Mobile Resource Guarantees and Policies

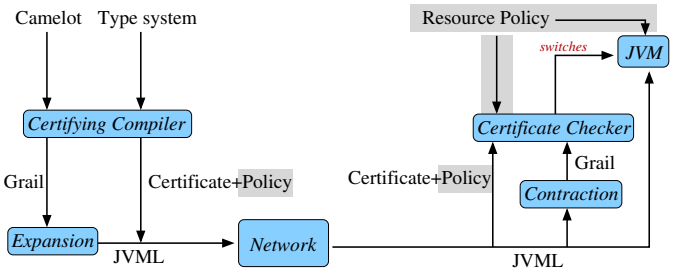
David Aspinall and Kenneth MacKenzie

LFCS, School of Informatics, The University of Edinburgh, U.K.

Abstract. This paper introduces notions of *resource policy* for mobile code to be run on smart devices, to integrate with the proof-carrying code architecture of the *Mobile Resource Guarantees* (MRG) project. Two forms of policy are used: *guaranteed* policies which come with proofs and *target* policies which describe limits of the device. A guaranteed policy is expressed as a function of a methods input sizes, which determines a bound on consumption of some resource. A target policy is defined by a constant bound and input constraints for a method. A recipient of mobile code chooses whether to run methods by comparing between a guaranteed policy and the target policy. Since delivered code may use methods implemented on the target machine, guaranteed policies may also be provided by the platform; they appear symbolically as assumptions in delivered proofs. Guaranteed policies entail proof obligations that must be established from the proof certificate. Before proof, a policy checker ensures that the guaranteed policy refines the target policy; our policy format ensures that this step is tractable and does not require proof. Delivering policies thus mediates between arbitrary target requirements and the desirability to package code and certificate only once.

1 Introduction

The *Mobile Resource Guarantees* project has built a proof-carrying code (PCC) infrastructure for ensuring resource bounds on mobile code (for an overview, see [AGH+05]). The infrastructure uses a certifying compiler from a high-level functional language called *Camelot* to a low-level language *Grail*, which is a functional presentation of a sub-language of the Java Virtual Machine Language (JVML). Thus, Grail programs are executed on a JVM but transmitted as standard class files, packaged together with PCC certificates. The architecture (with our extension) is shown below:



This is a fairly usual picture for proof-carrying code, except that we highlight the role of a *guaranteed resource policy* which is delivered as part of the certificate and a *target resource policy* which is the instance of the safety policy for code to meet resource usage restrictions imposed by the target machine.

The guaranteed resource policy is a specification, ultimately generated by the certifying compiler. It contains concrete bounds on the resource usage of the compiled Camelot program, in a standard format; it is guaranteed because it comes with a proof. The idea of the standard format is to allow mediation with an arbitrary target policy. In a general setting where the code delivery to a smart device takes place off-line (i.e. without communication back to the code producer), the recipient cannot communicate its target policy; it is therefore unrealistic to hope that the delivered code comes with a certificate stating exactly the required behaviour.

The certificate checker has responsibility (1) to check that the delivered policy would meet or exceed the target policy, and then (2) to check that the code indeed meets its guaranteed policy. Our design is to use the proof evidence in the certificate to establish (2), but allow the target certificate checker to use its own mechanism to establish (1), ideally as an efficient operation not involving proof checking or running a VCG. In more advanced scenarios, the certificate checker might use claimed policies to select between several possible implementations of a method supplied, for example, selecting the more favourable alternative in a time-space trade-off according to local conditions.

The target resource policy is an input both to the certificate checker and to the modified JVM. Usually in a PCC scenario, the safety check is entirely static. The certificate checker immediately denies execution to code which does not satisfy the target resource policy, switching off execution in the JVM. But the resource policy is also shown in the diagram as an input to the modified JVM: this is to allow, in principle, the possible *run-time monitoring* of resource usage to check conformance with the policy. A checker may decide to defer some resource bounds to dynamic checks if they cannot be ensured by the delivered policy, or, indeed, if the delivered proof certificate lacks static evidence that a particular claimed policy is met.

Contributions. Until now in our work on MRG and the closely related work, resource policies have not been considered explicitly. In MRG, we have used a fixed type system technology to express schematic constraints on Grail functions and methods which impose a single space bound on the overall program (linear on input size). This paper designs a significant extension, introducing:

- the extension of certificates with resource policies that can express complex bounds on several resources for individual methods;
- a language describing resource policies and its formal semantics;
- the specification of target resource policies on the target device;
- extended certificate checking to relate target resource policies to claimed guarantees, as well as the check that claimed guarantees are indeed satisfied.

As a simple example of a guaranteed policy, we are able to express statements such as:

> "for positive integer inputs `n` and `m`, the method call `calc(int m,int n)` requires at most 16+42∗m+9∗m∗n JVM instructions to be executed."

More complex concrete bounds statements constructed with polynomials, logarithms, and exponentials, are allowed, providing they satisfy some reasonable restrictions described later. As well as time costs, we consider heap space consumption, maximum stack depth, and costs related to specific method calls. The latter is useful, for example, to bound the number of calls to expensive or security critical library methods made by client code: e.g., a program to be run on a mobile phone may be allowed to send two text messages but no more. Formally, we consider cost metrics, such as heap space consumption, to be supplied with an ordering. This allows us to relate different policies in the checking process.

Guaranteed policies are checked against target policies which express limits of constrained devices. An example is:

> "for all inputs $\mathtt{n} < 10$ and $\mathtt{m} < 10$, executing the `calc(int m,int n)` method must take no more than 2000 instructions."

Here the client of the delivered code provides a promise about the way the code will be invoked, and asks for a hard limit on resource consumption in turn. Fixing the format of both forms of policy allows us to use a simple checking process.

Code that is run on a target machine is often a combination of delivered methods and methods supplied by the platform. In this case, the resource consumption of the delivered methods may depend on the precise behaviour of the platform library functions, which is unknown at the time that the mobile code is certified. To deal with this scenario, we allow delivered policies to refer symbolically to *provided* functional bounds on platform functions whose implementation is unknown. We may express statements such as:

> "for a positive integer input `m`, the method `throwdice(m)` takes at most $m * F(6)$ JVM instructions to execute, where $F(x)$ is the number of instructions taken by the platform function `rand(int x)`."

In this case, the delivered certificate will contain a proof of the guaranteed bound under the *assumption* that the symbolic bound is satisfied. To (soundly) compute an overall worst case bound the platform must supply a (guaranteed and proven) bound which can be used during certificate checking.

Resource usage statements such as the above may not always hold unless particular safety conditions are met; or the resource usage may depend on non-functional (intensional) factors, such as the layout of data in memory. Consider, for example, the different space behaviour between deep and shallow copying of objects in memory, or a method whose complexity depends on the length of a list represented as a linked list sequence of objects. To deal with (and in particular, to prove) such cases, our preferred approach is to combine resource statements with *high-level typing invariants* which are maintained by our compiled methods.

