Certification of Quantitative Properties of Programs [1]

Martin Hofmann, Hans-Wolfgang Loidl, Lennart Beringer
Institut für Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany

Abstract. In the context of mobile and global computing knowledge of quantitative properties of programs is particularly important. Here are some typical scenarios:

- A provider of distributed computational power may only be willing to offer this service upon receiving dependable guarantees about the required resource consumption.
- A user of a handheld device, wearable computer, or smart card might want to know that a downloaded application will definitely run within the limited amount of memory available.
- Third-party software updates for mobile phones, household appliances, or car electronics should come with a guarantee not to set system parameters beyond manufacturer-specified safe limits.

Requiring certificates of specified resource consumption will also help to prevent mobile agents from performing denial of service attacks using bona fide host environments as a portal. These lecture notes describe how such quantitative resource-related properties can be inferred automatically using type systems and how the results of such analysis can be turned into unforgeable certificates using a proof-carrying code framework.

Keywords. Type systems, Proof-carrying-code, program logics.

1. Introduction

These notes describe methods for inferring and certifying quantitative bounds on computing resources. Resources of particular interest are memory allocation, execution time, number of files or network connections, parameters to system calls, etc. By *certification* we understand the automatic creation of unforgeable certificates of the purported resource bound in the sense of *proof carrying code* (PCC [22]), see G. Necula's contribution to this volume.

In the context of mobile computing and computing on small devices such as embedded systems, mobile phones, smart cards present a host of applications for such an infrastructure. A user of a mobile phone might need to know for sure that a certain application provided by a third party does not allocate more memory than announced in the specification and furthermore obeys a specified limit on the number and cost of network connections made. A provider of shared computing power might only be willing to grant

[1]This material is based on results of the recently completed EU-funded project "Mobile Resource Guarantees" IST-2001-33149.

access to their system upon presentation of a certificate as to the total execution time and space consumption.

Our main tools will be type systems for automatically inferring resource bounds and formalised program logics to write down certificates in the sense of PCC. We now describe these two key techniques informally.

1.1. Type systems

Verification of even simple properties of arbitrary programs is infeasible and undecidable in general. On the other hand, the verification of simple properties of sensibly written programs can be quite easy. *Type systems* highlight such simple properties of sensibly written programs. Here are some examples of everyday type systems:

- The Java type system prevents uncaught exceptions, certain segmentation faults, etc. It also guarantees survival of the subsequent bytecode verification.
- The ML type system often prevents one from supplying arguments to a function in the wrong order etc. Slogan: "If it typechecks it works".
- *Abstract datatypes* promote modularity by preventing breaches of abstraction

In the research literature type systems for many more properties have been proposed, e.g., security of information flow, absence of array bound violation, and indeed resource bounds, to name a few.

A common feature of type systems is that they rely on program annotations, the types, whose well-formedness is given by an inductive definition, the typing rules. Checking whether a given annotation of a program meets the typing rules is known as type checking. For many type systems all or large parts of the typing annotations can be determined automatically in a most general fashion. This is known as *type inference* and is available for the ML type system, but not for Java's type system.

As opposed to *program analysis* [24] type systems try to give some feedback to the programmer in the form of types and also type error messages in case the inference fails. On the other hand, program analyses are often more flexible and powerful. Having said that, type systems and program analysis are often just two sides of the same coin and have been translated into each other [16, 10], indeed, the type systems that we describe below in Section 5 was originally construed as a program analysis.

An excellent reference for type systems is [26].

1.2. Program Logics

A *program logic* (for object code) is a set of syntax-directed rules for deriving statements of the following form:

If in heap h and stack E program phrase e terminates with result v and post-heap h' then $(h,E,v,h') \in \phi$.

This is written $e : \phi$. Typical proof rules for this judgement are as follows:

$$\frac{x.m() : \phi \Longrightarrow e_m(x) : \phi}{x.m() : \phi}$$

$$\frac{e_1 : E[x]{=}\mathsf{true} \Rightarrow \phi \quad e_2 : E[x]{=}\mathsf{false} \Rightarrow \phi}{\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \phi}$$

Program logics are usually complete in the sense that all semantically valid judgements are derivable using the proof rules, *but* some rules have semantic side conditions of the form $\phi_1 \subseteq \phi_2$ which may be arbitrarily difficult to establish. This is known as *relative completeness*. It says that the proof rules adequately break down proof obligations according to the program structure.

Often program logics use a *Hoare style* format [11] where the statements of interest take the form

"If in heap h and stack E such that $(h,E) \in \phi$ program phrase e terminates with result v and post-heap h' then $(v,h') \in \psi$."

This is traditionally written $\{\phi\}e\{\psi\}$ and called a *Hoare triple*. In order to relate ϕ (the *precondition*) to the ψ (the *postcondition*) one must use auxiliary variables, i.e., consider assertions of the form $\forall\vec{x}.\{\phi(\vec{x})\}e\{\psi(\vec{x})\}$. With this generalisation an assertion $e : \phi$ is equivalent to the Hoare triple $\forall h_1, E_1.\{h = h_1 \wedge E = E_1\}e\{\phi(E_1, h_1, v, h')\}$. Conversely, the Hoare triple $\forall\vec{x}.\{\phi(\vec{x})\}e\{\psi(\vec{x})\}$ is equivalent to the assertion $e : \forall\vec{x}.\phi(\vec{x}, E, h) \Rightarrow \psi(\vec{x}, v, h')$.

We prefer the single-assertion format, also known as VDM-style [14, 15], to Hoare-style because it does not need auxiliary variables and rules to manipulate them.

