A Program Logic for Resources

David Aspinall [a,*] Lennart Beringer [a] Martin Hofmann [b]
Hans-Wolfgang Loidl [b] Alberto Momigliano [a]

[a] *Laboratory for the Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland*

[b] *Institut für Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany*

Abstract

We introduce a reasoning infrastructure for proving statements on resource consumption in an abstract fragment of the Java Virtual Machine Language (JVML). The infrastructure is based on a small hierarchy of program logics, with increasing levels of abstraction: at the top there is a type system for a high-level language that encodes resource consumption. The infrastructure is designed to be used in a proof-carrying code (PCC) scenario, where mobile programs can be equipped with formal evidence that they have good resource behaviour.

This article presents the core logic in our infrastructure, a VDM-style program logic for partial correctness, that can make statements about resource consumption in a general form. We establish some important results for this logic, including soundness and completeness with respect to a resource-aware operational semantics for the JVML. We also present a second logic built on top of the core logic, which is used to express termination; it is also shown to be sound and complete. The entire infrastructure has been formalised in the theorem prover Isabelle/HOL, both to enhance confidence in the meta-theoretical results, and to provide a prototype implementation for PCC. We give examples to show the usefulness of this approach, including proofs of resource bounds on code resulting from compiling high-level functional programs.

Key words: Program Logic, Proof-carrying-code, Object-oriented Languages, Java Virtual Machine Language, Cost Modelling, Quantitative Type-Systems, Lightweight Verification

* Corresponding author
Email addresses: da@inf.ed.ac.uk (David Aspinall), lenb@inf.ed.ac.uk (Lennart Beringer), mhofmann@informatik.uni-muenchen.de (Martin Hofmann), hwloidl@informatik.uni-muenchen.de (Hans-Wolfgang Loidl), amomigl1@inf.ed.ac.uk (Alberto Momigliano).

1 Introduction

When we receive a program from another party, we want to know that it serves its intended purpose. Apart from having correct functional behaviour, it is important that a program meets non-functional requirements such as reasonable resource consumption. Indeed, non-functional requirements of application programs can often be *more* important than functional ones, as they impact the overall security and robustness of a system. For example, a game to be run on a mobile telephone should not make expensive network demands or drain the handset batteries excessively; this is more important to the user and network operator than, say, that the game calculates the correct score.

We want to equip mobile code with guarantees that resource constraints are satisfied, following the proof-carrying code paradigm (PCC, (1)). PCC has emerged as a powerful means to guarantee type-correct behaviour or the adherence to various security policies. Low-level code is sent together with a certificate which contains a formal logical proof that the program adheres to a given safety or security policy.

The bedrock of PCC is the logic for low-level code used to state and prove properties of interest. The logic must satisfy several crucial requirements. First, it must be expressive enough to state and prove the policies required for the programs of interest. Second, it must be sound: proofs in the logic must be honest to the behaviour of the code on the target machine. Third, we must meet engineering requirements for implementing proof-carrying code: we must have some way of effectively generating proofs in the logic and transmitting them in proof certificates, and efficiently checking them on receipt.

In this paper we present a program logic for low-level languages designed to meet these requirements, as part of a prototype proof-carrying code infrastructure focused on certifying resource usage of Java bytecode programs (this is part of our work on the *Mobile Resource Guarantees* project (2), see (3) for an overview). Our program logic is expressive enough to describe arbitrary cost metrics which are calculated by a resource-annotated semantics for a fragment of the Java Virtual Machine Language. The semantics is big-step operational formalisation which we take to define our fragment of the JVM and its resource behaviour; its correctness is self-evident and the resource annotations are in principle further justifiable by empirical evaluation of specific JVM behaviour.

The main part of the program logic is a VDM-style logic for partial correctness which, with respect to the annotated operational semantics, is provably sound and complete (relative to the underlying assertion language). Using the logic instead of the operational semantics directly has the added value of providing direct support for reasoning with invariants for recursive methods and loops (represented as tail recursive functions). To prove that a method satisfies some specification it suffices

to show that its body satisfies it assuming that any recursive call does so. Proving this principle sound for the operational semantics requires some effort which would otherwise have to be spent with each individual verification.

The program logic has two components: as well as the core logic of partial correctness, it has a logic for termination built on top, that is treated separately. The separation allows us to establish resource properties under the assumption of termination (partial correctness) following one approach, and establish termination using another approach and only where it is necessary (for example, on a case-by-case basis for individual methods).

The main part of the paper is concerned with the definition and meta-theoretical properties of the core logic and termination logic. Towards the end of the paper we will return to the original motivation of proof-carrying code, and describe how the logics can be fitted into a larger infrastructure, addressing the necessary engineering requirements.

This work represents a significant contribution to research in generalising proof-carrying code scenario to richer policies and richer languages; to our knowledge it represents one of the first applications of PCC for resources to Java bytecode. It also provides advances in program logics for low-level code; we believe that our logic has a number of advantages over many existing related approaches, not least including its good meta-theoretical properties, separate treatment of (partial) correctness and termination, and the functional treatment of iteration and local variables. The body of the paper and an extensive comparison at the end describes these and other advantages in detail. The entire infrastructure has been formalised in the theorem prover Isabelle/HOL, both to enhance confidence in the meta-theoretical results, and to provide a prototype implementation for PCC.

1.1 Outline

The structure of this paper is as follows. Section 2 presents the *Grail* language for which we build these logics. Grail is a subset of the Java Virtual Machine (JVM) language, written using a functional notation. The operational semantics of Grail is annotated with a mechanism for calculating resource costs, using operators for each expression former.

Section 3 presents the core VDM-style program logic for partial correctness for Grail. It allows an expressive form of assertion that facilitates compositional reasoning, and powerful rules for mutual recursion. Proofs of soundness and completeness for this logic are given, based on a full formalisation in Isabelle/HOL. Examples demonstrate the use of this logic for proving resource properties, making use of special rules for mutually recursive program fragments and method invoca-

tions with argument adaptation.

Section 4 builds a termination logic on top of the partial correctness logic, and thus arrives at a total correctness logic. For this termination logic proofs of soundness and completeness are given. An earlier example is picked up to demonstrate usefulness.

In Section 5 we return to the big picture and describe how our program logics are used in the *Mobile Resource Guarantees* architecture. Section 6 discusses related work on program logics and their theorem prover formalisations; finally, Section 7 summarises the main results of our work and sketches directions for further work.

2 The Grail language

Rather than working with JVML directly, we use a reformulation called the Guaranteed Resource Aware Intermediate Language (*Grail*). The version of Grail studied in this paper covers the most important features of the JVML, including boolean and integer values, reference values and objects, mutable fields (allowing aliasing), static and instance methods. We do not cover exceptions, arrays, or threads at this point.

The design of Grail was motivated by the following goals:

(1) Suitability as a target for a resource-transparent compiler from a functional high-level language;
(2) Proximity to standard mobile code formats, so that executability and code mobility using existing wire formats are obtained;
(3) Suitability as a basis for attaching resource assertions;
(4) Amenability to formal proofs about resource usage.

Grail uses a functional notation that abstracts from some JVM specific details such as the use of numerical variable names as locals, and it has the operational semantics of an impure functional language: mutable object fields, immutable local variables, and tail recursive functions instead of iteration. This makes it ideally suitable for compilation of high-level functional languages such as Camelot, which is a variant of OCaml with a resource-aware type system (4).

Code in Grail remains close to standard formats; it can be reversibly expanded into a subset of virtual machine languages including JVML used by Java, the JCVML of JAVACARD, or the MSIL used by .NET [1]

[1] At present, our tools cover only JVML.

Because the translation mechanism into a virtual machine language is fixed (5), the resource usage of Grail programs can be captured by considering the cost of execution of the translated form, and those costs can be used to instantiate a resource model which is part of the operational semantics.

Finally, we will demonstrate that Grail is amenable to formal proof by presenting our program logic and its implementation in Sections 3 and 4.

2.1 Syntax

At the level of classes and methods, the Grail representation of code retains the syntactic structure of Java bytecode. Method bodies, in contrast, are represented as collections of mutually tail-recursive A-normalised functions (6). In particular, only primitive operations (no let-expressions) can occur as e_1 in let-expressions `let` $x = e_1$ `in` e_2. Similar to the λ-JVM (7) and other functional intermediate languages (8), primitive instructions model arithmetic operations, object manipulation (object creation, field access) and the invocation of methods. Strengthening the syntactic conditions of ANF, actual arguments in function calls are required to coincide syntactically with the formal parameters of the function definitions. This condition allows function calls to be interpreted as immediate jump instructions; register shuffling at basic block boundaries is performed by the calling code rather than being built into the function application rule. As a result, the consumption of resources at virtual machine level may be expressed in a functional semantics for Grail: the expansion into bytecode does not require register allocation or the insertion of gluing code (9). In contrast to the aim of λ-JVM, Grail does not aim to represent arbitrary bytecode. Instead, the translation of raw bytecode into Grail is only defined on the image of our expansion – and on this subset, it is the reversal of the code expansion. As each Grail instruction form expands to a sequence of bytecode instructions that leaves the operand stack empty, the emitted code is highly structured. In particular, it satisfies all conditions identified by Leroy (10) as being required for efficient bytecode verification on Smartcards and other resource-constrained devices.

For the purpose of this paper, the syntactic restrictions of Grail may be largely ignored, since their significance concerns the relationship between Grail and actual bytecode rather than the program logic, whose correctness does not require them. The formal syntax treated in the remainder of this article therefore comprises a single category of expressions *expr* that is defined over mutually disjoint sets $\mathcal{M}$ of method names, $\mathcal{C}$ of class names, $\mathcal{F}$ of function names (i.e. labels of basic blocks), $\mathcal{T}$ of (static) field names and $\mathcal{X}$ of variables. These categories are, respectively, ranged over by m, c, f, t, and x. In addition, i ranges over immediate values (integers, and booleans `true` and `false`) and *op* over primitive operation of type $\mathcal{V} \Rightarrow \mathcal{V} \Rightarrow \mathcal{V}$ such as arithmetic operations or comparison operators, where $\mathcal{V}$ is the semantic category of values. Values are ranged over by v and comprise

immediate values, references r, and the special symbol $\perp$, which stands for the absence of a value. References are either null or of the form Ref l where $l \in \mathcal{L}$ is a location.

The syntax is given by the grammar below, which defines expressions *expr* and method arguments *args*. We write $\overline{a}$ to stand for lists of arguments.

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

This syntax is intentionally somewhat verbose, for example, by including the injections `var` and `imm` which might ordinarily be omitted using meta-syntactic conventions for names. We hope that the reader will forgive the additional verbosity in return for the reassurance that the following rules, programs and example verifications have all been formalised precisely as they are written here.

Expressions $e \in \mathit{expr}$ represent basic blocks and are built from operators, constants, and previously computed values (names). Expressions correspond to primitive sequences of bytecode instructions that may, as a side effect, alter the heap. For example, $x.t$ and $x.t{:=}y$ represent (non-static) `getfield` and `putfield` instructions, while $c \diamond t$ and $c \diamond t{:=}y$ denote their static counterparts. The binding $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is used if the evaluation of e_1 returns a value on top of the JVM stack while $e_1\ ;\ e_2$ represents purely sequential composition, used for example if e_1 is a field update $x.t{:=}y$. Object creation includes the initialisation of the object fields according to the argument list: the content of variable x_i is stored in field t_i. Function calls (`call`) follow the Grail calling convention (i.e. correspond to immediate jumps) and do not carry arguments. The instructions $x \cdot m(\overline{a})$ and $c.m(\overline{a})$ represent virtual (instance) and static method invocation, respectively.

A formal type system can be imposed on Grail programs to rule out bad programs by reflecting typing conditions enforced by the underlying virtual machine; our operational semantics and program logic are more general, although we always consider well-typed programs.

In the theoretical development, we assume that all method declarations employ distinct names for identifying inner basic blocks. A program P consists of a table FT mapping each function identifiers to an expression, and a table MT associating a list of method parameters (i.e. variables) and an expression (the initial basic block) to class names and method identifiers. We use the notations body_f and $\mathit{body}_{c,m}$ to denote the bodies of function f and method $c.m$, respectively, and $\mathit{pars}_{c,m}$ to denote the formal parameters of $c.m$. The variable `self` is a reserved name.

Figure 1 shows an example method for appending the list represented by argument

$$
\begin{array}{l}
\texttt{method}\ \mathsf{LIST}\ \mathsf{LIST.append}(\mathsf{l}_1,\mathsf{l}_2) = \texttt{call}\ f \\
\left[\begin{array}{ll}
f & \mapsto \texttt{let}\ \mathsf{v}_3 = \mathsf{l}_1.\mathsf{TAG}\ \texttt{in} \\
 & \quad \texttt{let}\ \mathsf{b} = \texttt{prim}\ (\lambda\, z\, y.\ \text{if } z = 2 \text{ then true else false})\ \mathsf{v}_3\ \mathsf{v}_3\ \texttt{in} \\
 & \quad \texttt{if}\ \mathsf{b}\ \texttt{then var}\ \mathsf{l}_2\ \texttt{else call}\ f_1 \\
f_1 & \mapsto \texttt{let}\ \mathsf{v}_3 = \mathsf{l}_1.\mathsf{HD}\ \texttt{in let}\ \mathsf{v}_2 = \mathsf{l}_1.\mathsf{TL}\ \texttt{in} \\
 & \quad \texttt{let}\ \mathsf{l}_1 = \mathsf{LIST.append}([\texttt{var}\ \mathsf{v}_2, \texttt{var}\ \mathsf{l}_2])\ \texttt{in let}\ \mathsf{tg} = \texttt{imm}\ 3\ \texttt{in} \\
 & \quad \texttt{new}\ \mathsf{LIST}\ [\mathsf{TAG} := \mathsf{tg}, \mathsf{HD} := \mathsf{v}_3; \mathsf{TL} := \mathsf{l}_1]
\end{array}\right]
\end{array}
$$

Fig. 1. Code of method append

l_2 to the list l_1. The code represents the pretty-printed output of a compiler that translates code written in the high-level functional language Camelot into Grail. It shows a method containing two functions whose bodies are shown; the body of the method itself is the term $\texttt{call}\ f$. [2]

2.2 Resource algebras

To admit reasoning about allocation and consumption of different computational resources, our operational semantics is annotated with a resource counting mechanism based on a general cost model provided by a notion of *resource algebra*. A resource algebra provides a basis for quantitative measurements such as (instruction) counters, but can also be used to monitor cost-relevant events like the allocation of fresh memory, calls to specific (native) methods, or the maximal height of the frame stack encountered during the execution of a program.

Definition 1 *A resource algebra $\mathcal{R}$ consists of a set R, together with operations:*

- $\mathcal{R}^{\texttt{null}}, \mathcal{R}^{\texttt{imm}}_{i}, \mathcal{R}^{\texttt{var}}_{x}, \mathcal{R}^{\texttt{prim}}_{op,x,y}, \mathcal{R}^{\texttt{getf}}_{x,t}, \mathcal{R}^{\texttt{putf}}_{x,t,y}, \mathcal{R}^{\texttt{gets}}_{c,t}, \mathcal{R}^{\texttt{puts}}_{c,t,y}, \mathcal{R}^{\texttt{new}}_{c,x_1,\ldots,x_n}$ *of type R, and*
- $\mathcal{R}^{\texttt{if}}_{x}, \mathcal{R}^{\texttt{call}}_{f}, \mathcal{R}^{\texttt{invs}}_{c,m,\bar{a}}$ *and* $\mathcal{R}^{\texttt{invv}}_{x,m,\bar{a}}$ *of type* $R \to R$*,*
- $\mathcal{R}^{\texttt{let}}_{x}, \mathcal{R}^{\texttt{comp}}$ *of type* $(R \times R) \to R$*.*

Resource algebras simply collect together constant costs and operations on costs for each expression former in the syntax. Our operational semantics and logics are then based on an arbitrary resource algebra. [3]

[2] In later examples, we do simplify notation somewhat, using some hopefully obvious shorthands from our implementation for standard primitive operations and omitting the `prim` tag, for example $\texttt{let}\ \mathsf{b} = \textit{iszero}\ \mathsf{v}_3\ \texttt{in}\ \ldots$.

[3] More can be said about the algebraic structure of resources and their operations and indeed more abstractly, somewhat in the spirit of (11; 12). This is pursued elsewhere (13).

An alternative way of cost accounting used elsewhere is by *instrumenting* the original code, i.e. by inserting additional program variables and instructions to maintain resource counters, but without otherwise changing the underlying program. Then the costs of executing the original program are obtained by reasoning about (or executing, or executing symbolically, or performing static analysis on) the instrumented program (14; 15; 16). Although highly flexible, instrumentation has certain disadvantages. Most obviously, if the instrumented code is executed at run-time, there will be extra costs associated in book-keeping and resource usage patterns themselves may change compared with the uninstrumented code (risking a change in program behaviour due to race conditions). Compared with static approaches, when run-time monitors are employed to enforce a security policy, we risk wasting computational resources by terminating offending computations. Fundamentally, instrumentation limits the costs one may consider to quantities where the domain and the modifying operations are expressible in the programming language, and the interaction between resource-counting and "proper" variables may become difficult to reason about. Our approach avoids these problems by including structured costs in the operational semantics; it retains flexibility by being parametrised on the form of the costs.

2.3 Resource algebra examples

A few motivating examples of resource algebras are collected in Figure 2.