This has been done for heap space usage bounds in the existing fixed policy scheme of MRG [BHMS05] based on the specialised type system of Hofmann and Jost [HJ03] for inferring space usage. In this paper we focus instead on new forms of resource statement not previously considered in the implementation of MRG, and how such statements can be expressed and related in our PCC architecture. The generation and proof of resource statements is beyond the scope of what is considered here, although in certain cases automatic generation of bounds for resources other than heap usage is feasible by extending existing techniques (for example, inferring stack depth by an extension of Hofmann-Jost is considered by Campbell [Cam05]).

Outline. In the next section we introduce Grail and the semantic notions for resource policies expressed on the Grail operational semantics, which is a simplified abstraction of the JVML semantics. Resource policies are statements in the Grail program logic, which is also introduced in Sect. 2. In Sect. 3 we introduce a simple language for describing two forms of resource policies, one for guaranteed policies delivered with code and the other for the target policy of a smart device. We use the standard format of Java security policy files augmented with dedicated forms of permission. In Sect. 4 we describe some mechanisms for checking policies and how this interacts with the usual proof checking process. Finally, Sect. 5 concludes with a summary of the status of our work on policies and a comparison with related work.

2 Resource policies for Grail

We want to make our resource policies precise and formalise their meaning. To do this, we first recall the Grail syntax, semantics and program logic, before considering the semantics of policies in Sect. 2.3.

2.1 Grail syntax and semantics

Grail is a functional language for writing imperative low-level code; we sketch a simplified version here. Together with intuition based on knowledge of the JVM, this sketch should suffice for an understanding of this paper; full details of Grail and its compilation scheme appear elsewhere [BMS03,MW04].

The simplified abstract syntax of Grail is as follows:

$$
\begin{array}{rl}
v ::= & \texttt{null} \;\mid\; i \\
a ::= & v \;\mid\; x \\
e ::= & a \;\mid\; op\ a\ a \;\mid\; \texttt{new}\ C \;\mid\; x.t \;\mid\; x.t{:=}a \\
& \mid\; \texttt{let val}\ x = e\ \texttt{in}\ e \;\mid\; \texttt{let val}\ () = e\ \texttt{in}\ e \;\mid\; \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid\; \texttt{call}\ f \;\mid\; C.m(\overline{a}) \\
op ::= & \texttt{add} \;\mid\; \texttt{sub} \;\mid\; \texttt{mul} \;\mid\; \texttt{div} \;\mid\; \texttt{=} \;\mid\; \texttt{<=} \;\mid\; \texttt{<} \;\mid\; \texttt{<=}
\end{array}
$$

A Grail program consists of a sequence of class definitions for class names C. Each class definition may contain declarations for fields t and for methods m. Each method m in turn declares a number of mutually recursive functions f together with an overall expression body.

Expressions e include arguments, which may be values v (the null reference and integer literals) or variables x. Integers `1` (or any non-zero value) and `0` are also used to represent boolean values `true` and `false` as on the JVM. The remaining expressions are formed from: binary operations, object construction, field selection and field update, binding and sequential composition (written as in SML, by binding to the unit value), function and method invocations. Strong syntactic restrictions ensure that all functions are tail recursive, so a function in Grail can be compiled directly into a branch instruction in the underlying virtual machine: this is reflected in the abstract syntax above by using the `call` expression which does not pass any arguments. Method invocation is different, and a method may have a number of arguments a which can be variables or literal values. To keep the presentation brief, we will only consider class (static) methods, although the full language includes instance methods, as well as many other features of JVML.

An example Grail program is shown in Fig. 1. This program defines a class `List` to represent linked lists, and a method `List.emptylist` which constructs a list of a given length whose `hd` fields all contain zero. The method is defined using two tail recursive functions `emptylist` and `emptylist_aux`. Programs in concrete Grail syntax are more verbose than the simplified abstract syntax shown above: we use the extra keywords `putfield`, `getfield` and `invokestatic` and some additional typing information is included, for example, on the `null` value and the `putfield` instruction. We will return to extend this example later.

Semantics. The semantics of Grail is given in terms of a *resource algebra* $\mathcal{R}$, extending a big-step evaluation relation based on the functional interpretation:

$$E \vdash h,\, e \;\Downarrow\; h',\, v,\, r$$

where E is an environment, h and h' are heaps (partial maps from locations to values), v is the result value (or () indicating the absence of a value) and r is a *resource value* from $\mathcal{R}$. The semantics is deterministic: whenever e evaluates in some E, h then v, h' and r are uniquely determined. Moreover, the resources r are a purely non-invasive annotation on the ordinary operational semantics; evaluation of an expression is not affected by the resources consumed in subexpressions (this is reminiscent of effects [TJ94]).

A resource algebra $\mathcal{R}$ has a carrier set R consisting of *resource values* $r \in R$, together with:

- A cost ordering $\leq \subseteq R \times R$
- For the atomic expressions, families of constants $\mathcal{R}^{\mathtt{null}} \in R$, $\mathcal{R}^{\mathtt{int}} \in R$, etc.
- For compound expressions, families of operations, e.g. $\mathcal{R}^{\mathtt{let}} \in R \times R \to R$.

```
class List {
        field int hd
        field List tl

        method static List emptylist(int n) =
          let
            val l = null[List]

            fun emptylist(int n,List l) =
              if n>0 then empty_aux(n,l) else l

            fun empty_aux(int n,List l) =
              let val cell = new <List()>()
                  val ()   = putfield cell <int List.hd> 0
                  val ()   = putfield cell <List List.tl> l
                  val n    = sub n 1
                  val l    = cell
                in
                   emptylist(n,l)
                end
          in
             emptylist(n,l)
          end
}
```

Fig. 1. Grail `List` class

The cost ordering expresses when one resource value is considered cheaper or better than another. The resource constants and operators are used to calculate costs by annotating the operational semantics; there is a constant or operator for each component of the syntax. An example rule is the rule for let-bindings (sequential composition with assignment):

$$\frac{E \vdash h,\, e_1 \;\Downarrow\; h_1,\, v_1,\, r_1 \qquad E[x := v_1] \vdash h_1,\, e_2 \;\Downarrow\; h_2,\, v,\, r_2}{E \vdash h,\, \texttt{let val}\; x = e_1\; \texttt{in}\; e_2 \;\Downarrow\; h_2,\, v,\, \mathcal{R}^{\texttt{let}}(r_1, r_2)}$$

For this paper we do not require additional properties of the resource algebra, although it is natural to impose further structure. For quantitative costs, for example, we may define the compound resource operators such as $\mathcal{R}^{\texttt{let}}$ in terms of an associative and commutative addition operator corresponding to sequential composition, as in the standard resource algebra described in Sect. 2.2.

The full definition of the semantics is in Table 2 at the end of the paper.

