(e)\} \cup \\ & \{(e,P) \mid \exists c\, m\, y.\ e = (c.m(y)\ \wedge\ P = SSpec(e)\} \cup \\ & \{(e,P) \mid \exists x\, m\, y.\ e = (x \cdot m(y)\ \wedge\ P = SSpec(e)\} \end{array}$$

We also define a predicate which is fulfilled if for all entries in G the entries in the specification tables contain the strongest specifications.

Definition 1.6.

$$\begin{array}{l} StrongTables\ G = \\ \forall\, e\, P.\ (e,P) \in G \longrightarrow \\ \quad ((\forall f.\ e = \texttt{Call}\ f \longrightarrow Spec\ f = SSpec(e)) \wedge \\ \quad\ (\forall\, c\, m\, y.\ e = c.m(y) \longrightarrow MSpec\ c\ m = SSpec(e)) \wedge \\ \quad\ (\forall\, x\, m\, y.\ e = x \cdot m(y) \longrightarrow (\exists\, E\, h\, l\, c.\ \ E\langle x \rangle = Ref\ l \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h.oheap(l) = Some(c) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad MSpec\ c\ m = SSpec(e)))) \end{array}$$

Indeed, *StrongG* is *consistent* if all the specification tables contain the strongest specifications:

Lemma 1.10. $StrongTables\ StrongG \Longrightarrow StrongG\, consistent$.

On the other hand, combining rules CUT2 and MUTREC with Lemma 1.9 yields

Lemma 1.11. $(StrongG\, consistent\ \wedge\ StrongG\, finite) \Longrightarrow \emptyset \rhd e : SSpec(e)$.

Consequently, completeness

Theorem 1.2. *(VDMcomplete) Let e be in expr and P in vdmassertion, then*

$$(StrongTables\ StrongG\ \wedge\ StrongG\, finite) \Longrightarrow \models e : P \Longrightarrow \emptyset \rhd e : P$$

follows by combining Lemmas 1.8, 1.10 and 1.11 and rule VCONSEQ.

1.5 CONCLUSIONS

This paper presented a resource-aware program logic for Grail, together with proofs of soundness and completeness. Although Grail makes several restrictions on the structure of the code, it is powerful enough to express general recursion, and we have developed derived rules to work with mutually recursive programs. The logic also covers dynamic method invocation, although most of our example programs are static. The entire logic is encoded in the Isabelle theorem prover and has been tested on non-trivial example programs such as in-place list reversal, to prove concrete resource bounds on space and time.

Several basic design decisions in developing this program logic are worth noting. Firstly, we prefer a VDM-style logic over a Hoare-style logic, since the former avoids complicated rules of adaptation over auxiliary variables, which are used in Hoare-style logics to propagate intermediate results from pre- to post-assertion. We make heavy use of the additional flexibility of formulating assertions, provided by a shallow embedding of the assertion language into the theorem prover. For us this outweighs the disadvantage of tying the logic to one particular prover. For the time being we restrict our infrastructure to a setup where the same theorem prover is used in the proof generation and the proof checking phase.

REFERENCES

[Ber02] L. Beringer. Cost Model. Deliverable d1b, LFCS, Edinburgh University, September 2002.

[BMS03] L. Beringer, K. MacKenzie, and I. Stark. Grail: a Functional Form for Imperative Mobile Code. In *Electronic Notes in Theoretical Computer Science*, volume 85, 2003.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hof98] M. Hofmann. Semantik und Verifikation. Lecture Notes, WS 97/98 1998.

[Jon90] C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.

[Kle99] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, 1999.

[Nec97] G. Necula. Proof-carrying Code. In *POPL'97 — Symposium on Principles of Programming Languages*, 1997.

[Nip02] T. Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *Computer Science Logic (CSL 2002)*, LNCS 2471, pages 103–119, 2002.

[von01] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.