If a program logic is formalised in a theorem prover then proofs of program properties are unforgeable certificates.

The rest of this lecture note is organised as follows: in Section 2 we describe a first-order functional programming language, Camelot, that contains primitives for memory management in the style of C/C++ and thus enables more accurate predictions of space usage than a garbage-collected language such as ML or Java does. Camelot is compiled into an abstract form of a subset of Java bytecode, called Grail.

In Section 2 the runtime behaviour of Grail is defined as a resource-aware operational semantics which we take as the formal model of execution and resource usage. Statements in the program logic to be described in Section 4 will refer to this operational semantics. On top of this program logic we will then define a derived logic, which reflects the structure of a high-level type system for the resource under consideration (Section 5).

2. Camelot and Grail

While certification must take place on the low level of bytecode or machine code, type checking and inference are easier to perform on high-level code which displays more structure than the low-level code it is compiled into and also restricts flow of control to certain patterns as opposed to arbitrary "spaghetti code". Of course, it is then necessary to transfer the results of type inference across the compilation process so as to aid the generation of certificates. This transfer process is the subject of Section 5. Here we present as a concrete example one high-level language, Camelot, which is a variant of Objective CAML that has memory primitives in the style of C/C++ and is compiled to Java Bytecode (JVM).

Let us begin with a tail-recursive version of the factorial function

```
val fac: int -> int
```

```
static int fac(int);
  Code:
   0:   iconst_1
   1:   istore_1
   2:   iload_0
   3:   iconst_1
   4:   if_icmplt 18
   7:   iload_1
   8:   iload_0
   9:   imul
   10:  istore_1
   11:  iload_0
   12:  iconst_1
   13:  isub
   14:  istore_0
   15:  goto    2
   18:  iload_1
   19:  ireturn
```

```
static int fac(int);
  Code:
    0:   $0 = 1
    1:   b = $0
f    :   $0 = n
    3:   $1 = 1
    4:   if ($0 < $1) then f_then else f_else
f_else:  $0 = b
    8:   $1 = n
    9:   $0 = $0 * $1
    10:  b = $0
    11:  $0 = n
    12:  $1 = 1
    13:  $0 = $0 - $1
    14:  n = $0
    15:  goto    f
f_then:  $0 = b
    19:  ireturn $0
```

Figure 1. Java bytecode in ordinary (left) and beautified (right) form

```
let rec f n b =
   if n < 1 then b else f (n - 1) (n * b)
in f n 1
```

In Camelot tail-recursive functions are understood as and translated into loops. Accordingly, this Camelot program corresponds to the following Java program

```
class Fac {
    static int fac(int n) {
        int b = 1;
        while (n >= 1) {
            b = b * n;
            n = n - 1;
        }
        return b;
    }}
```

The Java Bytecode corresponding to either the Camelot program or the Java program is given in the first column of Figure 1.

Recall that many JVM commands refer to the operand stack. If we explicitly denote the items on this stack by `$0, $1, $2,...`, starting from the top, then we obtain the beautified version of the bytecode given in the right column of Fig. 1.

In *Grail*, our version of JVM, we take this one step further by removing the stack altogether and allowing arithmetic operations on arbitrary variables. Moreover, we use a functional notation for jumps and local variables as exemplified by the code in the left column of Figure 2 which corresponds to the above JVM bytecode. Here is a genuinely recursive version of the factorial in Camelot:

```
val fac: int -> int
let rec fac n =
   if n < 1 then 1 else n * fac (n - 1)
```

The corresponding Grail code is found in the right column of Figure 2.

3. Syntax and semantics of Grail

Grail is an intermediate language that corresponds bijectively to a subset of well-structured bytecode. It is presented using an (impure) functional notation in order to sim-

```
method static int fac (int n) =
 let
    val b = 1
    fun f(int n, int b) =
         if n<1 then b
         else f_else(n,b)

    fun f_else(int n, int b) =
    let
         val b = mul b n
         val n = sub n 1
    in
       f(n,b)
    end
 in
    f(n,b)
 end
```

```
method static int fac (int n) =
let
  fun f_else(n) =
  let
      val n' = sub n 1
      val n' = invokestatic <Fac Fac.fac(int)> (n')
  in mul n n'
  end
in
  if n<1 then 1
  else f_else(n)
end
```

Figure 2. Grail versions of iterative, tail-recursive (left) and genuinely recursive (right) versions of the factorial

plify compilation from the source language Camelot. At the global level, Grail retains the JVM class and method structure, while method bodies themselves are represented as a collection of mutually first-order functions that correspond to basic blocks. The dynamic semantics is given by a big-step operational semantics that combines functional elements (evaluation environments) with imperative aspects (initial and final heaps). Grail admits a reversible expansion of expressions into JVM code such that let-bound variables can be interpreted as local registers and function calls as jumps. Indeed, the operational semantics coincides with the standard interpretation of JVM code for this expansion, provided some mild syntactic conditions are met: function calls may only appear in tail position, and function arguments must be variables and moreover coincide syntactically with the formal parameters of the invoked function. General recursion is available via method invocations whose occurrence is not limited to tail positions.

As a further component, judgements include a resource component that collects the cumulative consumption of computational resources such as the number of executed instructions or the maximal height of the frame stack observed during an execution. Resources that cannot be modelled in this cumulative way (like the number of currently open files) could in principle be modelled as further components of the heap, i.e. as if there were a global variable that contained the current value of the resource.