The first algebra, $\mathcal{R}^{\mathsf{Count}}$, represents a simple cost model of the JVM with four metrics, and was already presented in (17). The first component of a tuple models an instruction counter that approximates execution time. Charging all JVM instructions at the same rate, this counter is incremented roughly by the number of bytecode instructions each expression may be expanded to. For example, a field modification $x.t{:=}y$ typically expands to a bytecode sequence of length three, comprising two load operations that copy the contents of variables x and y onto the operand stack, and one `putfield` operation. In the rules for method invocations, we charge for pushing all arguments onto the operand stack, the invocation, and the returning instruction, plus, in the case of a virtual invoke, for loading the object reference and the virtual method lookup. The second and third components represent more specific instruction counters, for function calls (jumps), and method invocations, respectively. A similar counter for the instruction `new` that would monitor the dynamic allocation of objects has been omitted since this information may be inferred from the operational semantics by comparing the (size of the) initial and final heaps. Naturally, more fine-grained instruction counters can easily be defined, for example by counting the invocation of different methods separately or by charging complex instructions at an increased rate. Finally, the fourth component monitors the maximal frame stack height observed during a computation. This value is nonzero exactly for the frame-allocating instructions $x \cdot m(\overline{a})$ and $c.m(\overline{a})$.

	$\mathcal{R}^{\mathsf{Count}}$	$\mathcal{R}^{\mathsf{InvTr}}$	$\mathcal{R}^{\mathsf{PVal}}(\mathsf{C},\mathsf{M},P)$	$\mathcal{R}^{\mathsf{HpTr}}$
R	$\mathcal{N}\times\mathcal{N}\times\mathcal{N}\times\mathcal{N}$	$\overline{expr}$	$\mathcal{B}$	$\overline{\mathcal{H}}$
$\mathcal{R}^{\mathtt{null}}$	$\langle 1\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{imm}}_{i}$	$\langle 1\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{var}}_{x}$	$\langle 1\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{prim}}_{op,x,y}$	$\langle 3\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{new}}_{c,x_1,\ldots,x_n}$	$\langle (n+1)\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{getf}}_{x,t}$	$\langle 2\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{putf}}_{x,t,y}$	$\langle 3\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{gets}}_{c,t}$	$\langle 2\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{puts}}_{c,t,y}$	$\langle 3\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{let}}_{x}(p,q)$	$\langle 1\,0\,0\,0\rangle \oplus (p \sqcap q)$	$p@q$	$p\wedge q$	$p@q$
$\mathcal{R}^{\mathtt{comp}}(p,q)$	$p\sqcap q$	$p@q$	$p\wedge q$	$p@q$
$\mathcal{R}^{\mathtt{if}}_{x}(p)$	$\langle 2\,0\,0\,0\rangle \oplus p$	p	true	p
$\mathcal{R}^{\mathtt{call}}_{f}(p)$	$\langle 1\,1\,0\,0\rangle \oplus p$	p	true	p
$\mathcal{R}^{\mathtt{invv}}_{x,m,\overline{a}}(p)$	$\langle (4+\lvert\overline{a}\rvert)\,0\,1\,1\rangle \oplus p$	$(x\cdot m(\overline{a}))::p$	p	p
$\mathcal{R}^{\mathtt{invs}}_{c,m,\overline{a}}(p)$	$\langle (2+\lvert\overline{a}\rvert)\,0\,1\,1\rangle \oplus p$	$(c.m(\overline{a}))::p$	$((c=\mathsf{C}\wedge m=\mathsf{M}) \longrightarrow P(\overline{a}))\wedge p$	p

Fig. 2. Four example resource algebras

The definition of $\mathcal{R}^{\mathsf{Count}}$ uses two binary operations, $\oplus$ and $\sqcap$, to combine costs. The former one represents pointwise addition in all four components, while the latter one performs pointwise addition in the first three components, and the *max* operation in the fourth component. Note that we could have formulated the counters as four separate resource algebras and obtained $\mathcal{R}^{\mathsf{Count}}$ as their product.

The second resource algebra, $\mathcal{R}^{\mathsf{InvTr}}$, ranges over expression sequences and collects the (static or virtual) method invocations in the (dynamic) program order. Here, the value for each axiom is the empty list; branches and function invocations take the value of sub-executions; invocations prefix the respective method calls to the expression sequence (notation :: in the meta-logic), and the operations for binary program composition $\mathcal{R}^{\mathtt{let}}_{x}$ and $\mathcal{R}^{\mathtt{comp}}$ append (denoted @) the sequences in the order of evaluation of the sub-expressions.

The last two resource algebras, $\mathcal{R}^{\mathsf{PVal}}(\mathsf{C},\mathsf{M},P)$ and $\mathcal{R}^{\mathsf{HpTr}}$, concern a slight gen-

eralisation of the formal setup, where, in addition to pieces of syntax, values of resource algebras may also depend upon other components of the operational semantics (to be defined shortly), such as environments E and heaps h. This allows us to formulate policies that depend on the dynamic evaluation of syntactic items. The *parameter values* algebra $\mathcal{R}^{\mathsf{PVal}}(\mathsf{C},\mathsf{M},P)$ can be parametrised by a class name C, a method name M, and a *parameter policy* P which may refer to the environment and heap. The resource value in this algebra maintains a flag which acts as a monitor to ensure that each invocation of $\mathsf{C}.\mathsf{M}(\overline{a})$ satisfies the policy $P(\overline{a})$. For example, for each $k \in \mathcal{N}$, the policy

$$P_k(\overline{a}) \equiv \exists\, x\, n\, X.\, \overline{a} = [\mathtt{var}\ x] \ \wedge\ h \models_{list(n,X)} E\langle x\rangle \ \wedge\ n \leq k$$

stipulates that $\overline{a}$ consists of a single variable that represents a list of length [4] at most k. Value-constraining policies like this may can be used in the domain of embedded systems, where calls to external actuators must obey strict parameter limitations. See (3) for more motivation and a detailed example.

In general, resource algebras that depend on semantic components are sufficiently powerful to collect diagnostic traces along the chosen path of computation – the final example, $\mathcal{R}^{\mathsf{HpTr}}$ simply collects all intermediate heaps (the category $\mathcal{H}$ of heaps will be defined shortly). Like a syntactic trace originating from the algebra $\mathcal{R}^{\mathsf{InvTr}}$, the resulting word may be constrained by further policies, specified for example by security automata (18) or formulae from logics over linear structures (which can be encoded in our higher-order assertion language).

When not considering specific examples, we understand $\mathcal{R}$ to represent a fixed, but arbitrary resource algebra, and identify $\mathcal{R}$ with its carrier set.

2.4 Operational Semantics

The formal basis of the program logic is a big-step operational semantics that models an (impure) functional interpretation of Grail. Judgements relate expressions e to environments $E \in \mathcal{E}$ (maps from variables to values), initial and final heaps $h, h' \in \mathcal{H}$, result values $v \in \mathcal{V}$ and costs $p \in \mathcal{R}$. Heaps consist of two components, an object heap and a class heap. The object heap is a finite map of type $\mathcal{L} \rightarrow_{fin} \mathcal{C} \times (\mathcal{T} \rightarrow_{fin} \mathcal{V})$, i.e. an object consists of a class name and a field table. The class heap stores the content of static fields and is represented by a map

[4] The formal definition of datatype representation predicates such as $h \models_{list(n,X)} E\langle x\rangle$ is postponed until Section 3.5.1.

$\mathcal{C} \rightarrow_{fin} \mathcal{T} \rightarrow_{fin} \mathcal{V}$. The following operations are used for heaps:

$h(l).t$	field lookup of value at t in object at l,
$h[l.t \mapsto v]$	field update of t with v at l,
$h(l)$	class name of object at l,
$dom\ h$	domain of the heap h,
$freshloc(h)$	a location l chosen so that $l \notin dom\ h$.

A judgement $E \vdash h, e \Downarrow h', v, p$ reads "in variable environment E and initial heap h, code e evaluates to the value v, yielding the heap h' and consuming p resources". The rules defining our semantics are formulated relative to a fixed program P, whose components are accessed in the rules for functional calls and method invocations. The following rules use further notation which is explained in the commentary below.

$$\frac{}{E \vdash h, \mathtt{null} \Downarrow h, \mathsf{null}, \mathcal{R}^{\mathtt{null}}} \text{(NULL)} \qquad \frac{}{E \vdash h, \mathtt{imm}\ i \Downarrow h, i, \mathcal{R}^{\mathtt{imm}}_{i}} \text{(IMM)}$$

$$\frac{}{E \vdash h, \mathtt{var}\ x \Downarrow h, E\langle x\rangle, \mathcal{R}^{\mathtt{var}}_{x}} \text{(VAR)}$$

$$\frac{}{E \vdash h, \mathtt{prim}\ op\ x\ y \Downarrow h, op\ (E\langle x\rangle)\ (E\langle y\rangle), \mathcal{R}^{\mathtt{prim}}_{op,x,y}} \text{(PRIM)}$$

$$\frac{E\langle x\rangle = \mathsf{Ref}\, l}{E \vdash h, x.t \Downarrow h, h(l).t, \mathcal{R}^{\mathtt{getf}}_{x,t}} \text{(GETF)} \qquad \frac{E\langle x\rangle = \mathsf{Ref}\, l}{E \vdash h, x.t{:=}y \Downarrow h[l.t \mapsto E\langle y\rangle], \bot, \mathcal{R}^{\mathtt{putf}}_{x,t,y}} \text{(PUTF)}$$

$$\frac{}{E \vdash h, c \diamond t \Downarrow h, h(c).t, \mathcal{R}^{\mathtt{gets}}_{c,t}} \text{(GETS)} \qquad \frac{}{E \vdash h, c \diamond t{:=}y \Downarrow h[c.t \mapsto E\langle y\rangle], \bot, \mathcal{R}^{\mathtt{puts}}_{c,t,y}} \text{(PUTS)}$$

$$\frac{l = freshloc(h)}{E \vdash h, \mathtt{new}\ c\ [\overline{t_i := x_i}] \Downarrow h[l \mapsto (c, \{\overline{t_i := E\langle x_i\rangle}\})], \mathsf{Ref}\, l, \mathcal{R}^{\mathtt{new}}_{c,x_1,\ldots,x_n}} \text{(NEW)}$$

$$\frac{E\langle x\rangle = \mathsf{true} \quad E \vdash h, e_1 \Downarrow h_1, v, p}{E \vdash h, \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow h_1, v, \mathcal{R}^{\mathtt{if}}_{x}(p)} \text{(IFTRUE)}$$

$$\frac{E\langle x\rangle = \mathsf{false} \quad E \vdash h, e_2 \Downarrow h_1, v, p}{E \vdash h, \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow h_1, v, \mathcal{R}^{\mathtt{if}}_{x}(p)} \text{(IFFALSE)}$$

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

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

$$\frac{E \vdash h, body_f \Downarrow h_1, v, p}{E \vdash h, \texttt{call}\ f \Downarrow h_1, v, \mathcal{R}_f^{\texttt{call}}(p)} \quad \text{(CALL)}$$

$$\frac{Env(\texttt{self} :: pars_{c,m}, \mathsf{null} :: \overline{a}, E) \vdash h, body_{c,m} \Downarrow h_1, v, p}{E \vdash h, c.m(\overline{a}) \Downarrow h_1, v, \mathcal{R}_{c,m,\overline{a}}^{\texttt{invs}}(p)} \quad \text{(SINV)}$$

$$\frac{E\langle x\rangle = \mathsf{Ref}\, l \quad h(l) = c \quad Env(\texttt{self} :: pars_{c,m}, x :: \overline{a}, E) \vdash h, body_{c,m} \Downarrow h_1, v, p}{E \vdash h, x \cdot m(\overline{a}) \Downarrow h_1, v, \mathcal{R}_{x,m,\overline{a}}^{\texttt{invv}}(p)} \quad \text{(VINV)}$$

The first nine rules are rather straightforward — the costs are simply obtained by applying the corresponding components of the resource algebra to the syntactic components of the operation in question. In rules PUTF and PUTS, the return values are set to $\bot$, in accordance with the fact that the corresponding virtual machine codes do not leave a value on the operand stack. In rule NEW, the allocated object is initialised by assigning the content of variable x_i to field t_i, for all fields.

The two rules for conditionals contain no surprises either — the resources consumed during the execution of the respective branch are promoted to the conclusion, adjusted by the operation $\mathcal{R}_x^{\texttt{if}}$. Similarly, the rules for program composition, LET and COMP, combine the costs of the constituent expressions.

In the rules CALL, SINV and VINV, the function and method bodies, and the method parameters, are retrieved from the (implicit) program P. As discussed earlier, function calls occur in tail position and correspond to jump instructions, hence the body is executed in the same environment. The costs of the function call are taken into account at the end of the execution. In contrast, method calls can occur in non-tail positions, hence the rules for static and virtual method invocations execute the method body in an unaltered heap, but in an environment that represents a new frame. The semantic function $Env(\overline{x}, \overline{a}, E)$ constructs a fresh environment that maps parameter x_i to the result of evaluating a_i in environment E, i.e. to $E\langle x\rangle$ if $a_i = x$ and to a_i otherwise. The content of the `self` variable is set to the location of the invoking object in the case of rule VINV, and to the null value in rule SINV.

3 Program logic for partial correctness

The basis for reasoning and certificate generation is a general-purpose program logic for Grail where assertions are boolean functions over all semantic components occurring in the operational semantics, i.e. evaluation environments, pre- and post-heaps, result values, and values from a resource algebra. In this section, we define a logic of partial correctness (i.e. in particular, non-terminating programs satisfy any assertion), which is complemented by a termination logic in Section 4.

3.1 Assertions and validity

Deviating from the syntactic separation into pre- and post-conditions typical for Hoare-style and VDM-style program logics (19; 20), a judgement in our logic relates a Grail expression e to a single assertion A

$$\Gamma \rhd e : A$$

dependent on a context

$$\Gamma = \{(e_1, A_1), \ldots, \{e_n, A_n)\}$$

that stores verification assumptions for recursive program structures. [5]

Following the so-called "shallow embedding" style, we encode assertions as predicates in the formal higher-order meta-logic. Assertions range over the components of the operational semantics, namely the input environment E and initial heap h, and the post heap h', the result value v, and the resources consumed p. An assertion A thus belongs to the type

$$\mathcal{A} \equiv \mathcal{E} \to \mathcal{H} \to \mathcal{H} \to \mathcal{V} \to \mathcal{R} \to \mathcal{B}$$

where $\mathcal{B}$ is the set of propositional booleans. We use the notation of Isabelle/HOL for writing logical connectives and predicates, in particular, using λ-notation to define predicates:

$$A = \lambda E\ h\ h'\ v\ p.\ \cdots$$

and curried function application to denote their application to particular semantic values:

$$A\ E_1\ h_1\ h'_1\ v_1\ p_1$$

in the rules of the logic, the conclusions define assertions which hold for each form of expression, by applying assertions from the premises to appropriately modified

[5] Later on we sometimes use the term "specification" as a synonym for "assertion", especially when referring to assumptions or assertions used to define behaviour exactly.

values corresponding to the operational semantics. Axioms define assertions which are satisfied exactly by the corresponding evaluation in the semantics.

Although this "mega" assertion format is unusual, it has advantages. Compared to program logics with pre- and post-conditions, a single assertion allows us to simplify the treatment of auxiliary variables and admits a formulation of the rule for program composition that avoids the modification of the precondition typical for Hoare-style logics. We discuss this further in Section 3.3.

The validity of assertion A for expression e is defined by a partial correctness interpretation: A must be satisfied for all terminating executions of e.

Definition 2 *(Validity) Assertion A is* valid *for e, written* $\models e : A$*, if*

$$E \vdash h, e \Downarrow h', v, p \qquad \textit{implies} \qquad A\ E\ h\ h'\ v\ p$$

for all E, h, h′, v, and p.

This definition may be lifted to contexts Γ in the obvious way.

Definition 3 *(Contextual validity) Context* Γ *is* valid*, notation* $\models \Gamma$*, if all pairs* (e, A) *in* Γ *satisfy* $\models e : A$*. Assertion A is* valid for e in context Γ*, written* $\Gamma \models e : A$*, if* $\models \Gamma$ *implies* $\models e : A$*.*

We next turn to the description of our proof system and the proof of its soundness and completeness for this notion of validity.