Program logic. Grail has a program logic which is formulated to take advantage of the functional semantics. Statements in the Grail Logic are written:

$$G \rhd e : P[E, h, h', v, r]$$

where e is a program expression and P is a predicate over the components of the operational semantics; G is a collection of assumptions of statements of the form $e' : P'$. This statement has a partial correctness reading: it states that whenever e evaluates in input environment E and heap h, then P holds for E, h, and the resulting heap h', value v, and resources consumed r.

Here is the rule in the logic for let-bindings:

$$\frac{G \rhd e_1 : P_1 \qquad G \rhd e_2 : P_2}{\begin{array}{l} G \rhd \texttt{let}\ x\ e_1\ e_2 : \{\exists\ h_1\, v_1, r_1\, r_2.\ P_1[E, h, h_1, v_1, r_1] \wedge \\ \qquad\qquad\qquad\qquad P_2[E[x := v_1], h_1, h', v, r_2] \wedge \\ \qquad\qquad\qquad\qquad r = \mathcal{R}^{\texttt{let}}(r_1, r_2)\} \end{array}}$$

This rule says that a let expression satisfies the assertion which is formed by combining two assertions P_1 and P_2 for the subexpressions, whenever there is an intermediate result state and result h_1, v_1, r_1 and the overall resource r consumed is given by the `let` operator applied to the component costs r_1 and r_2.

The full definition of the logic is shown in Table 3 at the end of the paper. Predicates in the logic are like post-assertions in VDM: they range over both input and output without needing auxiliary variables as would be necessary in Hoare logic. This allows powerful but comparatively simple rules for adaptation, derived from the second consequence rule shown in Table 3. The main power of the logic comes with the last two rules for recursive function and procedure calls; these allow one to establish the correctness of a function or method body under the assumption that recursive calls are already correct.

The program logic enjoys good meta-theoretic properties; in particular, it is sound and relative complete. Grail's syntax, semantics, program logic and meta-theory have been all formalised in the theorem prover Isabelle; the formalisation serves both to provide strong confidence in the meta-theoretical results and to provide an experimental PCC environment. The program logic does not define a notation for predicates: these are written in the ambient higher-order logic of the theorem prover; this allows powerful specifications which can directly use the available library functions for arithmetic, etc. Our work here extends the Isabelle formalisation which is presented elsewhere [ABH+05,ABH+04]. In the overview here we elide some of the technicalities of the Isabelle encoding (for further details see *loc. cit.*).

2.2 A standard resource algebra

We will suppose that a standard resource algebra is fixed by the application framework, which implies that the producer and consumer of mobile code have some agreement over which costs are of interest and how they are calculated.

As an example resource algebra which collects four costs of interest, we consider resource quadruples:

$$r = (\mathit{clock},\ \mathit{space},\ \mathit{depth},\ \mathit{methcnts})$$

$$
\begin{aligned}
\mathcal{R}^{\mathtt{null}} = \mathcal{R}^{\mathtt{int}} = \mathcal{R}^{\mathtt{var}} &= (1,0,0,\{\}) \\
\mathcal{R}^{\mathtt{prim}}(r_1,r_2) &= r_1 + r_2 + (1,0,0,\{\}) \\
\mathcal{R}^{\mathtt{new}}_{C} &= (3, \mathit{size}(C), 0, \{\}) \\
\mathcal{R}^{\mathtt{getf}} &= (2,0,0,\{\}) \\
\mathcal{R}^{\mathtt{putf}}(r) &= r + (2,0,0,\{\}) \\
\mathcal{R}^{\mathtt{let}}(r_1,r_2) &= r_1 + r_2 + (1,0,0,\{\}) \\
\mathcal{R}^{\mathtt{comp}}(r_1,r_2) &= r_1 + r_2 \\
\mathcal{R}^{\mathtt{if}}(r_1,r_2) &= r_1 + r_2 \\
\mathcal{R}^{\mathtt{call}}(r) &= r + (1,0,0,\{\}) \\
\mathcal{R}^{\mathtt{meth}}_{C.m,,\overline{a_i}}(r) &= r + (2 + |\overline{a_i}|, 0, 1 + |\overline{a_i}|, \{C.m\})
\end{aligned}
$$

$$(t_1,s_1,d_1,ms_1) + (t_2,s_2,d_2,ms_2) = (t_1+t_2, s_1+s_2, \mathit{max}(d_1,d_2), ms_1 \cup_+ ms_2)$$

$$(t_1,s_1,d_1,ms_1) \leq (t_2,s_2,d_2,ms_2) = t_1 \leq t_2 \wedge s_1 \leq s_2 \wedge d_1 \leq d_2 \wedge ms_1 \subseteq ms_2$$

Note: resource values r_k have the form $r_k = (t_k, s_k, ms_k, d_k)$ for $k = 1, 2$.
A tuple (t, s, d, ms) stands for $(\mathit{clock}, \mathit{space}, \mathit{depth}, \mathit{methcnts})$.
The notation $|\overline{a_i}|$ denotes the length of the list $a_1 \ldots a_n$ and $\cup_+$ is multiset union.

Table 1. Standard resource algebra

where the first three components range over natural numbers, and the last over multisets of method names.

The costs have the following meaning:

- *clock* is a JVM instruction counter, counting bytecodes executed;
- *space* is the cumulative size of allocated objects on the heap;
- *depth* is an approximation of the maximum frame stack depth;
- *methcnts* counts how many times each method is invoked.

The resource operators for this standard algebra are given in Table 1.

The time and space resources measured here have a standard meaning. For *clock*, we count JVM instructions under the Grail to JVML translation.[1] For *space* we measure memory usage based on the sizes of instance fields in a Java class (the function $\mathit{size}(C)$). For *depth* we approximate frame stack space based on the number of method parameters. The stack space calculation could easily be made more precise by incorporating the size needed for the local variables of each method. Finally, the method invocation counter *methcnts* accumulates method names invoked.

For a particular JVM implementation, these measures could be used to calculate approximate real time and space bounds, for example, based on empirical measurements of timings and knowledge of object overhead used in heap layout.

Notice that values in this resource algebra are composed of four independent components that could each be calculated separately within separate resource

[1] The details of this translation explain why `if` expressions are apparently free: the guard in the conditional is compiled into a test-and-branch instruction which is already accounted for by $\mathcal{R}^{\mathtt{prim}}$; similarly, sequential composition is just juxtaposition.

algebras. Each kind of resource has different properties, and will be expressed separately in our policy language described in Sect. 3.

This example algebra is similar to the one considered in the main MRG prototype described in [ABH+04], although there the heap size and method count components were not included. Many other interesting resource algebras can be given in this general scheme; see [ABH+05] for some particular examples and [ABM05] for an application of a more constrained form of resource algebra than that considered here. The important fact is that the soundness and completeness of the Grail Logic hold for any resource algebra.