3.1. Syntax

Formally, Grail programs are formulated over the mutually disjoint categories $\mathcal{M}$ of method names $m, \ldots$, $\mathcal{C}$ of class names $c, \ldots$, $\mathcal{F}$ of function names (i.e. labels of basic blocks) $f, \ldots$, $\mathcal{T}$ of (virtual or static) field names $t, \ldots$, and $\mathcal{X}$ of variables $x, \ldots$, where $\texttt{self} \in \mathcal{X}$.

Neither the syntax nor the operational semantics of Grail mentions the operand stack explicitly.

For the remainder of these lecture notes, the global structure of a Grail program is represented by a *method table MT*, i.e. a finite map associating method declarations to $\mathcal{C} \times \mathcal{M}$-pairs. A method declaration $MT(c.m)$ consists of

- a list of (distinct) variables, the formal method parameters,
- a function table giving the definition of the basic blocks, and
- a Grail expression representing the initial basic block.

We identify these components as $pars_{c.m}$, $FT_{c.m}$, and $body_{c.m}$, respectively. The first method parameter is always the variable `self`. Function tables FT are also finite maps,

e	$\llbracket e \rrbracket$
`null`	`aconst_null`
`imm` i	`iconst` i
`var` x	`xload` x
`prim` $op\ x\ y$	`xload` x; `xload` y; $\llbracket op \rrbracket$
`new` $c\ [\overline{t_i := x_i}]$	`new` c; `dup`; `xload` x_1; ...; `xload` x_n; `invokespecial` $c\ m$
$x.t$	`aload` x; `getfield` t
$x.t{:=}y$	`aload` x; `xload` y; `putfield` t
$c \diamond t$	`getstatic` $c\ t$
$c \diamond t{:=}x$	`xload` x; `putstatic` $c\ t$
`let` $x{=}e_1$ `in` e	$\llbracket e_1 \rrbracket$; `xstore` x; $\llbracket e_2 \rrbracket$
e_1 ; e_2	$\llbracket e_1 \rrbracket$; $\llbracket e_2 \rrbracket$
`if` x `then` e_1 `else` e_2	`xload` x; `ifeq` TRUE f; $\llbracket e_2 \rrbracket$ where $f : \llbracket e_1 \rrbracket$
`call` f	`goto` f
$x \cdot m(a_1, \ldots, a_n)$	`aload` x; $\llbracket a_1 \rrbracket$; ...; $\llbracket a_n \rrbracket$; `invokevirtual` $c\ m$
$c.m(a_1, \ldots, a_n)$	$\llbracket a_1 \rrbracket$; ...; $\llbracket a_n \rrbracket$; `invokestatic` $c\ m$

Figure 3. Translation from Grail to JVM Bytecode

associating formal parameters (lists of distinct variables) and function bodies (expressions) to function names.

The syntax of expressions is given by the following grammar, where overbars represent (possibly empty) list of items.

$$e \in expr ::= a \mid \texttt{prim}\ op\ x\ x \mid \texttt{new}\ c\ [\overline{t_i := x_i}] \mid x.t \mid x.t{:=}x \mid c \diamond t \mid c \diamond t{:=}x \mid$$

$$\texttt{let}\ x{=}e\ \texttt{in}\ e \mid e\ ;\ e \mid \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{call}\ f \mid x \cdot m(\overline{a}) \mid c.m(\overline{a})$$

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

The first expression form injects constants (i ranges over numeric constants) and variables into the category of expressions. An expression `prim` $op\ x\ y$ represents the application of the binary primitive operation op to the arguments held in local registers x and y, where op is expected to correspond to a binary JVM instruction $\llbracket op \rrbracket$, as in $\llbracket \lambda\, x\, y.\ x + y \rrbracket =$ `iadd`. Object creation (instruction `new`) may include the initialisation of fields $t_1, \ldots, t_n$ with the content of registers $x_1, \ldots, x_n$. The next four instructions represent access and modification operations for instance and static fields, respectively. Instruction composition may either involve the storing of a value in a register or be performed anonymously. In the first case, the named composition `let` $x{=}e_1$ `in` e_2 is used, which updates register x with the result of evaluating e_1. In the latter case, the composition e_1 ; e_2 is used, where the constituent phrases are merely juxtaposed. This form of composition is appropriate if the evaluation of the JVM expansion of e_1 would not leave a value on the operand stack, as is the case in a field modification. The syntax of conditionals is unsurprising. As the arguments of function calls syntactically are understood to be precisely the formal parameters in the definition of the functions, the syntax for function calls does not include arguments at all. Finally, virtual and static method invocations are represented by instructions $x \cdot m(\overline{a})$ and $c.m(\overline{a})$, respectively, where arguments $\overline{a}$ can be variables or immediates.

While we omit a formalisation of the condition that function calls occur only at tail position (and of typing conditions), an intuitive interpretation of Grail programs can be obtained from the informal expansion $\llbracket . \rrbracket$ into JVM code given in Fig. 3. The prefix `x` indicates the type-dependency of the corresponding instruction. Here, the notation $f : \llbracket e \rrbracket$

$\mathit{dom}\ h$	the domain of the object heap of h
$h(l).t$	the content of field t of the object at location l in the object heap of h
$h[l.t \mapsto v]$	modification of the above instance field
$h(c).t$	the content of the field $c.t$ in the static heap of h
$h[c.t \mapsto v]$	modification of the static field $c.t$
$h[l \mapsto (c, \{\overline{t_i := v_i}\})]$	modification of object heap of h at location l, where the inserted object has class c and fields $t_1, \ldots, t_n$ with values $v_1, \ldots, v_n$, respectively
$\mathit{freshloc}(h)$	returns a location l with $l \notin \mathit{dom}\ h$
$\mathit{classOf}(h(v)) = c$	the dynamic class of the object at v in h is c

Figure 4. Notations for heaps and environments

is used to indicate and $FT(f) = e$ holds, where FT is the function table of the current method.