3.2 Proof system

The program logic comprises one rule for each expression form, and two logical rules, VAX and VCONSEQ. Again, we consider classes and methods for a fixed program P.

$$\frac{}{\Gamma \rhd \mathtt{null} : \lambda E\, h\, h'\, v\, p.\, h' = h \wedge v = \mathsf{null} \wedge p = \mathcal{R}^{\mathtt{null}}} \quad \text{(VNULL)}$$

$$\frac{}{\Gamma \rhd \mathtt{imm}\ i : \lambda E\, h\, h'\, v\, p.\, h' = h \wedge v = i \wedge p = \mathcal{R}^{\mathtt{imm}}_{i}} \quad \text{(VIMM)}$$

$$\frac{}{\Gamma \rhd \mathtt{var}\ x : \lambda E\, h\, h'\, v\, p.\, h' = h \wedge v = E\langle x \rangle \wedge p = \mathcal{R}^{\mathtt{var}}_{x}} \quad \text{(VVAR)}$$

$$\frac{}{\Gamma \rhd \texttt{prim}\ op\ x\ y : \lambda E\, h\, h'\, v\, p.\ v = op\ E\langle x\rangle\ E\langle y\rangle\ \wedge\ h' = h \wedge p = \mathcal{R}^{\texttt{prim}}_{op,x,y}} \quad \text{(VPRIM)}$$

$$\frac{}{\Gamma \rhd x.t : \lambda E\, h\, h'\, v\, p. \exists l.\ E\langle x\rangle = \mathsf{Ref}\, l \wedge h' = h\ \wedge v = h'(l).t\ \wedge p = \mathcal{R}^{\texttt{getf}}_{x,t}} \quad \text{(VGETF)}$$

$$\frac{}{\begin{array}{l}\Gamma \rhd x.t{:=}y : \lambda E\, h\, h'\, v\, p. \exists l.\ E\langle x\rangle = \mathsf{Ref}\, l \wedge p = \mathcal{R}^{\texttt{putf}}_{x,t,y} \wedge \\ \qquad\qquad h' = h[l.t \mapsto E\langle y\rangle]\ \wedge v = \bot\end{array}} \quad \text{(VPUTF)}$$

$$\frac{}{\Gamma \rhd c \diamond t : \lambda E\, h\, h'\, v\, p. h' = h\ \wedge v = h(c).t\ \wedge p = \mathcal{R}^{\texttt{gets}}_{c,t}} \quad \text{(VGETST)}$$

$$\frac{}{\Gamma \rhd c \diamond t{:=}y : \lambda E\, h\, h'\, v\, p. h' = h[c.t \mapsto E\langle y\rangle]\ \wedge v = \bot \wedge p = \mathcal{R}^{\texttt{puts}}_{c,t,y}} \quad \text{(VPUTST)}$$

$$\frac{}{\begin{array}{l}\Gamma \rhd \texttt{new}\ c\ \overline{[t_i := x_i]} : \lambda E\, h\, h'\, v\, p. \exists l.\ l = \mathit{freshloc}(h) \wedge p = \mathcal{R}^{\texttt{new}}_{c,x_1,\ldots,x_n} \wedge \\ \qquad\qquad h' = h[l \mapsto (c, \{\overline{t_i := E\langle x_i\rangle}\})]\ \wedge\ v = \mathsf{Ref}\, l\end{array}} \quad \text{(VNEW)}$$

$$\frac{\Gamma \rhd e_1 : A_1 \qquad \Gamma \rhd e_2 : A_2}{\begin{array}{l}\Gamma \rhd \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \lambda E\, h\, h'\, v\, p. \exists p'.\ p = \mathcal{R}^{\texttt{if}}_{x}(p') \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{true} \longrightarrow A_1\ E\ h\ h'\ v\ p') \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{false} \longrightarrow A_2\ E\ h\ h'\ v\ p') \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{true} \vee E\langle x\rangle = \mathsf{false})\end{array}} \quad \text{(VIF)}$$

$$\frac{\Gamma \rhd e_1 : A \qquad \Gamma \rhd e_2 : B}{\begin{array}{l}\Gamma \rhd \texttt{let}\ x{=}e_1\ \texttt{in}\ e_2 : \lambda E\, h\, h'\, v\, p. \exists\ p_1\ p_2\ h_1\ w.\ A\ E\ h\ h_1\ w\ p_1\ \wedge\ w \neq \bot \wedge \\ \qquad\qquad B\ (E\langle x := w\rangle)\ h_1\ h'\ v\ p_2) \wedge \\ \qquad\qquad p = \mathcal{R}^{\texttt{let}}_{x}(p_1, p_2)\end{array}} \quad \text{(VLET)}$$

$$\frac{\Gamma \rhd e_1 : A \qquad \Gamma \rhd e_2 : B}{\Gamma \rhd e_1 \,;\, e_2 : \lambda E\, h\, h'\, v\, p.\, \exists\, p_1\, p_2\, h_1.\, A\ E\ h\ h_1 \perp p_1\ \wedge\ B\ E\ h_1\ h'\ v\ p_2\ \wedge\ p = \mathcal{R}^{\mathtt{comp}}(p_1, p_2)} \quad \text{(VCOMP)}$$

$$\frac{\Gamma \cup \{(\mathtt{call}\, f, A)\} \rhd body_f : \Theta(A, f)}{\Gamma \rhd \mathtt{call}\, f : A} \quad \text{(VCALL)}$$

$$\frac{\Gamma \cup \{(c.m(\overline{a}), A)\} \rhd body_{c,m} : \Phi(A, c, m, \overline{a})}{\Gamma \rhd c.m(\overline{a}) : A} \quad \text{(VSINV)}$$

$$\frac{\Gamma \cup \{x \cdot m(\overline{a}), A)\} \rhd body_{c,m} : \Psi(A, x, c, m, \overline{a})}{\Gamma \rhd x \cdot m(\overline{a}) : A} \quad \text{(VVINV)}$$

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

The rules for function calls and method invocations make use of the following operators that model the effect of frame creation and the application of the resource-algebraic operations:

$$\Theta(A, f) = \lambda E\, h\, h'\, v\, p.\, A\ E\ h\ h'\ v\ (\mathcal{R}_f^{\mathtt{call}}(p)),$$

$$\Phi(A, c, m, \overline{a}) = \lambda E\, h\, h'\, v\, p.\ \forall\, E'.\, E = Env(\mathtt{self} :: pars_{c,m}, \mathsf{null} :: \overline{a}, E') \longrightarrow A\ E'\ h\ h'\ v\ (\mathcal{R}^{\mathtt{invs}}_{c,m,\overline{a}}(p))$$

$$\Psi(A, x, c, m, \overline{a}) = \lambda E\, h\, h'\, v\, p.\ \forall\, E'\ l. \left(\begin{array}{l} (E'\langle x \rangle = \mathsf{Ref}\, l \wedge h(l) = c\ \wedge \\ E = Env(\mathtt{self} :: pars_{c,m}, x :: \overline{a}, E') \end{array} \right) \longrightarrow A\ E'\ h\ h'\ v\ (\mathcal{R}^{\mathtt{invv}}_{x,m,\overline{a}}(p)).$$

3.3 Discussion

The axioms (VNULL to VNEW) directly model the corresponding rules in the operational semantics, with constants for the resource tuples. The VIF rule uses the appropriate assertion based on the boolean value in the variable x. Since the evaluation of the branch condition does not modify the heap we only existentially quantify

over the cost component p'. In contrast, rule VLET existentially quantifies over the result value w, the heap h_1 resulting from evaluating e_1, and the resources from e_1 and e_2. Apart from the absence of environment update, rule VCOMP is similar to VLET.

The rules for recursive functions and methods involve the context and generalise Hoare's original rule for parameterless recursive procedures. They require one to prove that the bodies satisfy assertions that are related to the concluding assertions A in a way that is compatible with the relationship between the hypothetical and the concluding judgements of the operational rules CALL, SINV and VINV. In rule VCALL, this compatibility condition only affects the resources, as the operational rule CALL leaves the environment, the heaps, and the result value untouched. Thus, the definition of Θ merely applies the $\mathcal{R}^{\texttt{call}}$ operator to the resources consumed by the body of f. In the case of VSINV and VVINV, the construction of a new frame in the operational rules corresponds to the universal quantification over the environment associated with the *caller*, E', in the definitions of operators Φ and Ψ. In both cases, the environment associated with the body, E, arises from this outer environment E' by the $Env(\_,\_,\_)$ function. Again, the costs of the method call are applied by requiring that the body satisfies a assertion whose resource component makes A true after the application of the appropriate operator from $\mathcal{R}$. As is the case in VCALL, the verification of the method bodies proceeds in contexts that extend Γ by the yet-to-be-proven tuple. Recursive calls or invocations may thus access the stipulated assertion via rule VAX.

The VCONSEQ consequence rule derives an assertion B that follows from another assertion A in the meta-logic HOL.

The rules for program composition, VLET and VCOMP, relate to the earlier discussion on the format of assertions. In Hoare-style program logics, the purpose of auxiliary variables is to link pre- and post-conditions by "freezing" the values of (program or other) variables in the initial state, so that they can be referred to in the post-condition. Formally, auxiliary variables need to be universally quantified in the interpretation of judgements in order to treat variables of arbitrary domain, and their interaction with the rule of consequence. This quantification may either happen explicitly at the object level or implicitly, where pre- and post-condition are predicates over pairs of states and the domain of auxiliary variables. Kleymann (21) showed that by instantiating the domain of auxiliary variables to the category of states one may embed VDM-style program logics in Hoare-style logics, and he studied meta-theoretic issues of both styles of logic. In particular, VDM's characteristic feature that allows the initial values of program variables to be referred to in the post-condition was modelled by giving both assertion components different types: pre-conditions are predicates on (initial) states while post-conditions are binary relations on states.

Kleymann's proposal to instantiate the domain of auxiliary variables to states also

allows one to employ further quantification in the definition of assertions. Such additional quantification typically ranges over semantic entities rather than program variables. As an example, consider an object-oriented language of terminating commands c, and a predicate $classOf(s,x,A)$ that is satisfied if the object pointed to by variable x in state s is of class A. A Kleymann-style judgement

$$\{\lambda\, Z\, s.\ Z = s\}c\{\lambda\, Z\, t.\ \forall\, A.\ classOf(Z,x,A) \rightarrow classOf(t,x,A)\}. \tag{1}$$

states that the class of object x remains unaffected by program c, with quantification occurring inside the post-condition. This formulation is highly preferable to an alternative formulation where the domain of auxiliary variables is chosen to be the category of class names. In the latter system, the above property would be expressed by a judgement like

$$\{\lambda\, A\, s.\ classOf(s,x,A)\}c\{\lambda\, A\, t.\ classOf(t,x,A)\}. \tag{2}$$

As a consequence, the type of assertions conceptually depends on the property one wishes to prove – an immediate obstacle to a compositionality if other program fragments require different instantiations.

Our approach of combining pre- and post-conditions enforces Kleymann's discipline. Were we adaptate our assertion format to an imperative language, the above property would be expressed as:

$$\Gamma \rhd c : \lambda\, s\, t.\ \forall\, A.\ classOf(s,x,A) \rightarrow classOf(t,x,A),$$

which uses the same inner quantification as (1) but avoids the auxiliary variable Z, similarly to a formulation in VDM.

A further difference to Hoare's calculus occurs in VLET. This rule combines rules for variable assignment and sequential program composition (the latter one corresponding more closely to rule VCOMP), motivated by the syntactic structure of Grail and its environment-based operational semantics. In contrast to Hoare's assignment rule

$$\frac{}{\{A[e/x]\}x := e\{A\}}$$

VLET does not involve any syntactic manipulation of formulae. In particular, the update of the environment (in an imperative setting: the assignment to the store) is modelled in exactly the same way as in the operational semantics, avoiding the (at first sight) counter-intuitive syntactic substitution in Hoare's rule. Partially, this difference originates from the shallowness of our embedding, where assertions are functions over states. Indeed, Nipkow's formalisation (see e.g. (22)) contains the following rule for basic instructions

$$\frac{}{\{\lambda\, s.\ A(f\ s)\}Do\ f\{A\}}\ ,$$

which yields

$$\overline{\{\lambda\, s.\, A(s[x \mapsto \mathit{eval}\ e\ s])\}x := e\{A\}}$$

when specialised to assignments by modelling $x := e$ as $\mathit{Do}\ (\lambda s.s[x \mapsto \mathit{eval}\ e\ s])$. However, while the substitution is indeed replaced by an application of the assertion to an updated state, this modification still happens in the pre-condition – Nipkow's rule is thus arguably closer to weakest precondition calculi (23). Finally, when combining judgements of sub-expressions, notice how our rules existentially quantify over intermediate states, while Hoare's rule for program composition

$$\frac{\{A\}c_1\{B\} \quad \{B\}c_2\{C\}}{\{A\}c_1;c_2\{C\}}$$

quantifies over intermediate assertions. Once again, our rule more closely related to the corresponding rule in VDM:

$$\frac{\{A\}c_1\{\lambda\, s\, t.\, B\, t \wedge C\, s\, t\} \quad \{B\}c_2\{D\}}{\{A\}c_1;c_2\{D \circ C\}}\ ,$$

which uses relational composition to combine post conditions.

3.4 Admissible rules

In addition to the proof rules given Section 3.2, we need some rules to simplify reasoning about concrete programs, and some others to help establish completeness. All of these rules involve the context of assumptions.

$$\frac{\Gamma \rhd e : A}{\Gamma \cup \Delta \rhd e : A} \qquad \text{(VWEAK)}$$

$$\frac{\Gamma \rhd e : A \quad \forall\, d\, B.\, (d,B) \in \Gamma \longrightarrow \Delta \rhd d : B}{\Delta \rhd e : A} \qquad \text{(VCTXT)}$$

$$\frac{\{(d,B)\} \cup \Gamma \rhd e : A \quad \Gamma \rhd d : B}{\Gamma \rhd e : A} \qquad \text{(VCUT)}$$

$$\frac{\Gamma \models ST \quad (e,A) \in \Gamma}{\emptyset \rhd e : A} \qquad \text{(MUTREC)}$$

$$\frac{\Gamma \models ST \quad (c.m(\overline{a}), ST(c,m,\overline{a})) \in \Gamma}{\emptyset \rhd c.m(\overline{b}) : ST(c,m,\overline{b})} \qquad \text{(ADAPTS)}$$

$$\frac{\Gamma \models ST \quad (x \cdot m(\overline{a}), ST(x,m,\overline{a})) \in \Gamma}{\emptyset \rhd y \cdot m(\overline{b}) : ST(y,m,\overline{b})} \qquad \text{(ADAPTV)}$$

The first two rules, VWEAK and VCTXT, are proven by an induction on the derivation of $\Gamma \rhd e : A$. A further cut rule, VCUT, follows easily from VCTXT. The cut rules eliminate the need for introducing a second form of judgement used previously in the literature (e.g., (22)) when establishing soundness of the rule MUTREC for mutually recursive program fragments. This proof will now be outlined.

First, we introduce the concept of *specification tables*. These associate an assertion A to each function name or method invocation.

Definition 4 *A* specification table *ST consists of the functions $FST : \mathcal{F} \to \mathcal{A}$, $sMST : \mathcal{C} \to \mathcal{M} \to \overline{args} \to \mathcal{A}$, and $vMST : \mathcal{X} \to \mathcal{M} \to \overline{args} \to \mathcal{A}$, where $\overline{args}$ is the type of argument lists. We write $ST(f)$, $ST(c,m,\overline{a})$ and $ST(x,m,\overline{a})$ for the respective access operations.*

Contexts whose entries arise uniformly from these specification tables are of particular interest.

Definition 5 *Context* Γ respects *specification table ST, notation $\Gamma \models ST$, if all $(e,A) \in \Gamma$ satisfy one of the three following conditions*

- $(e,A) = (\texttt{call}\, f, ST(f))$ *for some f with* $\Gamma \rhd body_f : \Theta(ST(f),f)$
- $(e,A) = (c.m(\overline{a}), ST(c,m,\overline{a}))$ *for some c, m and $\overline{a}$, and all $\overline{b}$ satisfy*

$$\Gamma \rhd body_{c,m} : \Phi(ST(c,m,\overline{b}),c,m,\overline{b}),$$

- $(e,A) = (x \cdot m(\overline{a}), ST(x,m,\overline{a}))$ *for some x, m and $\overline{a}$, and all c, y, and $\overline{b}$ satisfy*

$$\Gamma \rhd body_{c,m} : \Psi(ST(y,m,\overline{b}),y,c,m,\overline{b}).$$

Here, the operators Θ, Φ, and Ψ are those defined in Section 3.2. Using rule VCUT, is not difficult to prove that this property is closed under sub-contexts.

Lemma 6 *If $(e,A) \cup \Gamma \models ST$ then $\Gamma \models ST$.*

Based on Lemma 6, rule MUTREC can be proven by induction on the size of Γ. Notice that the conclusion relates e to A in the empty proof context – and thus, by rule VWEAK, in *any* context.

The remaining admissible rules ADAPTS and ADAPTV amount to variations of MUTREC for method invocations. In these rules, the expression in the conclusion may syntactically differ from the expression stored the context, as long as this difference occurs only in the method arguments (including the object on which a virtual method is invoked) and is reflected in the assertions. In Hoare logics, the adaptation of method specifications is related to the adaptation of *auxiliary variables*, a historically tricky issue in formal understandings of program logics (24; 22; 21). For example, Nipkow (22) adapts auxiliary variables in the rule of consequence, which allows him to adapt them also when accessing method specifications from contexts. In addition to admitting such an adaptation using universal quantification in the definition of method specifications (see the discussion in the previous section), our rules also allow differences in *syntactic* components, the method arguments.

In order to show ADAPTS sound, we first prove

Lemma 7 *If* $\Gamma \models ST$ *and* $(c.m(\overline{a}), ST(c,m,\overline{a})) \in \Gamma$ *then*

$$\Gamma \setminus (c.m(\overline{a}), ST(c,m,\overline{a})) \rhd c.m(\overline{b}) : ST(c,m,\overline{b}).$$

using rule VCUT. The conclusion in this lemma already involves method arguments, $\overline{b}$, that may be different from the arguments used in the context, $\overline{a}$. From this, rule ADAPTS follows by repeated application of Lemma 6. The proof of rule ADAPTV is similar.

3.5 Verification examples

Before treating the meta-logical issues of soundness and completeness, we first describe a number of verification examples which we have carried out using our implementation of the program logic in Isabelle/HOL. These examples show how to use the argument adaptation rules for method invocations, and how to specify and verify properties of resource consumption using resource algebras. We first outline our verification strategy.

Given a specification table ST, the verification of methods proceeds in groups of strongly connected components (SCCs) in the call graph, in topological order. For simplicity, suppose for the moment that we have only static methods. When verifying SCC N, all methods $c.m$ invoked in the bodies of methods in N are either in the same SCC, or are members of a smaller SCC and have already been verified, i.e.

$$\forall \overline{a}.\ \emptyset \rhd c.m(\overline{a}) : ST(c,m,\overline{a}) \tag{3}$$

has already been established. Supposing that N contains methods $c_1.m_1, \ldots, c_n.m_n$

with bodies $e_1, \ldots, e_n$, respectively, the verification of N proceeds in three stages. First, we define a context Γ_N that contains

- at least all entries $(c.m(\overline{a}), ST(c,m,\overline{a}))$ where $c.m(\overline{a})$ occurs in at least one body e_i, and $c.m$ is not in any SCC $M < N$, and
- at least one entry $(c_i.m_i(\overline{a}_i), ST(c_i,m_i,\overline{a}_i))$ for each $i \in \{1,\ldots,n\}$, where the $\overline{a}_i$ are distinct meta-variables.

Second, for each pair (c,m) for which there exists an $\overline{a}$ with $(c.m(\overline{a}), ST(c,m,\overline{a})) \in \Gamma_N$, we prove the lemma

$$\forall \overline{b}.\ \Gamma_N \rhd body_{c,m} : \Phi(ST(c,m,\overline{b}), c, m, \overline{b}).$$

In the proofs of these lemmas, all invocations of methods in N are verified using rule VAX, and all remaining method invocations are verified from (3) using VWEAK. Together, these lemmas yield a verification of $\Gamma_N \models ST$ in which each method body has been verified only once.