2.3 Formal notions of resource policy

Given a notion of resource consumption and a way to calculate it, we can go on to define a formal notion of resource policy. With respect to the Grail semantics, a *resource policy RP* for expressions is a predicate on environments E, heaps h and resource values r, written as:

$$RP[E, h, r]$$

Intuitively, the policy determines acceptable resource limits for expressions executed in the given environment and heap. This is simply a restricted form of assertion in the program logic: a policy for an expression is a specification of its resource consumption in terms of its input taken from the environment E and heap h. In the mechanised Isabelle implementation we again express these predicates in Isabelle HOL, the meta-logic used to formalise the Grail syntax, semantics and logic rules.

Note that in general the policy may rely on a type safety invariant (or more generally, some invariant involving object containment and separation), as mentioned on page 4. In this case we must use specifications in the program logic which are conditional on type safety before evaluating expressions, and ensure type safety of the output afterwards. A resource policy would be embedded as:

$$P_{RP}[E, h, h', v, r] \quad \triangleq \quad TS[E, h] \Longrightarrow TS'[E, h, h', v] \wedge RP[E, h, r]$$

where TS and TS' are domain-specific safety invariants supplied by the certifying compiler. If the input environment and heap do not satisfy the safety invariant the policy is satisfied vacuously. For Camelot and its space-aware type system, the type safety invariant refers to the integrity of heap representations of high-level Camelot datatypes and the free list used for space reuse; the translation of this into *derived assertions* [BHMS05] in the Grail Logic may be understood as a special case of the above where the resource policy states that no heap space is consumed.

Definition 1. *An expression* e conforms *to a policy RP, written* $e \models RP$*, just in case:*

$$\forall E\, h\, r. \quad E \vdash h,\, e \;\Downarrow\; h',\, v,\, r \;\Longrightarrow\; RP[E, h, r].$$

Notice that this is a *partial correctness* interpretation, in that conformance is only considered for terminating expressions. Termination may be treated as an orthogonal issue, using a related logic as proposed in [ABH+05], or we may impose run-time monitoring to ensure that programs do not diverge and violate their resource bounds.

Policy conformance is a special case of validity of assertions in the program logic, which means that we have a sound and complete logic for establishing conformance of resource policies.

Theorem 1 (cf. [ABH+05]). $\{\} \rhd e : RP[E, h, r]$ *if and only if* $e \models RP$.

Of course, we are ultimately interested in resource policies for method bodies; the environment declares the parameters of the method.

Example. The resource policy for the standard resource algebra given by

$$RP_{calc}[E, h, (t, s, d, ms)] = t \leq 16 + 42 * E(\mathtt{m}) + 9 * E(\mathtt{m}) * E(\mathtt{n})$$

formalises the example policy described in words for the `calc` method in Sect. 1. We claim that it is satisfied by the implementation of the `calc` method shown in Fig. 2, which shows a Grail program to count the results of `m` throws of an `n`-sided dice. The result of throwing a dice is represented by a call to a platform function `Platform.Random.rand(n)`. To establish the bound above we must assume additionally that the platform function satisfies a policy:

$$RP_{Platform.Random.rand}[E, h, (t, s, d, ms)] = t \leq 20$$

i.e., that the number of instructions executed in the random method is at most 20.

For a given resource algebra, we can relate different policies in a refinement ordering.

Definition 2. *A resource policy* RP_1 refines *another policy* RP_2*, if*

$$\forall e.e \models RP_1 \Longrightarrow e \models RP_2$$

If RP_1 refines RP_2, then by definition any expression which conforms to the first policy also conforms to the second (more permissive) policy.

Example. A refinement for RP_{calc} is the policy given by:

$$RP'_{calc}[E, h, (t, s, d, ms)] = t \leq 16 + 42 * E(\mathtt{m}) + 9 * E(\mathtt{m}) * E(\mathtt{n}) \ \wedge \ s \leq 2 * E(\mathtt{n})$$

which requires additionally that the `calc` method allocates no more than $2 * \mathtt{n}$ words of heap space during its execution, which is also satisfied by the example program in Fig. 2, which allocates a list of length `n`.

```
method static void addthrow(List l, int n) =
  let
     fun update_pos(List l) =
       let val i  = getfield l <int List.hd>
           val i  = add i 1
           val () = putfield l <int List.hd> i
        in () end

     fun addthrow(int n,List l) =
       if n=0 then update_pos(l) else addthrow_aux(l,n)

     fun addthrow_aux(List l, int n) =
       let val l    = getfield l <List List.tl>
           val n    = sub n 1
         in addthrow(n,l) end
   in addthrow(n,l) end

// Throw n-sided dice m times and count results in a list
method static List calc(int n, int m) =
  let
     val l = invokestatic <List List.emptylist (int)> (n)

     fun make_throws (List l, int n, int m) =
          if m=0 then l
                 else next_throw(l,n,m)

     fun next_throw (List l, int n, int m) =
        let val r  = invokestatic <int Platform.Random.rand(int)> (n)
            val () = invokestatic <void List.addthrow(List,int)> (l, r)
            val m  = sub m 1
        in make_throws(l,n,m) end
  in make_throws(l,n,m) end
```

Fig. 2. Grail program to count dice throws

Apart from adding requirements for further kinds of resource, one policy refines another of the form of RP_{calc} if it places a tighter bound on the resource consumption. We are interested in policies which place bounds on resource consumption in the manner of RP_{calc} above, but the form of resource policy allowed so far is much more general. For the framework here we at least want policies to respect the ordering of resources (downward closure):

Definition 3. *A resource policy RP* respects $\leq$ *for R, if for all E and h,*

$$\forall r, r' \in R. \quad r \leq r' \wedge RP[E, h, r'] \Longrightarrow RP[E, h, r].$$

Clearly our example policies respect the resource ordering of the standard resource algebra. From now on we restrict to policies which respect resource order-

ing on the resource algebra of interest. This ensures that alternative implementations of methods with better resource behaviour may be selected to implement any policy. It rules out, for example, policies which describe a minimum resource usage.

Based on these definitions we can define a simple theory of policy refinement and its implementation in Isabelle. But this semantic notion of policy is still too general: we introduce further restrictions in the next section, by introducing our two specific forms of policy liable to be useful in practice on constrained devices.

3 Expressing resource policies

Resource policies can be written in our formal logic in the same way as program logic assertions, but this is an internal format and it is too rich: it is more useful to consider a way of expressing policies that is meaningful for the user. For this we need to investigate a *policy language*. Since we have an infrastructure for the Java platform, we will extend Java's existing notion of *permissions* and *security policy*. Security policies in Java are specified in files created with the `policytool` program or otherwise. For example, the trivial policy file:

```
grant {
  permission java.security.AllPermission;
};
```

describes a policy which grants all permissions. More interestingly, the policy file:

```
grant codeBase "file:${user.dir}/" {
  permission java.io.FilePermission "/etc/passwd", "read";
};
```

gives special permissions to code executed from the user's home directory, to read the contents of the password file.

To express resource policies for our application, we will introduce new forms of permission for the guaranteed policy and for the target policy. Permissions in Java are usually associated with running code: the security manager will raise an exception if some method does not possess appropriate permissions. However, our overloading of the concept will be useful: we can extend the built-in mechanisms for loading policy files and comparing between them, as well as allow a mechanism for run-time instrumentation.