As a consequence of the structure of Grail expressions, the JVM operand stack *after* the execution of $[\![e]\!]$ is identical to the operand stack *prior* to its execution. Since operand stacks are empty prior to the first instruction of a method, the same condition is true between any two (expansions of) Grail expressions. This property strengthens a condition identified by Leroy [18] as being beneficial for the on-card verification of Java bytecode programs.

3.2. Operational semantics

The dynamic semantics is given by judgements $E \vdash h, e \Downarrow h', v, p$, to be read as: *evaluating e in variable environment E and initial heap h yields the result value v and final heap h′, consuming p resources.* As is characteristic for a big-step semantics, non-termination is modelled by the absence of a derivation rather than the existence of an infinite sequence of one-step reductions as would be the case for a small-step semantics. The semantic domains of the components occurring in operational judgements are as follows.

Values $v \in \mathcal{V}$ can either be references (tagged locations $\mathsf{Ref}\, l$ or null) or integers i

Environments $E \in \mathcal{E}$ are finite maps from variables to values. We let $\mathit{dom}\ E$ denote the domain of E, $E\langle x \rangle$ the access operation (where $x \in \mathit{dom}\ E$), and $E\langle x := v\rangle$ the update operation

Heaps $h \in \mathcal{H}$ consist of an object heap and a static heap. Object heaps are finite maps from locations to objects, where objects consist of a (dynamic) class name and a map from field names to values. Static heaps map class names to maps from field names to values. We use the abbreviations given in Figure 4

Resource counters $p \in \mathcal{R}$ can be values from specific sets that are endowed with some constants and two operations $\oplus$ (addition) and $\smile$ (maximum). For the purpose of these lecture notes, we restrict our attention to the resource component $\mathcal{R} = \mathbb{N}^4$ consisting of four counters. The four components of a resource tuple

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

summarise (in that order) the elapsed time (the approximate number of instructions executed), the number of function calls (jumps), the number of method invocations, and the maximal height of the frame stack observed during an execution. For $p = \langle p_1 \quad p_2 \quad p_3 \quad p_4 \rangle$ and $q = \langle q_1 \quad q_2 \quad q_3 \quad q_4 \rangle$, we define the operators $\oplus$ and $\smile$ by $p \oplus q = \langle p_1 + q_1 \quad p_2 + q_2 \quad p_3 + q_3 \quad p_4 + q_4 \rangle$ and $p \smile q = \langle p_1 + q_1 \quad p_2 + q_2 \quad p_3 + q_3 \quad max(p_4, q_4) \rangle$, respectively.
Generalising from the given notion of resource tuples, it would not be difficult to model the consumption of other resources, such as the invocations of particular (native) methods, or the power consumption with respect to specific cost models. The above notion of resources does not include an explicit counter for heap consumption since this can be derived by comparing the sizes of the heap before and after execution.

The rules defining the relation $E \vdash h, e \Downarrow h', v, p$ are as follows.

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

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

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

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

$$\frac{}{E \vdash h, c \diamond t \Downarrow h, h(c).t, \langle 1\ 0\ 0\ 0 \rangle}\ (\text{GETST})$$

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

$$\frac{l = freshloc(h)}{E \vdash h, \texttt{new}\ c\ [\overline{t_i := x_i}] \Downarrow h[l \mapsto (c, \{\overline{t_i := E\langle x_i \rangle}\})], \mathsf{Ref}\, l, \langle (n+3)\ 0\ 0\ 0 \rangle}\ (\text{NEW})$$

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

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

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

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

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

$$\frac{Env(pars_{c.m}, \mathsf{null} :: \overline{a}, E) \vdash h, body_{c.m} \Downarrow h_1, v, p}{E \vdash h, c.m(\overline{a}) \Downarrow h_1, v, \langle (2 + \mid \overline{a} \mid)\ 0\ 1\ 1 \rangle \oplus p} \text{ (SINV)}$$

$$\frac{classOf(h(E\langle x \rangle)) = c \quad Env(pars_{c.m}, E\langle x \rangle :: \overline{a}, E) \vdash h, body_{c.m} \Downarrow h_1, v, p}{E \vdash h, x \cdot m(\overline{a}) \Downarrow h_1, v, \langle (4 + \mid \overline{a} \mid)\ 0\ 1\ 1 \rangle \oplus p} \text{ (VINV)}$$

In the rules, the first component of resource tuples gives an approximate notion of execution time, roughly motivated by the number of JVM instructions executed according to the expansion given above. The first eight rules evaluate constants and variables, evaluate primitive operations and perform static and virtual field access and modification. In all cases, the functional aspects of the rules are standard. The resource vectors indicate that only a constant number of JVM instruction is executed, that no jumps or method invocations are involved, and that the frame stack height is not affected. The rule for object allocation extends the heap at a fresh location and initialises the fields. The rules for conditionals evaluate the branch condition and promote the result and the resources of the corresponding branch, incrementing the instruction counter by two – one increment for the loading of x onto the operand stack and one for performing the branch evaluation. Composition using the *let* form applies if the evaluation of $[\![e_1]\!]$ leaves a value on the operand stack, whose storage in variable x is responsible for the increment in the instruction counter. Anonymous composition applies if $[\![e_1]\!]$ does not leave a value on the operand stack, and merely composes the costs of the constituent phrases. In both cases, the use of the operator $\smile$ indicates that the maximum frame stack height is taken, while the addition is applied in first three components of the resource tuples. The rule for function application simply continues with the evaluation of the function body, without creating a new environment. This corresponds precisely to the execution of a jump instruction. The costs are modified by charging for one anonymous instruction and one jump. The notation $FT\, f$ refers to the entry for label f in the function table associated with the currently method. Finally, the two rules for method invocation create (in the semantic function *Env*) a local environment ("frame") that contains precisely the values for the formal parameters. These are obtained by evaluating the arguments a_i in the environment of the caller, E, and assigning the result to parameter p_i. The increments in the instruction counters depend linearly on the number of arguments and include constants for frame construction and deconstruction. The difference between static and virtual methods arises from the loading of the object reference onto the operand stack and the dynamic method resolution. Furthermore, a (cumulative) method invocation is observed, and the maximal frame stack height is also incremented.