In the third stage, from $\Gamma_N \models ST$ we obtain

$$\forall \overline{a}.\ \emptyset \rhd c_i.m_i(\overline{a}) : ST(c_i, m_i, \overline{a})$$

by rule ADAPTS for all i, and SCC N has been verified.

3.5.1 Append

Our first example is for the method LIST.append shown in Figure 1. The property we will prove is given in the specification table entry

$$\begin{array}{l} ST(\mathsf{LIST}, \mathsf{append}, \overline{a}) = \lambda E\, h\, h'\, v\, p. \\ \quad \forall\, Y\ n\ m.\ \left(\begin{array}{c} \exists\, X\ x\ y.\ \overline{a} = [\mathtt{var}\ x, \mathtt{var}\ y] \wedge h \models_{list(n,X)} E\langle x \rangle \ \wedge \\ h \models_{list(m,Y)} E\langle y \rangle \ \wedge\ X \cap Y = \emptyset \end{array} \right) \longrightarrow \\ \qquad (\exists\, Z.\ h' \models_{list(n+m,Z)} v \ \wedge\ Z \cap dom(h) = Y \wedge\ h =_{dom(h)} h'). \end{array}$$

which asserts that the result v represents a list of length $n+m$ in the final heap, provided that the arguments represent non-overlapping lists of length n and m in the initial heap, respectively.

The datatype representation predicate $h \models_{list(n,X)} v$ is defined by the rules:

$$\frac{h(l) = \mathsf{LIST} \quad h(l).\mathsf{TAG} = 2}{h \models_{list(0,\{l\})} \mathsf{Ref}\, l} \qquad \frac{\begin{array}{ll} h(l) = \mathsf{LIST} & l \notin X \\ h(l).\mathsf{TAG} \neq 2 & h \models_{list(n,X)} h(l).\mathsf{TL} \end{array}}{h \models_{list(n+1,X\cup\{l\})} \mathsf{Ref}\, l}\ .$$

This predicate captures that the heap h contains a well laid-out (non-overlapping) list of length n begining at location value $v = \mathsf{Ref}\, l$, and occupying locations X. Predicates such as this are generated from high-level datatype definitions which are translated into class and field structures for representation on the virtual machine.

The program proceeds by induction on the first argument, and allocates fresh memory when constructing the result. The region inhabited by the result, Z, thus overlaps with the region Y of the second argument, but not the region X of the first argument. Furthermore, the content of all locations in $dom(h)$ remains unchanged (equality $h =_{dom(h)} h'$). By universally quantifying over arguments x and y, the property is uniform in the choice of argument names.

We verify that

$$\forall \overline{a}.\ \emptyset \rhd \mathsf{LIST.append}(\overline{a}) : ST(\mathsf{LIST}, \mathsf{append}, \overline{a}) \tag{4}$$

holds following the strategy outlined above.

In the first step, the smallest context Γ_{append} satisfying the conditions for append's SCC is the singleton context

$$\Gamma_{\mathsf{append}} = \{(\mathsf{LIST.append}([\mathtt{var}\ \mathsf{v}_2, \mathtt{var}\ \mathsf{l}_2]), ST(\mathsf{LIST}, \mathsf{append}, [\mathtt{var}\ \mathsf{v}_2, \mathtt{var}\ \mathsf{l}_2]))\},$$

since the only invocation in the body of append is $\mathsf{LIST.append}([\mathtt{var}\ \mathsf{v}_2, \mathtt{var}\ \mathsf{l}_2])$. The second step consists of proving the lemma

$$\forall \overline{b}.\ \Gamma_{\mathsf{append}} \rhd body_{\mathsf{LIST,append}} : \Phi(ST(\mathsf{LIST}, \mathsf{append}, \overline{b}), \mathsf{LIST}, \mathsf{append}, \overline{b}), \tag{5}$$

by applying the syntax-directed proof rules automatically and using rule VAX at the invocation of $\mathsf{LIST.append}([\mathtt{var}\ \mathsf{v}_2, \mathtt{var}\ \mathsf{l}_2])$. The two remaining side conditions (one for each branch) may be discharged by case analysis on the data type representation predicate, instantiating quantifiers, and applying datatype preservation results such as

$$(h \models_{list(n,X)} v \wedge h =_X h') \longrightarrow h' \models_{list(n,X)} v$$

which are themselves proven by induction on $h \models_{list(n,X)} v$. The proof that the side conditions are fulfilled is thus, in general, difficult to automate.

From the lemma (5) we obtain immediately $\Gamma_{\mathsf{append}} \models ST$, so the correctness statement (4) follows using rule ADAPTS.

Although statement (4) proves the correctness of $\mathsf{LIST.append}(\overline{a})$ for arbitrary $\overline{a}$, the definition of $ST(\mathsf{LIST}, \mathsf{append}, \overline{a})$ implies that useful assertions only arise for cases where $\overline{a}$ is an argument list of length two. In other cases, the formula to the

left of the implication is false, resulting in the trivial assertion $\lambda E\, h\, h'\, v\, p.\, \mathsf{true}$ that is fulfilled by any program but hardly useful at any point at which append is invoked.

The specification of append may be refined to include quantitative aspects using the resource algebras. For example, we can verify that all four metrics that make up $\mathcal{R}^{\mathsf{Count}}$ depend linearly on the length of the list represented by the first argument. First, we define linear factors $AppTimeF, \ldots, AppHeapF$ and constants $AppTimeC, \ldots, AppHeapC$.

$AppTimeF$	35	$AppTimeC$	14
$AppCallF$	2	$AppCallC$	1
$AppInvF$	1	$AppInvC$	1
$AppStackF$	1	$AppStackC$	1
$AppHeapF$	1	$AppHeapC$	0

Next, we modify the above specification table entry to include a specification of the resource component and a term relating the size of the final heap to that of the initial heap.

$$
\begin{aligned}
&ST(\mathsf{LIST}, \mathsf{append}, \overline{a}) = \lambda E\, h\, h'\, v\, p.\\
&\quad \forall\, Y\ n\ m. \left(\begin{array}{c} \exists\, X\ x\ y.\ \overline{a} = [\mathtt{var}\ x, \mathtt{var}\ y] \wedge h \models_{list(n,X)} E\langle x \rangle \wedge \\ h \models_{list(m,Y)} E\langle y \rangle \ \wedge\ X \cap Y = \emptyset \end{array} \right) \longrightarrow \\
&\qquad \left(\begin{array}{l} \exists\, Z.\ h' \models_{list(n+m,Z)} v \ \wedge\ Z \cap dom(h) = Y \wedge\ h =_{dom(h)} h' \wedge \\ \qquad p = \langle\ (AppTimeF * n + AppTimeC) \\ \qquad\qquad (AppCallF * n + AppCallC) \\ \qquad\qquad (AppInvF * n + AppInvC) \\ \qquad\qquad (AppStackF * n + AppStackC)\ \rangle \wedge \\ \qquad |dom(h')| = |dom(h)| + AppHeapF * n + AppHeapC \end{array} \right)
\end{aligned}
$$

Finally, the verification of

$$\forall \overline{a}.\ \emptyset \rhd \mathsf{LIST.append}(\overline{a}) : ST(\mathsf{LIST}, \mathsf{append}, \overline{a})$$

with respect to this modified specification is structurally identical to the proof of property (4).

3.5.2 *Flatten*

To continue with another example program emitted by the Camelot compiler, Figure 3 shows the definition of a method that flattens a tree into a list.

$$
\begin{array}{l}
\mathit{method}\ \mathsf{LIST\ TREE.flatten(t)} = \mathtt{call}\ f \\
\left[\begin{array}{ll}
f & \mapsto \mathtt{let}\ \mathsf{v_4} = \mathsf{t.TAG}\ \mathtt{in} \\
 & \quad \mathtt{let}\ \mathsf{b} = \mathit{iszero}\ \mathsf{v_4}\ \mathtt{in} \\
 & \quad \mathtt{if}\ \mathsf{b}\ \mathtt{then}\ \mathtt{call}\ f_0\ \mathtt{else}\ \mathtt{call}\ f_1 \\
f_0 & \mapsto \mathtt{let}\ \mathsf{v_4} = \mathsf{t.CONT}\ \mathtt{in}\ \mathtt{let}\ \mathsf{tg} = \mathtt{imm}\ 2\ \mathtt{in} \\
 & \quad \mathtt{let}\ \mathsf{t} = \mathtt{new}\ \mathsf{LIST}\ [\mathsf{TAG} := \mathsf{tg}]\ \mathtt{in}\ \mathtt{let}\ \mathsf{tg} = \mathtt{imm}\ 3\ \mathtt{in} \\
 & \quad \mathtt{new}\ \mathsf{LIST}\ [\mathsf{TAG} := \mathsf{tg}; \mathsf{HD} := \mathsf{v_4}; \mathsf{TL} := \mathsf{t}] \\
f_1 & \mapsto \mathtt{let}\ \mathsf{v_3} = \mathsf{t.LEFT}\ \mathtt{in}\ \mathtt{let}\ \mathsf{v_2} = \mathsf{t.RIGHT}\ \mathtt{in}\ \mathtt{let}\ \mathsf{v_1} = \mathsf{TREE.flatten}(\mathtt{var}\ \mathsf{v_3})\ \mathtt{in} \\
 & \quad \mathtt{let}\ \mathsf{t} = \mathsf{TREE.flatten}(\mathtt{var}\ \mathsf{v_2})\ \mathtt{in}\ \mathsf{LIST.append}([\mathtt{var}\ \mathsf{v_1}, \mathtt{var}\ \mathsf{t}])
\end{array}\right]
\end{array}
$$

Fig. 3. Code of method flatten

Again, we define a specification table entry for flatten,

$$
\begin{array}{l}
ST(\mathsf{TREE}, \mathsf{flatten}, \overline{a}) = \lambda E\, h\, h'\, v\, p. \\
\quad \forall\, n\, x.\ (\exists\, X.\ \overline{a} = [\mathtt{var}\ x]\ \wedge\ h \models_{\mathit{tree}(n,X)} E\langle x \rangle) \longrightarrow \\
\qquad (\exists\, Z.\ h' \models_{\mathit{list}(2^n,Z)} v\ \wedge\ Z \cap \mathit{dom}(h) = \emptyset\ \wedge\ h =_{\mathit{dom}(h)} h')
\end{array}
$$

which universally quantifies over the argument name x. It asserts that the result v represents a list of length 2^n in the final heap, provided that the argument represents a balanced binary tree of height n in the initial heap. Moreover, the region inhabited by the result, Z, does not overlap with h (i.e. the list is represented in freshly allocated memory), and the content of all locations in $\mathit{dom}(h)$ remains unchanged. The datatype representation predicate $h \models_{\mathit{tree}(n,X)} v$ is defined in a similar way as the list predicate $h \models_{\mathit{list}(n,X)} v$, namely:

$$
\frac{h(l) = \mathsf{TREE} \quad h(l).\mathsf{TAG} = 0}{h \models_{\mathit{tree}(0,\{l\})} \mathsf{Ref}\ l}
\qquad
\frac{\begin{array}{ll} h(l) = \mathsf{TREE} & l \notin L \cup R \\ h(l).\mathsf{TAG} \neq 0 & h \models_{\mathit{tree}(n,L)} h(l).\mathsf{LEFT} \\ L \cap R = \emptyset & h \models_{\mathit{tree}(n,R)} h(l).\mathsf{RIGHT} \end{array}}{h \models_{\mathit{tree}(n+1,L \cup R \cup \{l\})} \mathsf{Ref}\ l}
$$

Once more, following the prescribed verification strategy, we prove

$$\forall \overline{a}.\ \emptyset \rhd \mathsf{TREE.flatten}(\overline{a}) : ST(\mathsf{TREE}, \mathsf{flatten}, \overline{a}). \tag{6}$$

Building on the verification of append, the context defined in the first step may be chosen as

$$\Gamma_{\mathsf{flatten}} = \left\{ \begin{array}{l} (\mathsf{TREE.flatten}([\mathtt{var}\ \mathsf{v}_2]), ST(\mathsf{TREE}, \mathsf{flatten}, [\mathtt{var}\ \mathsf{v}_2])), \\ (\mathsf{TREE.flatten}([\mathtt{var}\ \mathsf{v}_3]), ST(\mathsf{TREE}, \mathsf{flatten}, [\mathtt{var}\ \mathsf{v}_3])) \end{array} \right\}.$$

In the verification of

$$\forall \overline{b}.\ \Gamma_{\mathsf{flatten}} \rhd \mathit{body}_{\mathsf{TREE,flatten}} : \Phi(ST(\mathsf{TREE}, \mathsf{flatten}, \overline{b}), \mathsf{TREE}, \mathsf{flatten}, \overline{b}), \tag{7}$$

the two invocations of flatten are discharged by rule VAX, while the invocation of append is discharged by appealing to property (4) using VWEAK. Once more, the proofs of the side conditions involve case analysis on the datatype representation predicates, the instantiation of quantifiers, and the application of datatype preservation lemmas for trees and lists. The preconditions of the latter are satisfied thanks to the separation conditions in the specifications of append and flatten.

Similarly to the verification of append, we obtain $\Gamma_{\mathsf{flatten}} \models ST$ from the result (7), and the correctness of flatten, i.e property (6), follows using rule ADAPTS. As before, the specification of flatten is non-trivial for argument lists of the right shape, in this case argument lists of length one.

To verify resource consumption for this method using $\mathcal{R}^{\mathsf{Count}}$, we observe that the costs of flatten depend on those of append, plus the costs of two recursive invocations of flatten on subtrees. The resulting recurrence may expressed for the four additive metrics by:

$$\begin{aligned} \mathit{FlTime}\ n &= \mathit{FlCost}\ \mathit{AppTimeF}\ \mathit{AppTimeC}\ 38\ 22\ n \\ \mathit{FlCall}\ n &= \mathit{FlCost}\ \mathit{AppCallF}\ \mathit{AppCallC}\ 2\ 2\ n \\ \mathit{FlInv}\ n &= \mathit{FlCost}\ \mathit{AppInvF}\ \mathit{AppInvC}\ 1\ 1\ n \\ \mathit{FlHeap}\ n &= \mathit{FlCost}\ \mathit{AppHeapF}\ \mathit{AppHeapC}\ 2\ 0\ n \end{aligned}$$

where the function $\mathit{FlCost} : \mathcal{N} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}$ is defined by

$$\begin{aligned} \mathit{FlCost}\ \mathit{appF}\ \mathit{appC}\ \mathit{base}\ \mathit{step}\ 0 &= \mathit{base} \\ \mathit{FlCost}\ \mathit{appF}\ \mathit{appC}\ \mathit{base}\ \mathit{step}\ (\mathit{Suc}\ n) &= \mathit{step} + \mathit{appF} * (2^n) + \mathit{appC} \\ &\quad + 2 * (\mathit{FlCost}\ \mathit{appF}\ \mathit{appC}\ \mathit{base}\ \mathit{step}\ n), \end{aligned}$$

and the recurrence equation for the frame stack height is given by

$$\begin{aligned} \mathit{FlStack}\ 0 &= 1 \\ \mathit{FlStack}\ (\mathit{Suc}\ n) &= 1\ +\ \mathit{max}(\mathit{FlStack}\ n, 2^n + 1). \end{aligned}$$

The verification of the extended specification

$$\begin{array}{l} ST(\mathsf{TREE}, \mathsf{flatten}, \overline{a}) = \lambda E\, h\, h'\, v\, p. \\ \quad \forall\, n\, x.\ (\exists\, X.\ \overline{a} = [\mathtt{var}\ x]\ \wedge\ h \models_{\mathit{tree}(n,X)} E\langle x \rangle) \longrightarrow \\ \qquad \left(\begin{array}{l} \exists\, Z.\ h' \models_{\mathit{list}(2^n,Z)} v\ \wedge\ Z \cap \mathit{dom}(h) = \emptyset\ \wedge\ h =_{\mathit{dom}(h)} h'\ \wedge \\ \qquad p = \langle (\mathit{FlTime}\ n)\ (\mathit{FlCall}\ n)\ (\mathit{FlInv}\ n)\ (\mathit{FlStack}\ n) \rangle\ \wedge \\ \qquad |\mathit{dom}(h')| = |\mathit{dom}(h)| + \mathit{FlHeap}\ n \end{array} \right) \end{array}$$

again proceeds following the same structure as for the simpler specification (6). As expected, unfolding the recurrence equations leads to functions of exponential growth, since the index n in the predicate $h \models_{\mathit{tree}(n,X)} v$ denotes the height of the tree.