The starting point is the `Permission` class, which defines an abstract method `implies` that compares two permissions. If permission p_1 implies p_2, then code which is granted permission p_1 also has permission p_2. If we can implement this method in a way which is consistent with our formal interpretation of resource permissions, we can integrate some parts of our policy refinement checking into the Java security model.

Guaranteed policies. Guaranteed policies deliver a parametrised bound on resource consumption, expressed as a nondecreasing function of a measure on each of the inputs. For integer inputs, the measure takes the input parameter

unchanged; for other types we define a type-dependent coercion into the integers, in a standard way. For example:

```
permission ClockGuarantee  "List.calc(int m, int n)"
                           "16 + 42*m + 9*m*n"
permission SpaceGuarantee  "List.calc(int m, int n)" "2*n"
```

expresses the time and space bounds of the earlier example. Several permissions together define a resource policy for a method; resource policies for several methods define an overall guaranteed policy for a delivered program.

Target policies. The second form of policy is simpler and expresses some fixed hard limits of the particular target machine. For example:

```
permission ClockTarget "List.calc(int m, int n)"
                       "500, m<=3, n<=4"
permission SpaceTarget "List.calc(int m, int n)" "100"
```

Here, the absolute maximum execution time allowed for the `calc` method is 500 steps, and the target environment is providing a promise that the input parameters will satisfy the constraints shown. For heap space, the maximum new space consumed when evaluating the method is 100 words, irrespective of the input parameters to the method.

3.1 Permissions language

Formally, Java methods are selected by a *method descriptor*. Method descriptors are described by the following grammar:

$$\begin{array}{rcl} \mathit{mdesc} & ::= & \mathit{mspec}\,(\mathit{type}\ x, \ldots, \mathit{type}\ x) \\ \mathit{mspec} & ::= & C\#m \quad | \quad C\,.\,m \end{array}$$

A method descriptor can disambiguate overloaded methods. A static method has its usual Java name (e.g., `java.lang.Integer.parseInt`), whereas an instance method has a name of the form `java.lang.Integer#toString`.

Guaranteed policies delivered with the code are described by the following grammar (for clarity we elide the quotation marks required for Java policy files):

$$\begin{array}{rcl} G & ::= & \textbf{permission}\ \mathit{gdesc}\ \mathit{bound} \\ \mathit{gdesc} & ::= & \texttt{ClockGuarantee}\ \mathit{mdesc} \\ & | & \texttt{SpaceGuarantee}\ \mathit{mdesc} \\ & | & \texttt{DepthGuarantee}\ \mathit{mdesc} \\ & | & \texttt{MethcntGuarantee}\ \mathit{mdesc}, \mathit{mdesc} \\ \mathit{bound} & ::= & f \\ f & ::= & K \quad | \quad x \quad | \quad f + f \quad | \quad f * f \quad | \quad f\,\hat{}\,f \\ & | & \texttt{log}(f) \quad | \quad \texttt{min}(f, f) \quad | \quad \texttt{max}(f, f) \\ & | & \mathit{gdesc}\,(v, \ldots, v) \\ v & ::= & K \quad | \quad x \end{array}$$

while target policies are given by the grammar:

$$
\begin{aligned}
T &::= \textbf{permission}\ \mathit{tdesc}\ \mathit{limit}, \mathit{constraints} \\
\mathit{tdesc} &::= \texttt{ClockTarget}\ \mathit{mdesc} \\
&\quad \mid \texttt{SpaceTarget}\ \mathit{mdesc} \\
&\quad \mid \texttt{DepthTarget}\ \mathit{mdesc} \\
&\quad \mid \texttt{MethcntTarget}\ \mathit{mdesc}, \mathit{mdesc} \\
\mathit{limit} &::= K \\
\mathit{constraints} &::= x <= K, \ldots, x <= K
\end{aligned}
$$

where K denotes a non-negative integer constant and x denotes a variable occurring in the method descriptor or (in the case of instance methods only) the keyword `this`.

The permissions defined above mirror the four classes of cost defined in Sect. 2.2 and describe guaranteed and target bounds for these costs. The `Clock`, `Space` and `Depth` permissions take a single method descriptor, specifying the method to which the bounds apply. In contrast the `Methcnt` permissions take two method descriptors m_1 and m_2 say; the meaning is that when m_1 is invoked, it will cause no more than the specified number of invocations of m_2 to occur.

In the final expression former for f, we allow the bounds in guaranteed policies to refer to other guaranteed policies. For example, we may have a policy such as:

```
permission ClockGuarantee C.m(int n)
           4*n + 3*(ClockGuarantee D.rand(int k) (n))
```

which states that the execution time of the method `C.m` depends on that of `D.rand`.

Note that we do not allow arbitrary bounding functions in guaranteed policies, but only ones of the form f above. It is not hard to see that functions which are generated by the grammar above are all nondecreasing. This will be important in our policy-checking procedure.

3.2 Semantics of resource policies

So far, resource policies are purely symbolic; we now give their semantic interpretation. Recall that a guaranteed policy for a resource R consists of a method signature (for the method m say) followed by a bound f. The intention is that f describes a function which is an upper bound on the amount of R which is consumed by any invocation of m, the bound being given as a function of the inputs to m. To this end, we require that the variables appearing in f are a subset of the variables appearing in the signature of m (with the addition of `this` if m is an instance method). Since not all of the arguments of m may influence its resource consumption we do not insist that all arguments appear in f. For example, if we have a method `pow(int p,int q)` which calculates $\texttt{p}^{\texttt{q}}$ by repeated multiplication then it is probable that the execution time of `pow` would only depend on `q`.

In order to interpret the bounding expressions appearing in policies, we assume that every Java type t has an associated *measure* $\|\cdot\| : t \to \mathbb{N}$. For x of type `int` or `long`, $\|x\| = |x|$, the absolute value of x. For floating-point types, we put $\|x\| = \lceil |x| \rceil$. For heap-allocated objects o, we define $\|o\|$ to be the size of the object allocated in memory (*size*(C) when o is of class C).[2]