4. Program Logic

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 resource vector. In this section, we define a logic of partial correctness (i.e. in particular, non-terminating programs satisfy any assertion), and we will comment on a separate termination logic in Section 4.8.

4.1. Assertions and validity

Deviating from the syntactic separation into pre- and post-conditions typical for Hoare-style and VDM-style program logics [11, 14], 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.[1]

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

Compared to more traditional Hoare-style 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 4.3.

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 1 *(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.

[1] 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.

Definition 2 *(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.

4.2. 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.

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

$$\frac{}{\Gamma \rhd \texttt{imm}\,i : \lambda E\,h\,h'\,v\,p.\, h' = h \wedge v = i \wedge p = \langle 1\,0\,0\,0\rangle} \tag{VIMM}$$

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

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

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

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

$$\frac{}{\Gamma \rhd c\diamond t : \lambda E\,h\,h'\,v\,p.\, h' = h\ \wedge v = h(c).t\ \wedge p = \langle 1\,0\,0\,0\rangle} \tag{VGETST}$$

$$\frac{}{\Gamma \rhd c\diamond t{:=}y : \lambda E\,h\,h'\,v\,p.\, h' = h[c.t \mapsto E\langle y\rangle]\ \wedge v = \bot \wedge p = \langle 2\,0\,0\,0\rangle} \tag{VPUTST}$$

$$\frac{}{\begin{array}{r}\Gamma \rhd \texttt{new}\,c\ [\overline{t_i := x_i}] : \lambda E\,h\,h'\,v\,p.\,\exists l.\, l = \mathit{freshloc}(h) \wedge p = \langle (n+3)\,0\,0\,0\rangle \wedge \\ h' = h[l \mapsto (c, \{\overline{t_i := E\langle x_i\rangle}\})]\ \wedge\ v = \mathsf{Ref}\,l\end{array}} \tag{VNEW}$$

$$\frac{\Gamma \rhd e_1 : A_1 \qquad \Gamma \rhd e_2 : A_2}{\begin{array}{l}\Gamma \rhd \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \lambda E\,h\,h'\,v\,p.\,\exists p'.\, p = \langle 2\,0\,0\,0\rangle \oplus p' \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{true} \longrightarrow A_1\ E\ h\ h'\ v\ p') \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{false} \longrightarrow A_2\ E\ h\ h'\ v\ p') \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{true} \vee E\langle x\rangle = \mathsf{false})\end{array}} \tag{VIF}$$

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

$$\frac{\Gamma \rhd e_1 : A \qquad \Gamma \rhd e_2 : B}{\Gamma \rhd e_1\ ;\ e_2 : \lambda E\,h\,h'\,v\,p.\,\exists\ p_1\,p_2\,h_1.A\ E\ h\ h_1\ \bot\ p_1\ \wedge\ B\ E\ h_1\ h'\ v\ p_2\ \wedge\ p = p_1 \smile p_2} \tag{VCOMP}$$

$$\frac{\Gamma \cup \{(\texttt{call}\,f, A)\} \rhd \mathit{body}_f : \Theta(A,f)}{\Gamma \rhd \texttt{call}\,f : A} \tag{VCALL}$$

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

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

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

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:

$$\begin{array}{l}
\Theta(A,f) = \lambda E\, h\, h'\, v\, p.\, A\; E\; h\; h'\; v\; (\langle 1\;1\;0\;0\rangle \oplus p) \\
\Phi(A,c,m,\overline{a}) = \lambda E\, h\, h'\, v\, p. \\
\qquad\qquad \forall\, E'.\; E = Env(\texttt{self} :: pars_{c,m}, \mathsf{null} :: \overline{a}, E') \\
\qquad\qquad\quad \longrightarrow\; A\; E'\; h\; h'\; v\; (\langle (2 + \mid \overline{a} \mid)\; 0\; 1\; 1\rangle \oplus p) \\
\Psi(A,x,c,m,\overline{a}) = \lambda E\, h\, h'\, v\, p. \\
\qquad\qquad \forall\, E'\; l.\; (E'\langle x\rangle = \mathsf{Ref}\, l \;\wedge\; h(l) = c \;\wedge\; E = Env(\texttt{self} :: pars_{c,m}, x :: \overline{a}, E')) \\
\qquad\qquad\quad \longrightarrow\; A\; E'\; h\; h'\; v\; (\langle (4 + \mid \overline{a} \mid)\; 0\; 1\; 1\rangle \oplus p).
\end{array}$$

4.3. 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 modifies the resource vector to $\langle 1\;1\;0\;0\rangle \oplus p$. 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. See [15] for a detailed comparative discussion.

4.4. Soundness

In order to enable an inductive soundness proof we need to assign a semantic meaning to the auxiliary judgments involving nonempty contexts.

Definition 3 *(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 $E \vdash h, e \Downarrow_m h', v, p$ indicates that $E \vdash h, e \Downarrow h', v, p$ holds by a derivation of height at most m. We omit the formal definition.