3.5.3 Even/Odd

As a third example, we verify that a run of the mutually recursive methods even and odd (code in Figure 4) exhibits the expected alternation of method invocations. This demonstrates that our rule MUTREC indeed works for mutually recursive methods, and also shows how we may use resource algebras for reasoning about execution traces. In particular, we consider the resource algebra $\mathcal{R}^{\mathsf{InvTr}}$ that collects the static method invocations in execution order. We first define the auxiliary functions $f_{Even}, f_{Odd} : \mathcal{N} \to \overline{\mathit{expr}}$ that calculate the list of invocations occurring inside a method call, relative to a given input.

$$\begin{array}{lcl} \mathtt{method}\ \mathsf{BOOL}\ \mathsf{EO.even}(\mathsf{x}) & = & \mathtt{let}\ \mathsf{b} = \mathit{iszero}\ \mathsf{x}\ \mathtt{in}\ \mathtt{let}\ \mathsf{x} = \mathsf{x} - 1\ \mathtt{in} \\ & & \mathtt{if}\ \mathsf{b}\ \mathtt{then}\ \mathtt{imm}\ \mathtt{true}\ \mathtt{else}\ \mathsf{EO.odd}([\mathtt{var}\ \mathsf{x}]) \\ \mathtt{method}\ \mathsf{BOOL}\ \mathsf{EO.odd}(\mathsf{y}) & = & \mathtt{let}\ \mathsf{b} = \mathit{iszero}\ \mathsf{y}\ \mathtt{in}\ \mathtt{let}\ \mathsf{y} = \mathsf{y} - 1\ \mathtt{in} \\ & & \mathtt{if}\ \mathsf{b}\ \mathtt{then}\ \mathtt{imm}\ \mathtt{false}\ \mathtt{else}\ \mathsf{EO.even}([\mathtt{var}\ \mathsf{y}]) \end{array}$$

Fig. 4. Code of methods even and odd

$$
\begin{aligned}
f_{Even}\ 0 &= [\,] \\
f_{Even}\ (Suc\ n) &= if\ (\exists\, k.\ Suc\ n = 2*k)\ then\ (f_{Even}\ n)\ @\ [\mathsf{EO.even}([\mathtt{var\ y}])] \\
&\quad\ else\ (f_{Even}\ n)\ @\ [\mathsf{EO.odd}([\mathtt{var\ x}])] \\
f_{Odd}\ 0 &= [\,] \\
f_{Odd}\ (Suc\ n) &= if\ (\exists\, k.\ Suc\ n = 2*k)\ then\ (f_{Odd}\ n)\ @\ [\mathsf{EO.odd}([\mathtt{var\ x}])] \\
&\quad\ else\ (f_{Odd}\ n)\ @\ [\mathsf{EO.even}([\mathtt{var\ y}])]
\end{aligned}
$$

Next, the specification table entries for even and odd are defined. The specifications

$$
\begin{aligned}
&ST(\mathsf{EO},\mathsf{even},\overline{a})\ =\ \lambda E\,h\,h'\,v\,p. \\
&\quad \forall\, w.\ (\overline{a} = [\mathtt{var}\ w]\ \wedge\ E\langle w\rangle \geq 0) \longrightarrow \begin{pmatrix} v = is\_even(E\langle w\rangle)\ \wedge \\ p = \mathsf{EO.even}([\mathtt{var}\ w]) :: (f_{Even}(E\langle w\rangle)) \end{pmatrix} \\
&ST(\mathsf{EO},\mathsf{odd},\overline{a})\ =\ \lambda E\,h\,h'\,v\,p. \\
&\quad \forall\, w.\ (\overline{a} = [\mathtt{var}\ w]\ \wedge\ E\langle w\rangle \geq 0) \longrightarrow \begin{pmatrix} v = is\_odd(E\langle w\rangle)\ \wedge \\ p = \mathsf{EO.odd}([\mathtt{var}\ w]) :: (f_{Odd}(E\langle w\rangle)) \end{pmatrix}
\end{aligned}
$$

constrain the form of input arguments, prefix the outermost call to the list of internal invocations, and specify the result using the semantic functions

$$
\begin{aligned}
is\_even\ v &= if\ (\exists\, n.\ v = 2*n)\ then\ \mathtt{true}\ else\ \mathtt{false} \\
is\_odd\ v &= if\ (\exists\, n.\ v = 2*n)\ then\ \mathtt{false}\ else\ \mathtt{true}
\end{aligned}
$$

The context Γ_{EO} contains one entry for each method:

$$
\Gamma_{\mathsf{EO}} = \left\{ \begin{array}{l} (\mathsf{EO.even}([\mathtt{var\ y}]), ST(\mathsf{EO},\mathsf{even},[\mathtt{var\ y}])), \\ (\mathsf{EO.odd}([\mathtt{var\ x}]), ST(\mathsf{EO},\mathsf{odd},[\mathtt{var\ x}])) \end{array} \right\}
$$

The verification of $\Gamma_{\mathsf{EO}} \models ST$ unfolds each method body once, resolving the invocation of the opposite method by rule VAX. We thus obtain

$$
\emptyset \rhd \mathsf{EO.even}([\mathtt{var\ z}]) : ST(\mathsf{EO},\mathsf{even},[\mathtt{var\ z}])
$$

and

$$
\emptyset \rhd \mathsf{EO.odd}([\mathtt{var\ z}]) : ST(\mathsf{EO},\mathsf{odd},[\mathtt{var\ z}])
$$

for an arbitrary variable z using rule ADAPTS, and unfolding the definitions of f_{Even} and f_{Odd} yields results such as

$$\begin{aligned}\emptyset \rhd \mathsf{EO.even}([\mathtt{var\ z}]) : \lambda E\, h\, h'\, v\, p.\ E\langle \mathsf{z}\rangle = 4 \longrightarrow \\ v = \mathtt{true} \wedge \\ p = [\,&\mathsf{EO.even}([\mathtt{var\ z}]), \mathsf{EO.odd}([\mathtt{var\ x}]), \\ &\mathsf{EO.even}([\mathtt{var\ y}]), \mathsf{EO.odd}([\mathtt{var\ x}]), \\ &\mathsf{EO.even}([\mathtt{var\ y}])].\end{aligned}$$

3.5.4 Limits on parameter values

As a final example, we show the use of the resource algebra $\mathcal{R}^{\mathsf{PVal}}$ to impose limits on the arguments of invocations of a (native) method. The code fragment shown in Figure 5 shows a loop that repeatedly prepends a new element to a list l and then invokes C.M with l. In each iteration, the first argument, n, is intended to describe an upper bound on the length of l.

$$\left[\begin{array}{l} f \mapsto \mathtt{let}\ n = n+1\ \mathtt{in}\ \mathtt{let}\ l = \mathtt{new}\ \mathsf{LIST}\ [\mathsf{HD} := n, \mathsf{TL} := l]\ \mathtt{in} \\ \qquad \mathsf{C.M}(\mathtt{var}\ n, \mathtt{var}\ l)\ ;\ \mathtt{let}\ b = \mathit{if}\ n < 42\ \mathit{then}\ \mathtt{imm\ true}\ \mathit{else}\ \mathtt{imm\ false}\ \mathtt{in} \\ \qquad \mathtt{if}\ b\ \mathtt{then\ call}\ f\ \mathtt{else\ var}\ l \end{array}\right]$$

Fig. 5. A loop repeatedly invoking method C.M.

A policy that describes the desired relation is

$$P(\overline{a}) \equiv \exists\, x\, y\, n\, R.\ \overline{a} = [\mathtt{var}\ x, \mathtt{var}\ y] \ \wedge\ h \models_{\mathit{list}(n,R)} E\langle y\rangle \ \wedge\ n \leq E\langle x\rangle \ \wedge\ E\langle x\rangle \leq 42.$$

This expresses that the method must always be invoked on two arguments, the first argument must be a bound on the length of the list contained in the second argument; the list also has a fixed concrete limit on its length.

Provided that the body of C.M itself satisfies the policy, we can verify that all invocations of C.M arising from a call to f respect the policy, subject to conditions on the initial state. The verification of

$$\begin{aligned}\emptyset \rhd \mathtt{call}\ f : \lambda E\, h\, h'\, v\, p.\ (\exists\, m\, R.\ h \models_{\mathit{list}(m,R)} E\langle l\rangle \ \wedge\ m \leq E\langle n\rangle) \\ \longrightarrow p = (E\langle n\rangle < 42)\end{aligned}$$

proceeds very similarly to the proofs described earlier.

In order to prove that each derivable judgement is semantically valid, we first define a *relativised* notion of validity.

Definition 8 *(Relativised validity) Specification A is* valid for e at depth n, *written* $\models_n e : A$, *if*

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

Here, the judgement $E \vdash h, e \Downarrow_m h', v, p$ refers to an operational semantics that extends the semantics given in Section 2.4 by an explicit index which models the derivation height. Note that this counter is only used for meta-reasoning and is independent from the resources. It is immediate to show the equivalence of the two semantics and we omit the details.

The counter n in Definition 8 restricts the set of pre- and post-states for which A has to be fulfilled. It is easy to show that $\models e : A$ is equivalent to $\forall n.\ \models_n e : A$, and that relativised validity is downward closed, i.e. that for $m \leq n$, $\models_n e : A$ implies $\models_m e : A$.

Like unrestricted validity, relativised validity may be generalised to contexts:

Definition 9 *(Relativised context validity) Context* Γ *is* valid at depth n, *written* $\models_n \Gamma$, *if for all* $(e,A) \in \Gamma$, $\models_n e : A$ *holds. Assertion A is* valid for e in context Γ at depth n, *denoted* $\Gamma \models_n e : A$, *if* $\models_n \Gamma$ *implies* $\models_n e : A$.

As a benefit of the index and of relativised validity, proofs may be carried out by induction on the derivation height of the operational semantics (25; 21; 26).

In particular, we can prove the following lemma:

Lemma 10 *For* $\models_n \Gamma$ *and* $\Gamma \cup \{\texttt{call}\, f, A\} \rhd body_f : \Theta(A,f)$, *let*

$$\models_m (\Gamma \cup \{\texttt{call}\, f, A\}) \longrightarrow \models_m body_f : \Theta(A,f)$$

hold for all m. Then $\models_n \texttt{call}\, f : A$.

PROOF. By induction on n.

Similar results hold for static and virtual method invocations. From this, the following result may be proven:

Lemma 11 *If* $\Gamma \rhd e : A$ *then* $\forall n.\ \Gamma \models_n e : A$

PROOF. By induction on the derivation of $\Gamma \rhd e : A$.

Finally, the soundness statement is obtained from Lemma 11 by unfolding the definitions of (relativised) validity.

Theorem 12 *(Soundness) If* $\Gamma \rhd e : A$ *then* $\Gamma \models e : A$.

In particular, an assertion that may be derived using the empty context is valid: $\emptyset \rhd e : A$ implies $\models e : A$.

3.7 Completeness

The soundness of a program logic ensures that derivable judgements assert valid statements with respect to the operational semantics. Soundness is thus paramount to a trustworthy proof-carrying code system. In contrast, completeness of program logics has hitherto been mostly of meta-theoretic interest. For the intended use as the basis of MRG's hierarchy of program logics, however, this meta-theoretic motivation is complemented by a pragmatic motivation. The intention of encoding (possibly yet unknown) high-level type systems as systems of derived assertions requires that any property that (for a given notion of validity) holds for the operational semantics be indeed provable. Partially, this requirement concerns the expressiveness of the assertion language, which, thanks to our choice of shallow embedding, is guaranteed, as any HOL-definable predicate may occur in assertions. On the other hand, the usage of a logically incomplete ambient logic such as HOL renders the program logic immediately incomplete itself, via the rule of consequence. The by now accepted idea of *relative* completeness (27) proposes to separate reasoning about the program logic from issues regarding the logical language. In particular, the side condition of rule VCONSEQ only needs to *hold* in the meta-logic, instead of being required to be provable. Since Kleymann's work (21), it is customary to follow this suggestion for shallow embeddings of program logics in theorem provers.

In our setting, the role of *most general formulae*, originally introduced by Gorelik (28) to prove completeness of recursive programs, is played by *strongest specifications*. These are those assertions that are satisfied exactly for the tuples of the operational semantics.

Definition 13 *(Strongest specification) The strongest specification for e is defined by*

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

It is immediate that strongest specifications are valid

$$\models e : SSpec(e)$$

and imply all other valid specifications:

$$\text{If } \models e : A \text{ and } SSpec(e)\ E\ h\ h'\ v\ p \text{ then } A\ E\ h\ h'\ v\ p. \tag{8}$$

The context Γ_{strong} associates to each function label and each method declaration in the global program P its strongest specification

$$\Gamma_{strong} \equiv \left\{ (e, SSpec(e)) \mid \begin{array}{r} \exists f.\ e = \texttt{call}\, f \ \vee\ \exists\, c\ m\ \overline{a}.\ e = c.m(\overline{a}) \\ \vee\ \exists\, x\ m\ \overline{a}.\ e = x \cdot m(\overline{a}) \end{array} \right\}$$

By induction on e, we prove

Lemma 14 *For any e, we have* $\Gamma_{strong} \rhd e : SSpec(e)$.

This result can be used to show that Γ_{strong} respects the *strongest specification table*,

$$ST_{strong} \equiv (\lambda f.\ SSpec(\texttt{call}\, f), \lambda\, c\ m\ \overline{a}.\ SSpec(c.m(\overline{a})), \lambda\, x\ m\ \overline{a}.\ SSpec(x \cdot m(\overline{a}))).$$

Lemma 15 *We have* $\Gamma_{strong} \models ST_{strong}$.

The proof of this lemma proceeds by unfolding the definitions, using Lemma 14 in the claims for function calls and method invocations.

Next, we prove that

$$\Gamma_{strong} \models ST \text{ implies } \emptyset \rhd e : SSpec(e) \tag{9}$$

for arbitrary specification table ST, by applying rule VCTXT, where the first premise is discharged by Lemma 14 (i.e. Γ is instantiated to Γ_{strong}) and the second premise is discharged by rule MUTREC.

Combining property (9) and Lemma 15 yields $\emptyset \rhd e : SSpec(e)$, from which

Theorem 16 *(Completeness) For any e and A,* $\models e : A$ *implies* $\emptyset \rhd e : A$.

follows by rule VCONSEQ and property (8).

4 Program logic for termination

So far the program logic developed for Grail has been a logic for partial correctness. In this section we will develop a program logic for termination, with the judgement

$\rhd_T \{P\}\ e\downarrow$, to be read as "expression e terminates under pre-condition P." In formalising the concept of a pre-condition, we adopt the same approach as in the core logic and use a shallow embedding. Thus, pre-conditions are predicates over environments and heaps in the meta-logic and have the type $\mathcal{P} \equiv \mathcal{E} \to \mathcal{H} \to \mathcal{B}$.

With this termination logic a total correctness statement, that expression e fulfils assertion A and terminates under pre-condition P, is then the conjunction of statements of these two logics: $\emptyset \rhd e : A \ \wedge\ \rhd_T \{P\} e\downarrow$. In contrast to the usual approach of adding the termination-guaranteeing side-conditions directly into the rules for function calls and method invocations [6], our approach has the advantage of being modular: we do not have to modify the underlying logic for partial correctness at all. To add confidence to the results in this section, we have formalised the termination logic and its soundness proof in Isabelle/HOL.

As side conditions in the rules for let and for composition we will need a form of semantic "implication", relating an overall pre-condition P to the pre-condition P' for the second sub-expression, possibly after a value binding to the variable x. We express this as an assertion in the partial correctness logic, and define the following combinator.

Definition 17 *A* pre-condition implication *of two pre-conditions P, P' with a binding to variable x, written $P \longrightarrow_{\langle x:=\rangle} P'$, is an assertion defined as follows*

$$\lambda E\ h\ h'\ v\ p.\ P\ E\ h\ \longrightarrow\ (\exists i.\ v = i\ \wedge\ P'\ (E\langle x := i\rangle)\ h')$$

Recall that the operation $E\langle x := i\rangle$ performs an update of the variable x by the value i. A similar $\longrightarrow_{\langle \_:=\rangle}$ combinator expresses a pre-condition implication without value-binding.

4.1 Proof system

We now introduce a proof system for termination. It uses a relation symbol $\rhd_T \{P\}\ e\downarrow$, where P is a pre-condition and e an expression. This symbol can be instantiated in various ways, and the following sections will elaborate on the relationship with the semantic definition of termination.

While the core logic uses contexts to collect assumptions on methods and functions we rely here on meta-level implication for this purpose: To prove that a call $f()$ terminates for all inputs one must exhibit a measure function and prove for all n that the body of f terminates for inputs of measure up to n assuming that calls to f

[6] One notable exception is (29), where a termination logic is built on top of a Hoare-Logic for a while-language.

terminates for inputs of measure less than n. Unlike in the partial case such a rule can be justified by meta-level induction on the termination measure.

Note that a similar use of metalvel implication would be unsound for partial correctness. For each function (or method) $f()$ it trivially holds that either $f()$ satisfies the partial correctness assertion ϕ or else it is the case that its body satisfies it assuming that any call to $f()$ satisfies ϕ by "ex falso quodlibet". Thus, in the partial correctness case the access to assumptions on calls must be suitably restricted (e.g., using contexts) so as to rule out meta-level case distinctions as above, whereas no such restriction is necessary for termination logic due to the outside quantification over the measure.

Thus, we can use a shallow encoding of object-logic contexts, or, in other terms, a form of hypothetical-parametrical judgements as in type theory, where the meta-logic context encodes the object one. The drawback is that termination does not have an induction principle, yet the encoding abstracts from explicit context management operations such as axiomatic lookup or weakening, which are delegated to the meta-logic.