We will sometimes wish to deal with unbounded quantities: to facilitate this we consider values lying in the set $\widehat{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$. Arithmetic operators are extended from $\mathbb{N}$ to $\widehat{\mathbb{N}}$ in the obvious way: for example, $x + \infty = \infty$ and $\min(x, \infty) = x$ for all $x \in \widehat{\mathbb{N}}$. Given a bounding expression f and an environment E mapping identifiers to values (in $\widehat{\mathbb{N}}$), we define an interpretation $[\![f]\!]_E \in \widehat{\mathbb{N}}$ as follows:

$$
\begin{aligned}
[\![K]\!]_E &= K \\
[\![x]\!]_E &= \|E(x)\| \\
[\![f_1 + f_2]\!]_E &= [\![f_1]\!]_E + [\![f_2]\!]_E \\
[\![f_1 * f_2]\!]_E &= [\![f_1]\!]_E [\![f_2]\!]_E \\
[\![f_1 \hat{} f_2]\!]_E &= [\![f_1]\!]_E^{[\![f_2]\!]_E} \\
[\![\mathtt{log} f]\!]_E &= \begin{cases} 0 \text{ if } [\![f]\!]_E = 0 \\ \max\{k : 2^k \leq [\![f]\!]_E\} \text{ otherwise} \end{cases} \\
[\![\mathtt{min}(f_1, f_2)]\!]_E &= \min([\![f_1]\!]_E, [\![f_2]\!]_E) \\
[\![\mathtt{max}(f_1, f_2)]\!]_E &= \max([\![f_1]\!]_E, [\![f_2]\!]_E) \\
[\![g\,(v_1, \ldots, v_n)\,]\!] &= [\![T_g]\!]_{[\,x_i \mapsto [\![v_i]\!]_E\,]}
\end{aligned}
$$

where, in the last line, the x_i are the variables appearing in the first method descriptor inside g, and T_g stands for the bounding expression for the permission g. To interpret expressions involving other permissions, we first collect together all guaranteed policies from the delivered code and platform. We must disallow circular references between guarantee policies, as this could lead to infinite recursion while evaluating bounding expressions.

Note that if f is a bounding expression in the variables $\{x_1, \ldots, x_n\}$ then there is an induced function $\bar{f} : \widehat{\mathbb{N}}^n \to \widehat{\mathbb{N}}$ defined by

$$\bar{f}(u_1, \ldots, u_n) = [\![f]\!]_{\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}}.$$

Every such $\bar{f}$ is a nondecreasing function on $\widehat{\mathbb{N}}^n$.

Given this semantic interpretation for new forms of permission, it is straightforward to convert Java policies to their formal equivalents in the Isabelle program logic, so that policy conformance and refinement can be checked formally. However, we have designed the format of policies so that refinement can be checked simply, without needing arbitrary proof. This means that we must trust an implementation of the checking procedure, described next.

[2] This size function is not flexible enough for richer forms of resource specification such as those expressed by the Camelot type system; for such cases we would want to allow additional user-defined size functions as part of a proof certificate.

4 Checking policies

Suppose that we have been supplied with a guaranteed policy G for a static method m stating that a resource R is bounded by the function f.

Let the variables occurring in the expression f be $x_1, \ldots, x_n$, which are a subset of the set of formal arguments appearing in the signature for m.

Now suppose that we have a target policy T for the method m and the resource R. Recall that T consists of a signature for m (without loss of generality we assume that the formal arguments appearing in the signatures for m in G and T are identical) followed by a constant b and a sequence of constraints for the formal arguments of m. The interpretation of T is that the code consumer requires that no more than b units of R are consumed, provided that the inputs to m do not exceed the given bounds.

By eliminating redundant constraints and adding vacuous constraints $x_j \leq \infty$ for variables not explicitly constrained in T we form a set of constraints $\{x_1 \leq b_1, \ldots x_n \leq b_n\}$ (with $b_i \in \widehat{\mathbb{N}}$) where each x_i appears precisely once.

Recall that f induces a function $\bar{f} : \widehat{\mathbb{N}}^n \to \widehat{\mathbb{N}}$. The code producer has supplied a proof that the resource usage of m when applied to a given set of arguments is bounded above by the value of $\bar{f}$ applied to the appropriate subset of the arguments of m. Since $\bar{f}$ is nondecreasing it follows that the maximum resource usage of m subject to $\{x_i \leq b_i\}$ is

$$\sup \{\bar{f}(u_1, \ldots, u_n) : u_i \in \widehat{\mathbb{N}}, u_i \leq b_i\} = \bar{f}(b_1, \ldots, b_n) = [\![f]\!]_E$$

where E is the environment $\{x_1 \mapsto b_1, \ldots, x_n \mapsto b_n\}$ (the resource usage is unbounded if $[\![f]\!]_E = \infty$). Thus to check the validity of T we need merely check whether $[\![f]\!]_E \leq b$.

For instance methods we follow the same procedure, but one must also consider the variable `this`.

4.1 Remarks

The policy-checking strategy described above depends crucially on the fact that the bounding functions given in guaranteed policies are nondecreasing. Note that the grammar for bounding functions makes this property manifest; a simple syntactic check suffices to show that a purported bound is nondecreasing, and so no extra overhead of proof-checking is required to establish this. This is a particular advantage in the scenario mentioned in the introduction, where we may ship several possible implementations of library functions with different resource behaviour which are used by the target device; it would be possible try to optimise resource usage by rearranging its choice of methods within given target policies. More importantly, this step can be done more quickly than VCG and proof checking, avoiding the need to check proofs when a policy cannot be met.

We have considered policy-checking with more general policy formats. For example, the code producer might supply code with some certified bounding

function f and the consumer may require to know that some other bounding function g is satisfied for some set of inputs (the case we consider above is when g is constant). In this case, the consumer has to check that $g - f$ is positive for some set of inputs, and since $g - f$ could essentially be any function, this is a difficult problem. Furthermore, it is not possible for the code producer to provide any help (in the form of proof, for instance) since it has no *a priori* knowledge of the bounds that the consumer will require to be satisfied. Policy-checking in this general situation thus appears to be infeasible.

5 Conclusions

We have described a way of generalising the present proof-carrying code infrastructure of the MRG project to include resource policies based on assertions on bytecode expressed in the Grail program logic. Policies are naturally treated as special cases of assertions in the logic, but we want to express them in a simpler and more uniform way, in particular, to allow an efficient check of whether mobile code supplied with a guaranteed policy implements the target policy of particular device. To this end, we introduced syntax and semantics for two forms of policy embedded as Java permissions in Java security policy files. Using the Java file format and permissions mechanism allows us to implement a sound test for policy refinement inside Java. Then checking policy conformance is reduced to checking that the code satisfies the guaranteed policy claimed for it (delivered with the certificate) and that the guaranteed policy implies the policy desired by the client.

The security model here is quite analogous to the present security mechanisms in Java, where code is implicitly supplied with its "code base" (origin) which may be checked against permitted code bases, and where code may be supplied with cryptographic signatures and these signatures may be accepted according to the policy.

More work is required to implement our policies inside the full MRG architecture and try full-sized examples. So far we have constructed necessary extensions to the Grail Logic, conducted experimental verifications and implemented a prototype parser and a checker for resource policies. It remains to integrate with the Java platform (perhaps following the technique described in [GP05]), and to embed guaranteed policies in certificates — ultimately with automation extending that provided for the current fixed policy derived from the Camelot type system.