The counter n in Definition 3 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 $\models_m\ e : A$.

Definition 4 *(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* $\models_n e : A$*.*

The following lemma is proved by induction on n.

Lemma 1 *For* $\models_n \Gamma$ *and* $\Gamma \cup \{\mathtt{call}\,f, A\} \rhd body_f : \Theta(A, f)$*, let*

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

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

Similar results hold for static and virtual method invocations. From this, the following result may be proved by rule induction.

Lemma 2 *If* $\Gamma \rhd e : A$ *then* $\forall n.\ \Gamma\ \models_n e : A$

Finally, the soundness statement is obtained from Lemma 2 by unfolding the definitions of (relativised) validity.

Theorem 1 *(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 $\models e : A$.

4.5. Admissible rules

The following *admissible* rules are helpful for concrete program verifications and also in the proof of completeness. In addition to the proof rules given Section 4.2, 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.

$$\frac{\Gamma \rhd e : A}{\Gamma \cup \Delta \rhd e : A}\ (\textsc{vweak}) \qquad \frac{\Gamma \rhd e : A \quad \forall\, d\, B.\ (d,B) \in \Gamma \longrightarrow \Delta \rhd d : B}{\Delta \rhd e : A}\ (\textsc{vctxt})$$

$$\frac{\{(d,B)\} \cup \Gamma \rhd e : A \quad \Gamma \rhd d : B}{\Gamma \rhd e : A}\ (\textsc{vcut}) \qquad \frac{\Gamma \models ST \quad (e,A) \in \Gamma}{\emptyset \rhd e : A}\ (\textsc{mutrec})$$

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

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

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., [25]) 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 5 *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 6 *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 4.2. Using rule VCUT, is not difficult to prove that this property is closed under sub-contexts.

Lemma 3 *If* $(e,A) \cup \Gamma \models ST$ *then* $\Gamma \models ST$.

Based on Lemma 3, 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 [27, 25, 15]. For example, Nipkow [25] 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 4 *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 3. The proof of rule ADAPTV is similar.

4.6. 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 [7] 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 [15], 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 Gorelick [9] 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 7 *(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:

$$\text{If } \models e : A \text{ and } SSpec(e)\ E\ h\ h'\ v\ p \text{ then } A\ E\ h\ h'\ v\ p. \qquad (1)$$

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

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

By induction on e, we prove

Lemma 5 *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 6 *We have* $\Gamma_{strong} \models ST_{strong}$.

The proof of this lemma proceeds by unfolding the definitions, using Lemma 5 in the claims for function calls and method invocations.

Next, we prove that

$$\Gamma_{strong} \models ST \text{ implies } \emptyset \rhd e : SSpec(e) \qquad (2)$$

for arbitrary specification table ST, by applying rule VCTXT, where the first premise is discharged by Lemma 5 (i.e. Γ is instantiated to Γ_{strong}) and the second premise is discharged by rule MUTREC.

Combining property (2) and Lemma 6 yields $\emptyset \rhd e : SSpec(e)$, from which

Theorem 2 *(Completeness) For any e and A,* $\models e : A$ *implies* $\emptyset \rhd e : A$.

follows by rule VCONSEQ and property (1).

4.7. Verification examples

The following examples show how to use the argument adaptation rules for method invocations, and how to specify and verify properties of resource consumption. 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 (callers

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

Figure 5. Code of method append

after callees). 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.

4.7.1. Append

Our first example is for the method LIST.append shown in Figure 5. The property we will prove is given in the specification table entry

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

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 beginning 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. 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.

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

Finally, the verification of

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

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

4.7.2. Flatten

To continue with another example program emitted by the Camelot compiler, Figure 6 shows the definition of a method that flattens a tree into a list.

Again, we define a specification table entry for $\mathsf{flatten}$,

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

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 $dom(h)$ remains unchanged. The datatype representa-

```
method LIST TREE.flatten(t) = call f
┌f  ↦ let v4 = t.TAG in                                                              ┐
│      let b = iszero v4 in                                                          │
│      if b then call f0 else call f1                                                │
│f0 ↦ let v4 = t.CONT in let tg = imm 2 in                                          │
│      let t = new LIST [TAG := tg] in let tg = imm 3 in                             │
│      new LIST [TAG := tg; HD := v4; TL := t]                                      │
│f1 ↦ let v3 = t.LEFT in let v2 = t.RIGHT in let v1 = TREE.flatten(var v3) in      │
└      let t = TREE.flatten(var v2) in LIST.append([var v1, var t])                  ┘
```

Figure 6. Code of method flatten

tion predicate $h \models_{tree(n,X)} v$ is defined in a similar way as the list predicate $h \models_{list(n,X)} v$, namely:

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

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

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

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, 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

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

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.

4.8. Termination

So far the program logic developed for Grail has been a logic for partial correctness. This simplifies the development of our core logic. In order to deal with termination, we develop a separate program logic, with the judgement $\rhd_T \{P\}\ e \downarrow$, to be read as "expression e terminates under pre-condition P." More formally,

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

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 $\mathcal{P} \equiv \mathcal{E} \to \mathcal{H} \to \mathcal{B}$.

Since this termination logic is put on top of the existing partial correctness logic, we can use judgements of the latter in the side conditions of the former. In particular, the let rule uses such a side condition to relate the pre-condition of the let with the pre-condition needed for the let-body. The main complication in this logic is the requirement of a measure to prove termination of function and method calls. This adds complexity to these two rules, which would be present in the core logic already, had we chosen to come up with a total correctness logic to start with. For concrete examples of proving termination, finding an appropriate measure remains the main step.