Another noteworthy and to our knowledge original contribution is the proof of completeness of the logic even for mutually recursive functions without introducing ad hoc rules that deal with mutual recursion explicitly.

$$\frac{\forall E\ h.P'\ E\ h \longrightarrow P\ E\ h \quad \rhd_T \{P\}\ e\ \downarrow}{\rhd_T \{P'\}\ e\ \downarrow}\ (\textsc{TConseq})$$

$$\frac{}{\rhd_T \{P\}\ \texttt{null}\ \downarrow}\ (\textsc{TNull}) \qquad \frac{}{\rhd_T \{P\}\ \texttt{imm}\ i\ \downarrow}\ (\textsc{TImm})$$

$$\frac{}{\rhd_T \{P\}\ \texttt{var}\ x\ \downarrow}\ (\textsc{TVar}) \qquad \frac{}{\rhd_T \{P\}\ \texttt{prim}\ op\ x\ y\ \downarrow}\ (\textsc{TPrim})$$

$$\frac{\forall E\ h.\ P\ E\ h \longrightarrow E\langle x\rangle \neq \mathsf{null}}{\rhd_T \{P\}\ x.t\ \downarrow}\ (\textsc{TGetF}) \qquad \frac{\forall E\ h.\ P\ E\ h \longrightarrow E\langle x\rangle \neq \mathsf{null}}{\rhd_T \{P\}\ x.t{:=}y\ \downarrow}\ (\textsc{TPutF})$$

$$\frac{}{\rhd_T \{P\}\ c \diamond t\ \downarrow}\ (\textsc{TGetS}) \qquad \frac{}{\rhd_T \{P\}\ c \diamond t{:=}y\ \downarrow}\ (\textsc{TPutS})$$

$$\frac{}{\rhd_T \{P\}\ \texttt{new}\ c\ \overline{[t_i := x_i]}\ \downarrow}\ (\textsc{TNew})$$

$$\frac{\rhd_T\{\lambda E\ h.\ P\ E\ h\ \wedge\ E\langle x\rangle = \mathsf{true}\}\ e_1\ \downarrow \quad \rhd_T\{\lambda E\ h.\ P\ E\ h\ \wedge\ E\langle x\rangle = \mathsf{false}\}\ e_2\ \downarrow \quad \forall E\ h.\ P\ E\ h\ \longrightarrow\ E\langle x\rangle \in \{\mathsf{true},\mathsf{false}\}}{\rhd_T\{P\}\ \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \downarrow}\ (\textsc{TIf})$$

$$\frac{\rhd_T\{P\}\ e_1\ \downarrow \quad \rhd_T\{P'\}\ e_2\ \downarrow \quad \rhd e_1 : P \longrightarrow_{\langle \_:=\rangle} P'}{\rhd_T\{P\}\ e_1\ ;\ e_2\ \downarrow}\ (\textsc{TComp})$$

$$\frac{\rhd_T\{P\}\ e_1\ \downarrow \quad \rhd_T\{P'\}\ e_2\ \downarrow \quad \rhd e_1 : P \longrightarrow_{\langle x:=\rangle} P'}{\rhd_T\{P\}\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ \downarrow}\ (\textsc{TLet})$$

$$\frac{\forall n.\ \rhd_T\{\lambda E\ h.\ \exists m.\ m < n\ \wedge\ P\ m\ E\ h\}\ \mathtt{call}\ f\ \downarrow\ \longrightarrow\ \rhd_T\{P\ n\}\ body_f\ \downarrow}{\rhd_T\{\lambda E\ h.\ \exists n.\ P\ n\ E\ h\}\ \mathtt{call}\ f\ \downarrow}\ (\textsc{TCall})$$

$$\frac{\begin{array}{c}\forall n.\ \rhd_T\{\lambda E\ h.\ \exists m.\ m < n\ \wedge\ P\ m\ E\ h\}\ c.m(\overline{a})\ \downarrow\ \longrightarrow \\ \rhd_T\{\lambda E\ h.\ \exists E'.\ E = Env(\mathtt{self} :: pars_{c,m}, \mathsf{null} :: \overline{a}, E')\ \wedge\ P\ n\ E'\ h\}\ body_{c,m}\ \downarrow\end{array}}{\rhd_T\{\lambda E\ h.\ \exists n.\ P\ n\ E\ h\}\ c.m(\overline{a})\ \downarrow}\ (\textsc{TSInv})$$

$$\frac{\begin{array}{c}\forall n.\ \rhd_T\{\lambda E\ h.\ \exists m\ l.\ m < n\ \wedge\ E\langle x\rangle = \mathsf{Ref}\ l\ \wedge\ h(l) = c\ \wedge\ P\ m\ E\ h\}\ x\cdot m(\overline{a})\ \downarrow\ \longrightarrow \\ \rhd_T\{\lambda E\ h.\ \exists E'.\ E = Env(\mathtt{self} :: pars_{c,m}, x :: \overline{a}, E')\ \wedge\ P\ n\ E'\ h\}\ body_{c,m}\ \downarrow\end{array}}{\rhd_T\{\lambda E\ h.\ \exists n\ l.\ E\langle x\rangle = \mathsf{Ref}\ l\ \wedge\ h(l) = c\ \wedge\ P\ n\ E\ h\}\ x\cdot m(\overline{a})\ \downarrow}\ (\textsc{TInv})$$

4.2 Discussion

The TConseq rule allows us to weaken the pre-condition P' to P if it is a logical consequence from P'. For the leaf cases the only possibility of non-termination is a null-reference in a get- or put-field operation. The operational semantics gets stuck in this case. The if rule, TIf, requires termination of both branches under the added knowledge on the value in the header, which must be a boolean value. In the rules for composition, TComp, and for let, TLet, a side condition, phrased in terms of the partial correctness logic, is used to establish a link between the overall pre-condition P and the pre-condition P' before the second sub-expression. In order to deal with recursion, the rules TCall, TSInv and TInv use a well-founded relation $<$ and require that termination of the body must be provable, assuming termination of a Call (or Invoke) for an arbitrary, smaller measure, i.e. for $m < n$. Note that any well-founded ordering can be used in these rules. It is also worth noting

that this formalisation, with an inner existential quantifier in the pre-condition, is easier to apply than the version that uses an outer universal quantification over m, since the value binding is deferred until the pre-condition is exposed in the proof. As usual for such measures, the variable n is existentially quantified in the conclusions of the TCALL, TSINV and TINV rules, since it captures the value of the measure at the call point. The rules for static and dynamic method invocation account for the modification of the environment, by applying the *Env* operator to the outer environment E'. The TINV rule contains another clause in its pre-condition, stating that the variable x points to an instance of class c.

4.3 Soundness

We now define the notion of termination semantically as follows.

Definition 18 *(Termination) The expression e* terminates under the pre-condition *P, written* $\models_T \{P\}\ \mathtt{e}\downarrow$*, iff for all environments E and heaps h*

$$P\ E\ h\ \longrightarrow\ \exists h'\ v\ p.\ E \vdash h, e \Downarrow h', v, p$$

To spell it out, a Grail expression e terminates under a pre-condition P, if for all states, i.e. environments E and heaps h, that fulfil P, a final state, comprised of heap h', value v and resources p, exists in the operational semantics.

We now prove that the rules for the proof system in Section 4.1 are sound w.r.t. this definition of termination, i.e. we will prove them as lemmas on the semantic definition of termination.

Theorem 19 *(TSoundness) The relation* $\models_T \{P\}\ e\downarrow$ *is closed under the proof rules for* $\rhd_T\{P\}\ e\downarrow$ *in Section 4.1.*

PROOF. By cases on e: the leaf cases are straightforward. In the TCOMP and TLET cases, soundness of the partial correctness logic is used to propagate the pre-condition P through the let header. Likewise for the header in the conditional. The TCALL, TSINV and TINV cases are proven by well-founded induction of the predicate $\lambda n.\forall E\ h.P\ n\ E\ h\ \longrightarrow (\exists h'\ v\ p.\ E\ \vdash\ h,\ e'\ \Downarrow h',\ v\ , p)$ over n, where e' is $\mathtt{call}\, f$, $c.m(\overline{a})$, or $x \cdot m(\overline{a})$, respectively.

4.4 Completeness

In this section we present a proof of completeness for the termination logic. In the proof we only cover functions, but the extension to methods can use the same

techniques for a suitably modified definition of IH_k below. Note that we instantiate the well founded relation $<$ to "strictly less" on the naturals. The overall structure of the completeness proof is as follows: as most general formulae we use pairs of expression e and its weakest pre-condition $wp\ e$. Having a standard consequence rule it suffices to show $\rhd_T \{wp\ e\}\ e\downarrow$. We reduce this to a proof of Lemma 27, which augments the weakest precondition with a descending chain of naturals, and the lemma can be proven by induction over this chain.

Definition 20 *The* weakest pre-condition *of an expression $e \in expr$, written $wp\ e$, is*

$$\lambda\, E\ h.\ \exists\, h'\ v\ p.\ E \vdash h, e \Downarrow h', v, p$$

Definition 21 *The* bounded weakest pre-condition *of an expression $e \in expr$ for $n \in \mathbb{N}$, written $wp'\ e\ n$, is*

$$\lambda\, E\ h.\ \exists\, m \leq n.\ \exists\, h'\ v\ p.\ E \vdash h, e \Downarrow_m h', v, p$$

The index m in the operational semantic judgement is the same index on the derivation height that was used in the formulation of relativised validity in Definition 9. It is easy to show that $wp\ e$ is a pre-condition and that it is the weakest such pre-condition.

Lemma 22 *For all expressions $e \in expr$ and pre-conditions $P \in \mathcal{P}$,*

$$\models_T \{wp\ e\}\ e\downarrow \tag{10}$$
$$\models_T \{P\}\ e\downarrow \longrightarrow \forall E\ h.P\ E\ h \longrightarrow wp\ e\ E\ h \tag{11}$$

We make use of the following lemmas, which are shown by unfolding the definition of wp', using rules of the semantics and simple logical reasoning. The last sub-lemma (16) is shown by induction over e.

Lemma 23 *For all expressions $e \in expr$, function names $f \in \mathcal{F}$, environments $E \in \mathcal{E}$, heaps $h \in \mathcal{H}$, $n, n' \in \mathbb{N}$,*

$$wp\ e\ E\ h\ = \exists n.\ wp'\ e\ n\ E\ h \tag{12}$$
$$wp'\ e\ n\ E\ h\ \longrightarrow \exists n.\ wp'\ e\ n\ E\ h \tag{13}$$
$$wp'\ e\ n\ E\ h\ \wedge\ n \leq n'\ \longrightarrow\ wp'\ e\ n'\ E\ h \tag{14}$$
$$wp'\ (body_f)\ n\ E\ h = wp'\ (\texttt{call}\, f)\ (n+1)\ E\ h \tag{15}$$
$$\rhd_T \{\lambda E\ h.\ \mathsf{false}\}\ e\ \downarrow \tag{16}$$

In the proof for completeness, we use a variant of the call rule, which follows directly from TCALL. This version of the rule goes back to Sokołowski (30).

$$\frac{\begin{array}{c}\forall E\ h.\neg P\ 0\ E\ h \\ \forall n.\rhd_T\ \{P\ n\}\ \texttt{call}\, f\ \downarrow \longrightarrow \rhd_T\ \{P\ (n+1)\}\ body_f\ \downarrow\end{array}}{\rhd_T\{\lambda E\ h.\ \exists n.\ P\ n\ E\ h\}\ \texttt{call}\, f\ \downarrow} \quad \text{(TCALL')}$$

The completeness theorem for the termination logic is now stated as follows. It establishes the fact that, if $\models_T \{P\}e\downarrow$ holds, then this can be proven using the proof rules for $\rhd_T$ alone.

Theorem 24 *(TCompleteness) If $\rhd_T$ is any relation that validates the proof rules in Section 4.1, then $\models_T \subseteq \rhd_T$.*

PROOF. Follows from Lemma 25 and TCONSEQ with Lemma 22(11).

Lemma 25 *For all expressions $e \in expr$, the following holds: $\rhd_T\{wp\ e\}\ e\ \downarrow$*

PROOF. Proof by induction over e. The leaf cases are trivial. The cases TCOMP, TLET and TIF use completeness of the core logic and deterministic evaluation of the operational semantics. The TCALL case uses TCONSEQ with Lemma 27 and Lemma 23(12).

Without loss of generality we assume that the function names are sorted as $f_1,\ldots,f_M$.

Definition 26 *For expression $e \in expr$, $k,M \in \mathbb{N}$, $k \leq M$, $n_1,\ldots,n_k \in \mathbb{N}$, let*

$$IH_k \equiv \forall\ 1 \leq i \leq k.\ \rhd_T \{wp'\ (\texttt{call}\ f_i)\ n_i \wedge \forall 1 \leq j < i.\ n_j \geq n_{j+1}\}\ \texttt{call}\ f_i\ \downarrow \longrightarrow \rhd_T\{wp'\ e\ n_k \wedge \forall 1 \leq j < k.\ n_j \geq n_{j+1}\}\ e\ \downarrow$$

Lemma 27 *For all $k \leq M$, IH_k holds.*

PROOF. By induction over $M-k$. The base case, with $k=M$, follows from Lemma 28, the induction case from Lemma 29 and the induction hypothesis.

The following lemma proves the base case, i.e. IH_M.

Lemma 28 *For all expressions $e \in expr$, $n_1,\ldots,n_M \in \mathbb{N}$,*

$$\forall\ 1 \leq i \leq M.\ \rhd_T \{wp'\ (\texttt{call}\ f_i)\ n_i \wedge \forall 1 \leq j < i.\ n_j \geq n_{j+1}\}\ \texttt{call}\ f_i\ \downarrow \longrightarrow \rhd_T\{wp'\ e\ n_M \wedge \forall 1 \leq j < M.\ n_j \geq n_{j+1}\}\ e\ \downarrow$$

PROOF. By structural induction over e. For all non-call cases, apply the syntax-oriented rule, then for each sub-expression TCONSEQ with Lemma 23(14), and then the induction hypothesis. The call case follows directly via TCONSEQ with Lemma 23(14) and monotonicity of the $\geq$ chain.

The following lemma allows us to add information on the function f_{k+1} into the meta-context. This is used to discharge the implication in IH_{k+1} in the proof of Lemma 27. Note, that this step corresponds to the cut-rule on an explicit context, which was used in the proof for completeness of the core logic.

Lemma 29 *For all $n_1, \ldots, n_{k+1} \in \mathbb{N}$, if IH_{k+1} then*

$$\forall\, 1 \leq i \leq k.\ \rhd_T \{wp'\ (\texttt{call}\ f_i)\ n_i \ \wedge\ \forall 1 \leq j < i.\ n_j \geq n_{j+1}\}\ \texttt{call}\ f_i\ \downarrow \longrightarrow$$
$$\rhd_T \{wp'\ (\texttt{call}\ f_{k+1})\ n_{k+1} \ \wedge\ \forall 1 \leq j < k+1.\ n_j \geq n_{j+1}\}\ \texttt{call}\ f_{k+1}\ \downarrow$$

PROOF. TCONSEQ via Lemma 23(13) then TCALL'. This leaves to prove that $\rhd_T \{wp'\ (\texttt{call}\ f_{k+1})\ (n_{k+1}+1)\ \wedge\ \forall 1 \leq j < k+1.\ n_j \geq n_{j+1}+1\}\ body_{f_{k+1}}\downarrow$, assuming $\rhd_T \{wp'\ (\texttt{call}\ f_{k+1})\ n_{k+1}\ \wedge\ \forall 1 \leq j < k+1.\ n_j \geq n_{j+1}\}\ \texttt{call}\ f_{k+1}\downarrow$. This follows via TCONSEQ with Lemma 23(15) and monotonicity of the $\geq$ chain from $\rhd_T \{wp'\ (body_{f_{k+1}})\ n_{k+1}\ \wedge\ \forall 1 \leq j < k+1.\ n_j \geq n_{j+1}\}\ body_{f_{k+1}}\downarrow$, which we know from the induction hypothesis IH_{k+1}.

4.5 Example

As example program we use a variant of the even/odd program from Section 3.5.3, which uses only functions and decrements x. We emphasise that we can prove termination of even/odd without any additional rules for mutual recursion and we do the proof directly on the rules in Section 4.1. We focus on the specifics of a proof of mutually recursive functions.

We need to specify the pre-condition under which the functions terminate. Typically this involves characterising the possible parameter values and class membership for the methods involved. The format of the pre-condition should fit the structure of the pre-condition in the call rule to be existentially quantified over the measure. In this case we can use the same parameterised pre-condition for both functions: $P_{e/o} \equiv \lambda n\ E\ h.\ 0 \leq E\langle x\rangle \wedge n = E\langle x\rangle$.

We need to provide a measure for each function, which assures termination. With the format of the pre-condition above, we can read the measure straight off: $M_{e/o} \equiv \lambda E\ h.\ E\langle x\rangle$. We prove termination of `even`, i.e.

$$\rhd_T \{\lambda E\ h.\ \exists n.\ P_{e/o}\ n\ E\ h\}\ \texttt{call even}\downarrow$$

In the proof we first apply the TCALL rule, adding information on the pre-condition for `even` to the meta-context. Only syntax-directed rules are used to traverse the body of the function. When encountering `odd` we first use TCONSEQ and then prove termination of `odd` under the pre-condition $\lambda E\ h.\exists n'.\ P_{e/o}\ n'\ E\ h\ \wedge\ n' = n-1$ which is implied by the pre-condition at the call point. We use the constraint $n' = n-1$ to relate the measure for `even`, namely n, to that of `odd`, namely n', and then apply the TCALL rule. Following the proof idea in the completeness proof, we could use the weaker constraint $n' \leq n$; however, this would give a longer proof and thus we pick the exact value for this program. When arriving at the second call to `even`, we use the first clause in the meta-context, i.e. :

$$\rhd_T\{\lambda E\ h.\ \exists m.\ m < n\ \wedge\ 0 \leq E\langle x\rangle\ \wedge\ m = E\langle x\rangle\}\ \texttt{call even}\ \downarrow$$
$$\rhd_T\{\lambda E\ h.\ n-1 < n'\ \wedge\ 0 \leq E\langle x\rangle\ \wedge\ n-1 = E\langle x\rangle\}\ \texttt{call odd}\ \downarrow$$

These clauses were added by the two invocations of the TCALL rule. We now instantiate the measure to $n-2$ for `even`. After using TCONSEQ only the side-conditions generated by the syntax-oriented rules remain. These are solved using soundness of $\rhd$, rules of the operational semantics and some auxiliary lemmas, formalising non-interference of variables in the evaluation of sub-expressions. A case distinction over $n > 0$ is necessary to separate the recursion path, from the non-recursive path through the program, and basic arithmetic completes the proof.

5 An infrastructure for resource certification

In this section we describe the infrastructure for certification of resources which we build on top of the program logics presented above. This is based on a multi-layered approach (shown in Figure 6).

To begin with, we use an applied type system. While the complexity of proving general program correctness often restricts the research on program verification to only security-critical systems, increasingly complex type systems have found their way into main-stream programming and are accepted as useful tools in software development. Given this complexity, soundness proofs become subtle, and the user of the code has to trust both the proof and the translation of the high-level code into the object-code. Our approach in guaranteeing the absence of bad behaviour is to translate types into proofs in a suitably specialised program logic. Going by this route, a formal certificate is generated that can be independently checked.