Related Work. Other researchers have worked on inferring and proving static bounds on different kinds of resources both for high-level languages and low-level ones, using type-based and logical techniques e.g., [CW00,VH04,HP99,BPS05]. Recent work [CEI05] has explored combinations of static and dynamic methods, which would also be useful in our setting. In this paper we concentrated on the mechanism of describing policies rather than the mechanism of inferring, proving or dynamically checking them; our approach would still be applicable if we used

other techniques for those steps. Moreover, the basic ideas for the PCC architecture based on delivering policies with code are not specific to resource usage policies. Although important for general adoption, there seems to be relatively little published work on how policies are described and delivered in other PCC settings. One of the original PCC architectures described by Necula [Nec98] proposed the negotiation of policy between code producer and code consumer, in principle allowing the certificate to be specially adapted to the target requirements. In later work this was simplified to a fixed type-safety policy which would also work for a store-and-forward network.

Away from PCC, there is other related work on specification of resource behaviour for compiled Java programs. Most developments focus on dynamic checking. The Java Resource Management API (RM API)[CHS+03] (a development of the JRes interface [CvE98]) is a flexible mechanism which allows Java-based platforms to manage and monitor resource consumption. Resource policies are implemented via *resource domains*, which supply units of a given resource to applications (more precisely, to Java *isolates*) and also allow client applications to query availability of resources. To expose a resource through the RM API, the implementation of the resource includes code which records consumption and destruction, requiring modification of system classes. CPU time is monitored by a separate thread which periodically polls the operating system. Another resource accounting system for Java is J-SEAL2 [BHV01,J-S], which performs its resource accounting via bytecode instrumentation to insert calls to record resource allocations; CPU time again requires special handling, but in this case an estimate is calculated from the number of bytecode instructions executed. In both Java RM API and J-SEAL2, resource monitoring is dynamic; this can result in more accurate tracking and allocation of resources than static prediction as in our approach, but it requires runtime overhead and the need for recovery mechanisms in the case of resource exhaustion.

We have introduced a very simple policy language here. There are connections to work on policy languages in other domains of computer security, such as the use of Datalog, or the generation of large databases from policies, as in SELinux [LS01]. It would be interesting to consider whether one may usefully express our policies in languages such as these. However, the fundamental problem here is different: rather than querying some policy database to see if some access should be granted, the central question we have considered is *refinement* between policies, given that we already have guaranteed conformance for some program and a particular given policy.

Future work. There are several directions for further work. Most importantly, we need more automated mechanisms to provide the resource bound guarantees which are delivered with mobile code, and ways to analyse and predict the resource behaviour of existing platform library functions. We also need to undertake practical experiments on a particular platform to calibrate our cost model, and ensure that our resource guarantees can indeed be fulfilled on a particular architecture. Knowledge of an architecture and in particular additional hooks (such as provided by the Real-Time Java Specification [B+00]) could help pro-

vide sharper guarantees, including modelling of garbage collection, for example. Concerning the certificate checking mechanism, it would be interesting to investigate the combination of static and dynamic techniques, as outlined in the introduction. We plan to pursue some of these activities in the recently started EPSRC Project ReQueST, which is investigating resource bound certification for Grid computing.

Acknowledgements. We're grateful to our colleagues working on the MRG project for discussions on the topic of this paper, in particular to Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Ian Stark for collaboration on resource algebras and to Lennart, Martin, Alberto, and Hans-Wolfgang Loidl for their collaboration on the Grail Logic. Our work was supported by the European Community as part of the MRG and Mobius projects (IST-2001-33149 and FP6-015905), as well as by the EPSRC ReQueST project (EP/C537068/1). This paper reflects only the authors' views and neither the Community nor EPSRC is liable for any use that may be made of the information contained within it.

References

[ABH+04] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In *Proc. of 17th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, Lecture Notes in Computer Science, Heidelberg, September 2004. Springer.

[ABH+05] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. Technical Report EDI-INF-RR-0296, Informatics, University of Edinburgh, July 2005.

[ABM05] David Aspinall, Lennart Beringer, and Alberto Momigliano. Optimisation validation. Technical report, Informatics, University of Edinburgh, December 2005.

[AGH+05] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, number 3362 in Lecture Notes in Computer Science, pages 1–26. Springer-Verlag, 2005.

[B+00] Greg Bollella et al. *The Real-time Specification for Java*. Addison-Wesley, 2000.

[BHMS05] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In Andrei Voronkov Franz Baader, editor, *Proc. Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004*, volume 3425 of *Lecture Notes in Computer Science*, pages 347–362. Springer, Feb 2005.

[BHV01] Walter Binder, Jane G. Hulaas, and Alex Villazón. Portable resource control in Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 139–155, New York, NY, USA, 2001. ACM Press.

[BMS03] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. *Electronic Notes in Theoretical Computer Science*, 85(1), June 2003.

[BPS05] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In B. Aichernig and B. Beckert, editors, *Proceedings of SEFM'05*. IEEE Press, 2005.

[Cam05] Brian Campbell. Folding stack memory usage prediction into heap. In *Proceedings of Quantitative Aspects of Programming Languages Workshop, ETAPS 2005*, April 2005.

[CEI05] Ajay Chander, David Espinosa, and Nayeem Islam. Enforcing resource bounds via static verification of dynamic checks. In *Proc. ESOP 2005*, Lecture Notes in Computer Science, Heidelberg, 2005. Springer-Verlag LNCS.

[CHS+03] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciaran Bryce. Sun Microsystems Technical Report TR-2003-124: A resource management interface for the Java platform, May 2003.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for Java. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 21–35, New York, NY, USA, 1998. ACM Press.

[CW00] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symp. Principles of Prog. Lang. (POPL)*, pages 184–198. ACM, 2000.

[GP05] Stephen Gilmore and Matthew Prowse. Proof-carrying bytecode. In *Proceedings of First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '05)*, Edinburgh, Scotland, April 2005.

[HJ03] 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*, volume 38 of *ACM SIGPLAN Notices*, pages 185–197, New York, January 2003. ACM Press.

[HP99] J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *Proc. International Conference on Functional Programming (ACM). Paris, September '99*, 1999.

[J-S] J-SEAL2 website. See www.jseal2.com.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In Clem Cole, editor, *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 2001.