The resulting termination logic has been formalised in Isabelle and been proven sound and complete. Thus it can be used as the basis for proving termination of general JVM code. However, we have not developed derived assertions for this logic, yet. We do not elaborate on the details of the termination logic here and refer the interested reader to [3].

5. From Type Systems to Program Logic

In this section we discuss how to map a high-level type system, which encodes information on resource consumption, down to a specialised program logic, which facilitates the process of independently proving given resource bounds.

5.1. The General Approach

A type system for a programming language defines a subset of all possible programs, which are in some sense "well behaved". The idea is to constrain the set of acceptable programs in such a way, that set membership can be tested efficiently by a type checking, or even better a type inference algorithm. Once a program passes type checking, the programmer can be sure that the program is "well behaved" and therefore does not need to consider bugs that might be due to ill-typed programs. This idea is often paraphrased as *well-typed programs don't go wrong*.

In general we can state this property as follows: if a program term t is well-typed, i.e. belongs to the set of well-typed programs, the compiled code $\ulcorner t \urcorner$ will adhere to the safety policy Φ defined by the type system:

$$t \in T \Longrightarrow \ulcorner t \urcorner \models \Phi$$

Examples for modern type systems that define interesting safety policies are:

- Absence of security violations [1, 28]
- Bounds on memory usage [8, 12]
- Absence of (unwanted) aliasing [13]
- Asymptotic bounds on runtime [20, 17]

The classical approach to proving correctness of such a type system is to extend the type system to configurations of a small step operational semantics, e.g. heaps and environments and then to prove that a) well-typed configurations are closed under the reduction rules of the semantics, b) no "wrong" (under the purported safety policy) configuration admits a typing.

An alternative approach consists of proving the soundness of the type system directly by induction on the typing rules. This often requires an extension of the purported safety policy, corresponding to a strengthening of the inductive hypothesis.

Usually, such developments are carried out on paper and therefore prone to glossing over some technicalities. The inference system itself will then be hardcoded into the compiler. This adds both the type system itself and its implementation in the compiler to the trusted code base (TCB) of the system, which the consumer of mobile code has to implicitly trust. This problem can be tackled by encoding the entire soundness proof of the type system in a theorem prover. However, this is considerable effort and still leaves the type system in the TCB.

A related problem is that the typing rules become increasingly complicated when expressing interesting properties. Hence, the soundness proof is far from obvious and even given the soundness proof there is a considerable danger of bugs in the implementation of the type inference process. This makes the size of the TCB an important issue in the design of a PCC system.

Another problem when using high-level type systems to realise a PCC infrastructure is that certificate checking, i.e. proving properties of the program, is done on the object code level, rather than the high level of the type system, where none of the high level program structure is present any more. Thus, typically a complete proof has to be used as the certificate for a particular piece of code. This yields very large certificates, despite various efforts in compressing proofs [4] or using techniques such as oracle strings to just record the inference path [23].

Finally, several different type systems might be developed to describe different resources, or properties, of the code. Combining these type systems would be very useful, but difficult, since they are usually not very modular. A common language as the basis for all these type systems would therefore be desirable.

In addressing these issues, we do not take the classical approach of hand-crafting a type system and then formalising the entire soundness proof to obtain security. Instead we define for each type τ and context Γ a *derived assertion* $D(\Gamma,\tau)$, which expresses the property of interest on the level of the program logic. This derived assertions must be related to the compilation $\ulcorner\cdot\urcorner$ of a program term t as follows

$$\Gamma \vdash t : \tau \Longrightarrow \ulcorner t \urcorner : D(\Gamma,\tau)$$

We then can prove once and for all the soundness of derived rules for this form of assertions. To simplify reasoning with these derived rules they should be mainly syntax-directed, with simple side conditions, that can be easily checked. Most of the difficult reasoning is done in the soundness proof already, which can also be done in an automated theorem prover to improve the trust in the system. We remark that the derived rules roughly correspond to the cases in a proof of type soundness by induction on typing rules (the second approach to correctness of a type system described above).

In our approach the program logic on bytecode level (or "bytecode logic") becomes the *assembler language for formal correctness*. Types in the high-level language are compiled into assertions for this program logic, and thus retain some information on the structure of the high-level code. With such a bytecode logic, there is no need to trust the type system itself, since any property derived via the type system is proven on the level of the logic. This follows the spirit of the foundational PCC [2] approach, that aims to minimise the trusted code base. On a more general issue, this program logic can serve as the basis for combining different type systems, which are all compiled down to this logic.

As a very simple example, assume a Camelot function that takes a list as input and produces a list as output: $List\ \alpha \longrightarrow List\ \alpha$. To encode traditional type-correctness for this language we translate the type into the following specification for $f(x)$:

$$\lambda\, E\ h\ h'\ v\ p.h \models_{list} E\langle x\rangle \Longrightarrow h' \models_{list} v$$

As before the predicate $h \models_{list} v$ asserts that v points to a linked list in heap h.

To prove this property several approaches have been taken in the literature. One could directly encode the logic into a theorem prover and perform the proof for this particular property using these rules [21]. However, for many interesting properties a proof on this level becomes very complicated even with small programs. In particular, the side conditions often talk about relationships between entire heaps, that need to be suitably instantiated. Another classical approach is to first collect all verification conditions (VCs) falling out of the rules of the program logic and then to pass it to (a maybe specialised) prover to solve these conditions. Again, depending on the complexity of the underlying program logic these VCs may be very complex, beyond the scope of what a prover can automatically solve. At the other extreme of using automated proving tools, one could formalise the entire proof of type soundness in a prover, and then instantiate it for the particular example by hand.