At the basis we have our (trusted) *operational semantics* that is extended with general "effects" that encode the basic security-sensitive operations (for example, heap allocation if the security policy is bounded heap consumption). The Foundational PCC approach (31) performs proofs directly on this level and thereby reduces the

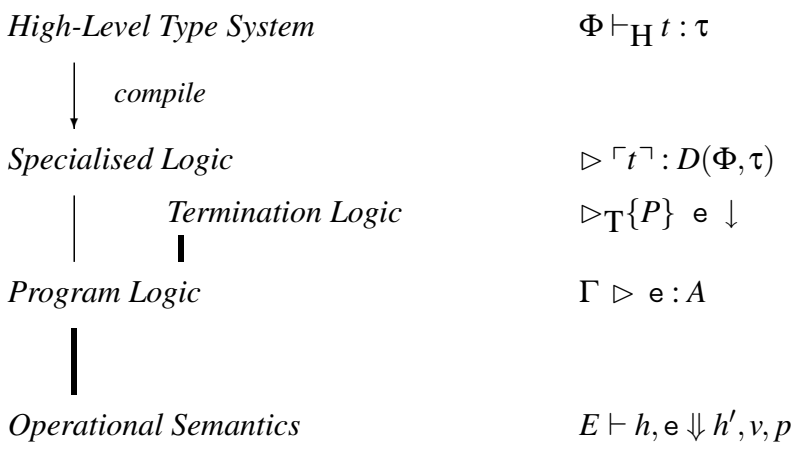

Fig. 6. A family of logics for resource consumption

size of the trusted code base (TCB). Similarly, since we formalise the entire hierarchy of logics in a theorem prover we do not need to include any of these logics into the TCB.

On the next level we have our general-purpose *program logic* for partial correctness. Its role is to serve as a platform at which various higher level logics may be unified. The latter purpose makes logical completeness of the program logic a desirable property, which has hitherto been mostly of meta-theoretic interest. Of course, soundness remains mandatory, as the trustworthiness of any application logic defined at higher levels depends upon it. Our soundness and completeness results establish a tight link between operational semantics and program logic, as shown in Figure 6.

While assertions in the core logic make statements on partial program correctness, as we have seen, the *termination logic* is defined on top of this level to certify termination. This separation improves modularity in developing these logics, and allows us to use judgements of the partial correctness logic when talking about termination.

On top of the general-purpose logic, a *specialised logic* (for example the heap logic of (32)) is defined that captures the specifics of a particular security policy. This logic uses a restricted format of assertions, called *derived assertions*, which reflects the information of the high-level type system. Therefore, the specialised logic can be embedded into the core logic. Judgements in the specialised logic have the form $\triangleright \ulcorner \mathtt{t} \urcorner : D(\Phi, \tau)$, where the expression $\ulcorner \mathtt{t} \urcorner$ is the result of compiling a high-level term t down to a low-level language, and the information in the high-level type system is encoded in a special form of assertion that depends on the context and type associated to t, $D(\Phi, \tau)$. Depending on the property of interest, this level may be further refined into a hierarchy of proof systems, for example if parts of the

soundness argument of the specialised assertions can be achieved by different type systems. In contrast to the general-purpose logic, this specialised logic is not expected to be logically complete, but it should provide support for automated proof search. In the case of the logic for heap consumption, this is achieved by formulating a system of derived assertions whose level of granularity is roughly similar to the high-level type system. However, the rules are expressed in terms of code fragments in the low-level language. Since the side conditions of the typing rules are computationally easy to validate, automated proof search is supported by the syntax-directness of the typing rules. At points where syntax-directness fails — such as recursive program structures — the necessary invariants are provided by the type system.

Thus, on the top level we find a *high-level type system*, that encodes information on resource consumption. In the judgement $\Phi \vdash_H t : \tau$, the term t has an (extended) type τ in a context Φ. This level is of immediate relevance for the programming languages area, and many type-based inferences have been suggested. The case we have worked out is the Hofmann & Jost type system for heap usage (33). In our work, however, we give a framework for tying such analyses into a fully formalised infrastructure for reasoning about resource consumption.

However, for illustration purposes, let us instantiate this hierarchy with a a simple security property of "well-formed datatypes", related to memory safety polices considered in PCC. To model datatypes, we use a version of the previously defined predicate $h \models_\tau a$, expressing that an address a in heap h is the start of a (high-level) data-type τ. In the core program logic we can express the fact that the a method $c.m_f$ compiled from the high-level function $f :: \tau\ list \to \tau\ list$ preserves a well-formed data-type as follows:

$$\rhd c.m_f(x)\ :\ \lambda E\ h\ h'\ v.\ h \models_{list} E\langle x\rangle\ \longrightarrow\ h' \models_{list} v$$

To abstract over the details of modelling data-structures, we construct a specialised logic for this security property, by restricting the form of the assertions to reflect the high-level property to be formalised. These *derived assertions* $D(\Phi,\tau)$ must have the following property for a program term t and a high-level type τ:

$$\Phi \vdash_H t : \tau\ \Longrightarrow\ \rhd\ulcorner t \urcorner : D(\Phi,\tau)$$

The definition of the derived assertion carries with it the high-level types of the variables, and accesses low-level predicates, which are needed to express the security policy but should be hidden in the specialised logic itself:

$$D(x : list, y : list, list) \equiv \lambda E\ h\ h'\ v\ p.\ \ h \models_{list} E\langle x\rangle\ \wedge\ h \models_{list} E\langle y\rangle\ \longrightarrow\ h' \models_{list} E\langle x\rangle\ \wedge\ h' \models_{list} E\langle y\rangle\ \wedge\ h' \models_{list} v$$

Based on this format we could phrase a rule for a high-level list-cons operator, which abstract from the representation predicates:

$$\frac{\triangleright \ulcorner t_1 \urcorner : D(\Phi, \tau) \quad \triangleright \ulcorner t_2 \urcorner : D(\Phi, \tau\ \mathit{list})}{\triangleright \ulcorner \mathit{cons}(t_1, t_2) \urcorner : D(\Phi, \tau\ \mathit{list})}$$

The soundness of the rules in the specialised logic can then be proven once and for all on top of the core logic. Each rule roughly corresponds to each case in the type soundness proof at the high level. The main advantages of the specialised logic are proofs in this logic being largely syntax-directed with simpler side conditions. Since most of the rules in this logic are syntax-directed, it is possible to turn (high-level) type checking into an (effective) tactic in a proof assistant. In comparison, using an interactive prover to prove such assertions becomes very complicated, except for very simple programs. Even in the presence of a verification condition generator, side conditions arise that are currently too difficult for the automated facilities of a proof assistant, hence considerable manual intervention is necessary. As we remarked earlier, performing a soundness proof of an entire type system may be in itself a daunting task. Moreover, this hardly provides any guarantees with the respect to the compiled code level.

6 Related Work

There has been an outstanding amount of research about formalising the safety of Java/JVM/JavaCard, see (34) for a review up to 2001. For our purposes, we review only work related to program logics, which we can divide in roughly three categories: imperative and object-oriented logics, pointer logics and bytecode logics. Note, however, that none of this related work contains any formalized account of resources.

6.1 Imperative and object-oriented logics

Most closely related to our work on the meta-theoretical side are Nipkow's implementation of Hoare logic in (25), the Java-light logic by von Oheimb (35), Kleymann's thesis (21), and Hofmann's (26) work on completeness of program logics. The formalized logic by Nipkow in (25) is for a while-language with parameterless functions, with proofs of soundness and completeness. Although several techniques we use in our proofs are motivated by this work, we have made progress on the treatment of mutual recursion and adaptation. In particular, in covering mutual recursion we do not specify a separate derivation system but only use a derived rule MUTREC that keeps our core logic small and easy to handle when proving sound-

ness and completeness. Several options for formalising either VDM or Hoare-style program logics have been explored by Kleymann in his thesis (21). In particular, this work demonstrates how to formalise an adaptation rule that permits to modify auxiliary variables. This logic for total correctness of a while language with functions is closely related to our termination logic. In particular, the call rule is similar, but he does not cover methods. We used these results, together with our own previous experience on VDM-style logics (36) to design the Grail program logic. The techniques used in our completeness proof are based on those by one of the authors in (26).

With respect to other relevant program logics, the one for Java-light by von Oheimb (35) is encoded in Isabelle and proven sound and complete. It covers more object-oriented features, but it works on a higher level than our logic for a bytecode language and does not address resources. Moreover, because of the focus on meta-theoretical properties, it is hardly suitable for concrete program verification, as it can also be seen from the only example provided (`http://isabelle.in.tum.de/Bali/src/Bali5/AxExample.html`). He also discusses, as an alternative to his own logic, a rule for method invocation in a total correctness logic for Java-light that allows a similar handling of mutual recursion as our system. Both Kleymann and von Oheimb develop logics combining functional correctness and termination in one logic, yielding a far more complicated system than our termination logic.

DeBoer et al. (37; 38) present a sound and complete Hoare-style logic for a sequential object-oriented language with inheritance and subtyping. In contrast to our approach, the proof system employs a specific assertion language for object structures, a WP calculus for assignment and object creation, and Hoare rules for method invocation. The approach is heavily based on syntactical substitutions, yet the logic is not compositional. Recently a tool supporting the verification of annotated programs (flowcharts) yielding verification conditions to be solved in HOL has been produced (39). This also extends to multi-threaded Java (40), where also a non-formalized proof of the soundness and completeness of its assertion-based proof system is provided.

Abadi and Leino combine a program logic for a simple object-oriented language with a type system (41; 42). The language supports sub-classing and recursive object types. In a judgement, specifications as well as types are attached to expressions. In contrast to our logic, it uses a global store model, with the possibility of storing pointers to arbitrary methods in objects. As a result of this design decision this logic is incomplete.

Homeier (29) also develops a termination logic, separate from a partial correctness Hoare-logic for a simple while-language with procedures. His treatment of mutual recursion builds on procedure entrance specifications that relate the state at the beginning of procedure with that at a call point in the procedure. These entrance

specifications can be used to encode a descending chain, as we use it in the completeness proof, but in terms of logics his work uses heavier machinery (a separate logic for entrance specifications). In contrast to our work, he doesn't give formal completeness results for his termination logic.

Several projects aim at developing program logics for subsets of Java, mainly as tools for program verification. Müller and Poetzsch-Heffter present a sound Hoare-style logic for a Java subset (43). Their language covers class and interface types with subtyping and inheritance, as well as dynamic and static binding, and aliasing via object references, which has been implemented in the *Jive* tool (44). As part of the *LOOP* (45) project, Huisman and Jacobs (46) present an extension of a Hoare logic that includes means for reasoning about abrupt termination and side-effects, encoded in PVS/Isabelle. In (47) a set of sound rules for the sequential imperative fragment of Java are given, based on JML, combining all possible termination modes in a single logic. *Krakatoa* (48) is a tool for verifying JML-annotated Java programs that acts as front-end to the *Why* system (49), using Coq to model the semantics and conduct the proofs. *Why* produces proof-obligations for programs in imperative-functional style via an interpretation in a type theory of effects and monads. JACK (50) is based on a weakest preconditions calculus, which is asserted rather than semantically derived from a machine model. It generates proof obligations from annotated Java sources. The proof obligations can be discharged using different systems, from automated tools such as *Simplify* to interactive theorem provers. Much effort is invested in making the system usable by Java programmers by means of an Eclipse plug-in and a proof obligation viewer. Recently (51) this has been extended to a logic for sequential bytecode, where annotations are expressed in a Bytecode Modelling Language, which is the target of a JVM compiler. The logic is based on a weakest precondition calculus, whose soundness is reported to have been proven w.r.t. to the operational semantics of the JVM in (52). In addition to defining a predicate transformer for each bytecode, the approach requires the computation of the WP of the whole program using its control flow graph. In the context of the *KeY* project (53), Beckert (54) presents a sequent calculus for a version of Dynamic Logic dedicated to the JAVACARD language with judgments of the form $\Gamma \vdash \langle \mathbf{e} \rangle \phi^U$, where e is a program, Γ encodes preconditions and U is a state update used to cope with aliasing. Loops are dealt with by unrolling and method calls by symbolic execution of the body. Soundness and completeness are not proven. This has been integrated with the *KeY* interactive prover and applied to security properties in (55) (for example only `ISOException` thrown at top level). An attempt is made to modularize the approach to methods by utilising pre and post-condition rather than execution of the body. (56) extends the approach to deal with JAVACARD transactions by introducing a *trace* modality. A related system is *KIV* (57); in particular, (58) presents a formalized operational semantics and sound dynamic logic for full JAVACARD, based on strongest postconditions. Expressions and blocks are flattened (a sort of ANF), new instructions are added (for example, to mark the end of a block) yielding judgments of the form $\Gamma \vdash \langle H; e \rangle \phi, \Delta$, where H is the initial heap. Program rules may be applicable also on the left hand side

of the turnstyle, making the over 50 rules a rich calculus best suited to interactive verification. Furthermore, the encoding is not definitional, but relies on dozens of axioms over operations such as replacement of variables, flattening of blocks etc.

There are also systems more oriented towards *lightweight* static validation: the main one is ESC/JAVA2 (59), a tool that attempts to find common run-time errors (such as null references) in JML-annotated Java programs by static analysis of the program code and its formal annotations. It trades off soundness (and completeness) in favour of full automation.

6.2 Pointer logics

A related issue is the verification of pointer programs. We do not discuss the theoretical papers on Separation Logic (60), although our specifications do include the encoding of separation principles. This is due to our emphasis on formalising our logic in a theorem prover and the unavailability of tools based on Separation Logic at this moment in time. Historically, some of the first formal verification of pointer programs in (61) (and later (62)) used a model where the store is incorporated in the assertion logic. More recent is the verification of several algorithms, including list manipulating programs and the Schorr-Waite graph-marking algorithm, by Bornat (63) using the Jape system. This approach employs a Hoare logic for a while-language with components that are semantically modelled as pointer-indexed arrays. Separation conditions are expressed as predicates on (object) pointers. (64) uses a Hoare logic in the style of Gordon (65) to reason about pointer programs in a simple while-language, including a declarative proof of the correctness of the Schorr-Waite algorithm. An Isabelle/HOL implementation of separation logic following the previous paper is presented in (66), although the author reports proofs (typically in-place reversal) to be slightly more complicated than in (64). Furthermore, little or no support for automation is currently available, both for proof search and for generating invariants.

6.3 Bytecode logics

Differently from us, most approaches work on traditional bytecode, albeit with certain restrictions and are based on weakest precondition calculi. Only two (67; 68) have a formalized verification. Bannwart and Müller (69) present a sound and complete bytecode logic for a fragment of sequential JVML – 15 instructions, although under some "well-formedness" restrictions, namely only virtual methods with one parameter, always yielding a value and ending with a `return`, which cannot occur anywhere else. The scenario is the context of proof-transforming compilers, i.e. compilers that modify source-level correctness proofs. Thus, the system shares the object model and many of the structural rules with (43). The unstructured nature

of bytecode is dealt with by the notion of *instruction specification* and its WP calculus, following closely (70). Quigley (67) is mainly concerned with reconstructing high-level control structures such as loops in the bytecode: hence the Hoare rules are overly complex, limited (no method calls) and not appropriate for a PCC scenario. In (68), the authors instantiate the PCC framework of (71) to Jinja bytecode (a dozen instructions including exception handling) and a safety policy concerning arithmetic overflow. Provability is defined semantically and a generic sound and complete VCG based on flow-graph analysis produces annotations in a dedicated assertion language, basically first-order arithmetics with stack primitives. As the framework is generic, other safety polices are possible, provided an appropriate language satisfying certain restrictions is given. The user is required to provide the invariants, although (72) presents preliminary results to automatically extract annotations from untrusted static analysers (in this case, interval analysis). Finally, Moore (73) used ACL2 to verify the correctness of simple programs in a fragment of the JVM, directly from the operational semantics, by encoding an interpreter for the byte code as a LISP function. In (74) the same approach was used to prove some meta-theoretical properties of the JVM.

7 Conclusion

7.1 Summary

In this article we have presented program logics for a compact representation of virtual machine languages, based on a formalisation in the theorem prover Isabelle/HOL. Founded on an operational semantics that includes a flexible notion of resources, we have first discussed a sound and complete logic of partial correctness, in which pre- and post-conditions are combined to yield specifications that are relations over the components of the operational semantics. We have presented derived rules for mutually recursively program fragments and method invocation with parameter adaptation, with proofs based on (also derived) cut rules. The usefulness of these rules was demonstrated on non-trivial examples, including the verification of quantitative specifications and other non-functional properties.

In the second part of this article, we have exhibited a logic for termination which relies on partial correctness assertions formulated in the core logic as side conditions of the rules for let and if constructs. Thus we achieve a modular treatment of termination, which leaves the underlying partial logic unchanged. This facilitates the hierarchical design of logics as championed in this paper. Noteworthy technical features of the termination logic are the use of an implicit context, which simplifies the judgement, and that it does not need a special rule or proof system for mutual recursion, as justified formally by the completeness proof for the termination logic.

Thanks to the formalised proofs of soundness, both program logics may be used as a basis of a PCC system such as the one we have been developing in the context of the *Mobile Resource Guarantees* project (2). Our approach of implementing a hierarchy of logics addresses a critical issue in the design of PCC systems, the trade-off between expressiveness and automation. Indeed, the proofs of the example programs described in this article appear difficult to automate, and contain typically 100-200 proof commands. In order to exploit structure available in typed high-level languages, we are developing derived proof systems that are interpreted high-level type systems with respect to certain compilation strategies. As is the case in our first such logic (32), an interpretation of Hofmann-Jost's LFD system (33), the soundness proofs of such derived logics often hide the complexity of low-level proofs. Instantiations of (existential) quantifiers and unfolding of datatype representation predicates are performed during the derivation of the rules, whereas their (often: syntax-directed) application only involves side conditions that are computationally easy to discharge. Complementing the communication of typing derivations, the role of type systems in the process of certificate generation becomes that of deriving either invariants (in the formalisation of LFD: method specifications) or other hints that suffice for the prover to reconstruct proofs. Current and future work in this direction will aim to represent type systems for a variety of resources, and their modular combination.