[MW04] Kenneth MacKenzie and Nicholas Wolverson. Camelot and grail: resource-aware functional programming on the JVM. In *Trends in Functional Programing*, volume 4, pages 29–46. Intellect, 2004.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[VH04] Pedro Vasconcelos and Kevin Hammond. Inferring costs for recursive, polymorphic and higher-order functional programs. In *IFL 2003: Proceedings of the 15th International Workshop on the Implementation of Functional Languages*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

$$\frac{}{E \vdash h,\ a\ \Downarrow\ h,\ eval_E(a),\ cost(a)}$$

$$\frac{r_1 = cost(a_1) \qquad r_2 = cost(a_2)}{E \vdash h,\ op\ a_1\ a_2\ \Downarrow\ h,\ op(eval_E(a_1), eval_E(a_2)),\ \mathcal{R}^{\texttt{prim}}(r_1, r_2)}$$

$$\frac{l = \mathsf{freshloc}(h)}{E \vdash h,\ \texttt{new}\ c\ \Downarrow\ h[l \mapsto (c, \{t_i := initval_i\})],\ l,\ \mathcal{R}^{\texttt{new}}}$$

$$\frac{E\langle x\rangle = l \qquad l \in dom(h)}{E \vdash h,\ x.t\ \Downarrow\ h,\ h(l).t,\ \mathcal{R}^{\texttt{getf}}}$$

$$\frac{E\langle x\rangle = l \qquad l \in dom(h)}{E \vdash h,\ x.t{:=}a\ \Downarrow\ h[l.t \mapsto eval_E(a)],\ (),\ \mathcal{R}^{\texttt{putf}}(cost(a))}$$

$$\frac{E \vdash h,\ e_1\ \Downarrow\ h_1,\ v_1,\ r_1 \qquad E \vdash h_1,\ e_2\ \Downarrow\ h',\ v,\ r_2}{E \vdash h,\ \texttt{let val}\ () = e_1\ \texttt{in}\ e_2\ \Downarrow\ h',\ v,\ \mathcal{R}^{\texttt{comp}}(r_1, r_2)}$$

$$\frac{E \vdash h,\ e_1\ \Downarrow\ h_1,\ v_1,\ r_1 \qquad E\langle x := v_1\rangle \vdash h_1,\ e_2\ \Downarrow\ h',\ v,\ r_2}{E \vdash h,\ \texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \Downarrow\ h',\ v,\ \mathcal{R}^{\texttt{let}}(r_1, r_2)}$$

$$\frac{E \vdash h,\ e_1\ \Downarrow\ h_1,\ v_1,\ r_1 \qquad v_1 \neq \texttt{false} \qquad E \vdash h_1,\ e_2\ \Downarrow\ h',\ v,\ r_2}{E \vdash h,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \Downarrow\ h',\ v,\ \mathcal{R}^{\texttt{if}}(r_1, r_2)}$$

$$\frac{E \vdash h,\ e_1\ \Downarrow\ h_1,\ \texttt{false},\ r_1 \qquad E \vdash h_1,\ e_3\ \Downarrow\ h',\ v,\ r_2}{E \vdash h,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \Downarrow\ h',\ v,\ \mathcal{R}^{\texttt{if}}(r_1, r_2)}$$

$$\frac{E \vdash h,\ f_{body}\ \Downarrow\ h',\ v,\ r}{E \vdash h,\ \texttt{call}\ f\ \Downarrow\ h',\ v,\ \mathcal{R}^{\texttt{call}}(r)}$$

$$\frac{\{x_i := eval_E(a_i)\} \vdash h,\ C.m_{body}\ \Downarrow\ h',\ v,\ r}{E \vdash h,\ C.m(\overline{a})\ \Downarrow\ h',\ v,\ \mathcal{R}^{\texttt{meth}}(r)}$$

Notes:

- argument evaluation is defined by $eval_E(x) = E(x)$ and $eval_E(v) = v$;
- argument costs are defined as $cost(\texttt{null}) = \mathcal{R}^{\texttt{null}}$, $cost(i) = \mathcal{R}^{\texttt{int}}$, $cost(x) = \mathcal{R}^{\texttt{var}}$;
- the function $\mathsf{freshloc}(h)$ returns a fresh location l not in the domain of h;
- $initval_i$ stands for the initial value of the field t_i in class c;
- f_{body} and $C.m_{body}$ denote the definition of function f and method $C.m$ respectively.

Table 2. Grail operational semantics

$$\frac{e : P \ \in G}{G \rhd e : P} \qquad \frac{G \rhd e : P \quad P \Longrightarrow Q}{G \rhd e : Q}$$

$$\frac{}{G \rhd a : \{h' = h \ \wedge \ v = eval_E(x) \ \wedge \ r = cost(a)\}}$$

$$\frac{}{\begin{array}{l} G \rhd op\ a_1\ a_2 : \{h' = h \ \wedge \ \ v = op(eval_E(a_1), eval_E(a_2)) \wedge \\ \qquad\qquad\qquad\qquad r = \mathcal{R}^{\texttt{prim}}(cost(a_1), cost(a_2))\} \end{array}}$$

$$\frac{}{G \rhd \texttt{new}\ C : \{v = \mathsf{freshloc}(h) \ \wedge \ h' = h[v \mapsto (C, \{t_i := initval_i\})] \ \wedge \ r = \mathcal{R}^{\texttt{new}}\}}$$

$$\frac{G \rhd e_1 : P_1 \qquad G \rhd e_2 : P_2}{\begin{array}{l} G \rhd \texttt{let val}\ x = e_1\ \texttt{in}\ e_2 : \{\exists\ h_1\, v_1, r_1\, r_2.\ P_1[E, h, h_1, v_1, r_1] \wedge \\ \qquad\qquad\qquad\qquad P_2[E[x := v_1], h_1, h', v, r_2] \ \wedge \\ \qquad\qquad\qquad\qquad r = \mathcal{R}^{\texttt{let}}(r_1, r_2)\} \end{array}}$$

$$\frac{G \rhd e_1 : P_1 \qquad G \rhd e_2 : P_2}{\begin{array}{l} G \rhd \texttt{let val}\ ()\, = e_1\ \texttt{in}\ e_2 : \{\exists\ h_1\, r_1\, r_2.\ P_1[E, h, h_1, (), r_1] \wedge \\ \qquad\qquad\qquad\qquad P_2[E, h_1, h', v, r_2] \ \wedge \\ \qquad\qquad\qquad\qquad r = \mathcal{R}^{\texttt{comp}}(r_1, r_2)\} \end{array}}$$

$$\frac{G \rhd e_1 : P_1 \qquad G \rhd e_2 : P_2 \qquad G \rhd e_3 : P_3}{\begin{array}{l} G \rhd \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \{\exists\ h_1\, v_1\, r_1\, r_2.\ P_1[E, h, h_1, v_1, r_1] \wedge \\ \quad (v_1 \neq \texttt{false} \Longrightarrow P_2[E, h_1, h', v, r_2]) \wedge \\ \quad (v_1 = \texttt{false} \Longrightarrow P_3[E, h_1, h', v, r_2]) \ \wedge \ r = \mathcal{R}^{\texttt{if}}(r_1, r_2)\} \end{array}}$$

$$\frac{G, \texttt{call}\ f : P \ \ \rhd f_{body} : \{P[E, h, h', v, \mathcal{R}^{\texttt{call}}(r)]\}}{G \rhd \texttt{call}\ f : P}$$

$$\frac{G, c.m(y) : P \rhd m_{body} : \{P[E, h, h', v, \mathcal{R}^{\texttt{meth}}(r)]\}}{G \rhd c.m(y) : P}$$

Note: assertions in braces $\{\ldots\}$ have standard free variables E, h, h', v, r.
The notation $P[E_1, h_1, h_1', v_1, r_1]$ indicates the instantiation of a predicate.

Table 3. Grail Logic