Another research strand is concerned with extending Grail and its program logic to include more features of the JVML, including exceptions and threads. This is mostly relevant to the forthcoming ST FET Integrated Project MOBIUS (Mobility, Ubiquity and Security), see `http://mobius.inria.fr/`.

Following onto our work on the termination logic, we are generalising the key ideas to formalise other security properties in a similar framework, which we call "partial termination". The rationale is that classical termination might be a too strong property for the purpose of guaranteeing resource-bounded computation. It prohibits for example the treatment of demon processes that are designed to run forever, but should not consume resources. Furthermore, the necessity of defining a measure in the termination logic complicates automatic proofs of termination. The development of partial termination proceeds in two stages. First insecure behaviour is defined using an inductive axiomatisation. Then a logic for security is defined on top of this axiomatisation. For example, if insecurity is introduced by function calls to a certain class of functions, possibly under side-conditions of parameter size, we add an axiom for this case. All other rules in the logic are then oblivious to the underlying security policy and only propagate the information through the program. For most security policies it should be possible to re-use the same techniques for soundness and completeness proofs. They are likely to be simpler, though, since no explicit measure will be necessary.

As regards the resource algebras, we are currently exploring further ways of structuring the operations and exploring formal relations to static notions such as effects. As was remarked earlier, the traces collected by resource algebras such as $\mathcal{R}^{\mathsf{InvTr}}$ and $\mathcal{R}^{\mathsf{HpTr}}$ can be constrained in a more flexible way than using precise specifications such as f_{Even} and f_{Odd}. For example, Schneider's *EM policies* (18) (short for: security policies that are enforceable by monitoring system execution) and the associated specification formalism, *security automata*, appear well-suited to be formulated as (variants of) $\mathcal{R}^{\mathsf{InvTr}}$ or other resource algebras, which would yield a program logic in which satisfaction of EM policies can be certified. Similarly, Skalka's *history effects* (75) represent a static way to approximate properties of *event histories*, i.e. of abstractions of traces with respect to arbitrary observable properties. Interpreting history effects as labelled transition systems allows Skalka to employ model checking techniques to (statically) verify assertions modelled as formulae in a temporal logic. As was remarked in the introduction, the usage of type systems is at the heart of our approach to certificate generation, and the potential to exploit type-and-effect systems was indeed one of the motivations to consider a generalised form of resources. Finally, Beckert and Mostowski's *throughout* modality (76) requires the quantified property to be satisfied in all intermediate states of a computation. The satisfaction of this modality appears to follow a similar regime as the validation of the policy that enforces limits of parameter values, as a boolean variable may be used to signal violation. We thus expect our logic to be sufficiently expressive to capture *strong invariants* such as the example properties described in (56).

7.3 Acknowledgements

This research was supported by the MRG project (IST-2001-33149) which was funded by the EC under the FET proactive initiative on Global Computing. We would like to thank all of the researchers and who contributed to MRG, including D. Sannella, I. Stark, S. Gilmore, K. MacKenzie, O. Shkaravska, M. Prowse and M. Konečný. We also benefited from discussions with Peter Lee, Tobias Nipkow, and David von Oheimb.

References

[1] G. Necula, Proof-carrying Code, in: POPL'97 — Symposium on Principles of Programming Languages, ACM Press, Paris, France, January 15–17, 1997, pp. 106–116.

[2] D. Sannella, M. Hofmann, Mobile Resource Guarantees, EU OpenFET Project, `http://www.dcs.ed.ac.uk/home/mrg/` (2002).

[3] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, I. 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, Vol. 3362 of LNCS, Springer, 2005, pp. 1–26.
URL http://www.springerlink.com/link.asp?id=f0p7u1repyl4pxyg

[4] K. MacKenzie, N. Wolverson, Camelot and Grail: resource-aware functional programming on the jvm, in: Trends in Functional Programing, Vol. 4, Intellect, 2004, pp. 29–46.

[5] L. Beringer, K. MacKenzie, I. Stark, Grail: a functional form for imperative mobile code, in: V. Sassone (Ed.), Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop, no. 85.1 in Electronic Notes in Theoretical Computer Science, Elsevier, 2003.
URL http://homepages.inf.ed.ac.uk/stark/graffi.html

[6] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen, The essence of compiling with continuations, in: Proceedings ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993, Vol. 28(6), ACM Press, New York, 1993, pp. 237–247.
URL citeseer.ist.psu.edu/flanagan93essence.html

[7] C. League, V. Trifonov, Z. Shao, Functional Java bytecode, in: Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics, 2001, pp. 1–6, Workshop on Intermediate Representation Engineering for the Java Virtual Machine.
URL http://flint.cs.yale.edu/flint/publications/lamjvm.pdf

[8] A. W. Appel, Compiling with Continuations, Cambridge University Press, 1992.

[9] L. Beringer, K. MacKenzie, I. Stark, Grail: a functional form for imperative mobile code, in: Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop, no. 85.1 in Electronic Notes in Theoretical Computer Science, Elsevier, 2003, pp. 1–21.
URL http://www.ed.ac.uk/ stark/graffi.html

[10] X. Leroy, Bytecode verification for Java smart cards, Software Practice and Experience 32 (4) (2002) 319–340.
URL http://pauillac.inria.fr/ xleroy/publi/oncard-verifier-spe.pdf

[11] B. Grobauer, Cost recurrences for DML programs, in: International Conference on Functional Programming, 2001, pp. 253–264.
URL citeseer.ist.psu.edu/grobauer01cost.html

[12] C. B. Jay, M. Cole, M. Sekanina, P. Steckler, A monadic calculus for parallel costing of a functional language of arrays, in: Proceedings of the Third International Euro-Par Conference on Parallel Processing, Springer-Verlag, 1997, pp. 650–661.

[13] D. Apinall, L. Beringer, A. Momigliano, Generalised resources for grail, draft (2005).

[14] A. Chander, J. Mitchell, I. Shin, Mobile code security by Java bytecode instrumentation, in: In DARPA Information Survivability Conference & Exposition (DISCEX II), 2001, pp. 1–15.
URL citeseer.ist.psu.edu/chander01mobile.html

[15] A. Chander, D. Espinosa, N. Islam, P. Lee, G. C. Necula, Enforcing resource

bounds via static verification of dynamic checks., in: Sagiv (77), pp. 311–325.
[16] G. Czajkowski, T. von Eicken, JRes: a resource accounting interface for Java, SIGPLAN Not. 33 (10) (1998) 21–35.
[17] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, A. Momigliano, A program logic for resource verification, in: Proceedings of 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004), Vol. 3223 of LNCS, Springer, Heidelberg, 2004, pp. 34–49.
[18] F. B. Schneider, Enforceable security policies., ACM Transactions on Information and System Security 3 (2000) 30–50.
[19] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.
[20] C. Jones, Systematic Software Development Using VDM, Prentice Hall, 1990.
[21] T. Kleymann, Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs, Ph.D. thesis, LFCS, University of Edinburgh (1999).
URL http://www.lfcs.informatics.ed.ac.uk/reports/98/ECS-LFCS-98-392/ECS-LFCS-9
[22] T. Nipkow, Hoare logics in Isabelle/HOL, in: H. Schwichtenberg, R. Steinbrüggen (Eds.), Proof and System-Reliability, Kluwer, 2002, pp. 341–367.
[23] E. W. Dijkstra, A Displicipline of Programming, Prentice-Hall, 1976.
[24] C. Pierik, F. S. de Boer, Modularity and the rule of adaptation., in: Rattray et al. (78), pp. 394–408.
[25] T. Nipkow, Hoare Logics for Recursive Procedures and Unbounded Nondeterminism, in: J. C. Bradfield (Ed.), Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Proceedings, Vol. 2471 of LNCS, Springer, 2002, pp. 103–119.
[26] M. Hofmann, Semantik und Verifikation, Lecture Notes, TU Darmstadt (1998).
[27] S. A. Cook, Soundness and completeness of an axiom system for program verification, SIAM J. Comput. 7 (1) (1978) 70–90, see Corrigendum in SIAM J. Comput. 10, 612.
[28] G. A. Gorelick, A complete axiomatic system for proving assertions about recursive and non-recursive programs, Tech. Rep. 75, University of Toronto (1975).
[29] P. Homeier, Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures, Ph.D. thesis, University of California (1995).
[30] S. Sokołowski, Total Correctness for Procedures, in: J. Gruska (Ed.), Mathematical Foundations of Computer Science, Vol. 53 of LNCS, Springer, 1977, pp. 475–483.
[31] A. W. Appel, Foundational proof-carrying code, in: LICS'01 — 16th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2001, pp. 247–258.
URL http://www.cs.princeton.edu/ appel/papers/fpcc.pdf
[32] L. Beringer, M. Hofmann, A. Momigliano, O. Shkaravska, Automatic certifi-

cation of heap consumption, in: A. V. Franz Baader (Ed.), Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005. Proceedings, Vol. 3452 of LNCS, Springer, 2005, pp. 347–362.

[33] M. Hofmann, S. Jost, Static Prediction of Heap Space Usage for First-Order Functional Programs, in: POPL'03 — Symposium on Principles of Programming Languages, ACM Press, New Orleans, LA, USA, 2003, pp. 185–197.

[34] P. H. Hartel, L. Moreau, Formalising the Safety of Java, the Java Virtual Machine and Java Card, ACM Computing Surveys 33 (4) (2001) 517–558.
URL http://www.ecs.soton.ac.uk/ lavm/papers/acmcs.pdf

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

[36] F. Tang, M. Hofmann, Generating Verification Conditions for Abadi and Leino's Logic of Objects, in: FOOL9 — Ninth International Workshop on Foundations of Object-Oriented Languages, Portland, OR, 2002, pp. 1–11.
URL http://www.dcs.ed.ac.uk/home/fhlt/docs/ht-fool9.pdf

[37] F. de Boer, A WP-calculus for OO, in: Foundations of Software Science and Computation Structures, Vol. 1578 of LNCS, Springer, 1999, pp. 135–149.

[38] C. Pierik, F. S. de Boer, A syntax-directed Hoare logic for object-oriented programming concepts., in: E. Najm, U. Nestmann, P. Stevens (Eds.), FMOODS, Vol. 2884 of LNCS, Springer, 2003, pp. 64–78.

[39] F. d. Boer, C. Pierik, Computer-aided specification and verification of annotated object-oriented programs, in: B. Jacobs, A. Rensink (Eds.), Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), Vol. 209 of IFIP Conference Proceedings, Kluwer, 2002, pp. 163–177.

[40] E. Ábrahám, F. S. de Boer, W. P. de Roever, M. Steffen, An assertion-based proof system for multithreaded Java., Theor. Comput. Sci. 331 (2-3) (2005) 251–290.

[41] M. Abadi, R. Leino, A Logic of Object-Oriented Programs, in: TAPSOFT '97: Theory and Practice of Software Development, Vol. 1214 of LNCS, Springer, 1997, pp. 682–696.

[42] R. Leino, Recursive Object Types in a Logic of Object-oriented Programs, Nordic Journal of Computing 5 (4) (1998) 330–360.

[43] P. Müller, A. Poetzsch-Heffter, A Programming Logic for Sequential Java, in: ESOP'99 — European Symposium on Programming, LNCS 1576, 1999, pp. 162–176.

[44] J. Meyer, P. Müller, A. Poetzsch-Heffter, The JIVE system—implementation description, available from www.informatik.fernuni-hagen.de/pi5/publications.html (2000).

[45] J. van den Berg, B. Jacobs, The LOOP compiler for Java and JML, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), LNCS, Springer, 2001, pp. 299–315.

[46] M. Huisman, B. Jacobs, Java Program Verfication via a Hoare Logic with Abrupt Termination, in: FASE'00 — Fundamental Approaches to Software

Engineering, LNCS 1783, 2000, pp. 284–303.

[47] B. Jacobs, E. Poll, A logic for the Java Modeling Language JML, in: FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering, Springer, London, UK, 2001, pp. 284–299.

[48] C. Marché, C. Paulin-Mohring, X. Urbain, The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML, Journal of Logic and Algebraic Programming 58 (1–2) (2004) 89–106.
URL http://krakatoa.lri.fr

[49] J.-C. Filliâtre, Why: a multi-language multi-prover verification tool, Research Report 1366, LRI, Université Paris Sud (Mar. 2003).
URL http://www.lri.fr/ filliatr/ftp/publis/why-tool.ps.gz

[50] L. Burdy, A. Requet, J.-L. Lanet, Java applet correctness: A developer-oriented approach, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003: Formal Methods: International Symposium of Formal Methods Europe, Vol. 2805 of LNCS, Springer, 2003, pp. 422–439.

[51] L. Burdy, M. Pavlova, Java bytecode specification and verification, manuscript (2005).

[52] G. Barthe, M. Pavlova, G. Schneider, Precise analysis of memory consumption using program logics, manuscript (2005).

[53] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P. H. Schmitt, The KeY tool, Software and Systems Modeling 4 (1) (2005) 32–54.

[54] B. Beckert, A dynamic logic for the formal verification of Java Card programs, in: JavaCard '00: Revised Papers from the First International Workshop on Java on Smart Cards: Programming and Security, Springer, London, UK, 2001, pp. 6–24.

[55] W. Mostowski, Formalisation and verification of Java Card security properties in Dynamic Logic, in: M. Cerioli (Ed.), Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland, Vol. 3442 of LNCS, Springer, 2005, pp. 357–371.

[56] R. Hähnle, W. Mostowski, Verification of safety properties in the presence of transactions, in: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, T. Muntean (Eds.), Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop, Vol. 3362 of LNCS, Springer, 2005, pp. 151–171.

[57] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, A. Thums, Formal system development with KIV, in: FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, London, UK, 2000, pp. 363–366.

[58] K. Stenzel, A formally verified calculus for full Java Card., in: Rattray et al. (78), pp. 491–505.

[59] D. R. Cok1, J. R. Kiniry, Esc/Java2: Uniting ESC/Java and JML. progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an internet voting tally system, in: G. Barthe, L. Burdy, M. Huisman, et al. (Eds.), Construction and Analysis

of Safe, Secure, and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004, no. 3362 in LNCS, Springer, 2005, pp. 108–129.

[60] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, Washington, DC, USA, 2002, pp. 55–74.

[61] D. C. Luckham, N. Suzuki, Verification of array, record, and pointer operations in Pascal, ACM Transactions on Programming Languages and Systems 1 (2) (1979) 226–244.

[62] K. R. M. Leino, Toward reliable modular programs, Ph.D. thesis, California Institute of Technology, available as Technical Report Caltech-CS-TR-95-03. (1995).

[63] R. Bornat, Proving pointer programs in Hoare logic, in: MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction, Springer, London, UK, 2000, pp. 102–126.

[64] F. Mehta, T. Nipkow, Proving pointer programs in higher-order logic, in: F. Baader (Ed.), Automated Deduction — CADE-19, Vol. 2741 of LNCS, Springer, 2003, pp. 121–135.

[65] M.J.C. Gordon, Mechanizing programming logics in higher-order logic, in: G.M. Birtwistle, P.A. Subrahmanyam (Eds.), Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification), Springer, Banff, Canada, 1988, pp. 387–439.

[66] T. Weber, Towards mechanized program verification with separation logic, in: J. Marcinkowski, A. Tarlecki (Eds.), Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings, Vol. 3210 of LNCS, Springer, 2004, pp. 250–264.

[67] C. L. Quigley, A programming logic for Java bytecode programs., in: D. A. Basin, B. Wolff (Eds.), TPHOLs, Vol. 2758 of LNCS, Springer, 2003, pp. 41–54.

[68] M. Wildmoser, T. Nipkow, Asserting bytecode safety., in: Sagiv (77), pp. 326–341.

[69] F. Y. Bannwart, P. Müller, A logic for bytecode, in: Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE), Electronic Notes in Theoretical Computer Science, Elsevier, 2005, pp. 233–250, to appear.

[70] N. Benton, A typed logic for stacks and jumps, Microsoft Research (2004).

[71] M. Wildmoser, T. Nipkow, Certifying machine code safety: Shallow versus deep embedding, in: K. Slind, A. Bunker, G. Gopalakrishnan (Eds.), Theorem Proving in Higher Order Logics (TPHOLs 2004), Vol. 3223 of LNCS, Springer, 2004, pp. 305–320.

[72] M. Wildmoser, A. Chaieb, T. Nipkow, Bytecode analysis for proof carrying code, in: Proceedings of the 1st Workshop on Bytecode Semantics, Verification and Transformation, Electronic Notes in Computer Science, 2005, pp. 16–30.

URL http://www4.in.tum.de/ wildmosm/publications/WildmoserCN-Bytecode05.pdf
[73] J. S. Moore, Inductive assertions and operational semantics., in: D. Geist, E. Tronci (Eds.), Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings, Vol. 2860 of LNCS, Springer, 2003, pp. 289–303.
[74] J. S. Moore, Proving theorems about Java and the JVM with ACL2, NATO Science Series Sub Series III Computer and Systems Sciences 191 (2003) 227–290.
[75] C. Skalka, S. F. Smith, History effects and verification., in: W.-N. Chin (Ed.), APLAS, Vol. 3302 of LNCS, Springer, 2004, pp. 107–128.
[76] B. Beckert, W. Mostowski, A program logic for handling Java Card's transaction mechanism, in: M. Pezzè (Ed.), Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2003, Warsaw, Poland, Vol. 2621 of LNCS, Springer, 2003, pp. 246–260.
[77] S. Sagiv (Ed.), Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings, Vol. 3444 of LNCS, Springer, 2005.
[78] C. Rattray, S. Maharaj, C. Shankland (Eds.), Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Proceedings, Vol. 3116 of LNCS, Springer, 2004.