Our approach lies between these extremes, in choosing a specialised logic, built on top of derived assertions. Using the format mentioned above, we define rules for this particular format without ever unfolding its definition. The resulting rules are largely syntax-directed and the side-conditions are much simpler than in the general program logic. Once the soundness of these rules have been proven, most of the complexity of the proof is hidden behind this new specialised logic.

Continuing with our list example, we can define derived assertions as follows:

$$\begin{aligned} D(x : \mathit{List}\ \alpha, y : \mathit{List}\ \alpha, \mathit{List}\ \alpha) \equiv \lambda\ E\ h\ h'\ v\ p.\ & \\ & h \models_{list} E\langle x\rangle \wedge h \models_{list} E\langle y\rangle \Longrightarrow \\ & h' \models_{list} E\langle x\rangle \wedge h' \models_{list} E\langle y\rangle \wedge h' \models_{list} v \end{aligned}$$

Now, we can define rules for the image of compiling high-level constructs such as cons down to bytecode:

$$\frac{\ulcorner e_1 \urcorner : D(\Gamma, \alpha) \quad \ulcorner e_2 \urcorner : D(\Gamma, \mathit{List}\ \alpha)}{\ulcorner \mathit{cons}(e_1, e_2) \urcorner : D(\Gamma, \mathit{List}\ \alpha)}$$

Here $\ulcorner \mathit{cons}(e_1, e_2) \urcorner$ is the code snippet produced by the compiler as the result of the high-level construct $\mathit{cons}(e_1, e_2)$. Of course, altering the compiler, if only slightly, requires us to redo the proof of this rule. However, for any given program it can be applied in a syntax-directed fashion without having to do much logical reasoning.

5.2. Extended Example: The Cachera-Jensen Analysis

Cachera, Jensen, Pichardie, and Schneider [6] have developed the entire meta-theoretic correctness proof for a simple program analysis in the Coq theorem prover. We will use the same analysis as an extended example illustrating the method of derived assertions.

The analysis guarantees a bounded (independent of input) heap consumption by showing that no cycle in the control flow graph contains a memory allocation. Accordingly, memory allocations can only happen a fixed number of times.

We use a first-order fragment of Camelot [19], with lists as the only composed datatype and expressions in let-normal-form meaning arguments to functions must be variables (k are constants, x variables, f function names):

$$e \in expr ::= k \mid x \mid \texttt{nil} \mid \texttt{cons}(x_1,x_2) \mid f(x_1,\ldots,x_{n_f}) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$
$$\mid \texttt{match}\ x\ \texttt{with}\ \texttt{nil} \Rightarrow e_1 ; \texttt{cons}(x_1,x_2) \Rightarrow e_2$$

First, to be amenable to our method, the program analysis must be translated into a type system. We define such a type system as follows ($\Sigma(f)$ is a type signature mapping function names to $\mathbb{N}$):

$$\frac{\vdash_H e : n \quad n \le m}{\vdash_H e : m}\ (\text{WEAK}) \qquad \frac{}{\vdash_H k : 0}\ (\text{CONST}) \qquad \frac{}{\vdash_H x : 0}\ (\text{VAR})$$

$$\frac{}{\vdash_H f(x_1,\ldots,x_{n_f}) : \Sigma(f)}\ (\text{APP}) \qquad \frac{}{\vdash_H \texttt{nil} : 0}\ (\text{NIL}) \qquad \frac{}{\vdash_H \texttt{cons}(x_1,x_2) : 1}\ (\text{CONS})$$

$$\frac{\vdash_H e_1 : m \quad \vdash_H e_2 : n}{\vdash_H \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : m+n}\ (\text{LET}) \qquad \frac{\vdash_H e_1 : n \quad \vdash_H e_2 : n}{\vdash_H \texttt{match}\ x\ \texttt{with}\ \texttt{nil} \Rightarrow e_1 ; \texttt{cons}(x_1,x_2) \Rightarrow e_2 : n}\ (\text{MATCH})$$

To give a relationship with the Jensen-Cachera analysis let us say that a function is *recursive* if it can be found on a cycle in the call graph. Let us say, that a function *allocates* if its body contains an allocation, i.e., a subexpression of the form $\texttt{cons}(x_1,x_2)$. One can show that a program is typeable iff no recursive function allocates and that moreover in this case the type of a function bounds the number of allocations it can make. In order to establish correctness of the type system and, more importantly, to enable generation of certificates as proofs in our program logic, we will now develop a derived assertion and a set of syntax-directed proof rules for it that mimic our typing rules and permit the automatic translation of any typing derivation into a valid proof.

Recall that $\Gamma \rhd \texttt{e} : A$ is a judgement of the core logic, and that A is parameterised over variable environment, pre- and post-heap (see [3] for more details on encoding program logics for these kinds of languages).

Based on this logic, we can now define a *derived assertion*, which captures the fact that the heap h' after the execution is at most n units larger than the heap h before execution[2]: $D(n) \equiv \lambda E\ h\ h'\ v\ p.\ |dom(h')| \le |dom(h)| + n$. We can now write *derived rules* of the canonical form $\rhd e : D_e(n)$ to arrive at a program logic for heap consumption:

$$\frac{\rhd e : D(n) \quad n \le m}{\rhd e : D(m)}\ (\text{DWEAK}) \qquad \frac{}{\rhd k : D(0)}\ (\text{DCONST}) \qquad \frac{}{\rhd x : D(0)}\ (\text{DVAR})$$

$$\frac{}{\rhd f(x_1,\ldots,x_{n_f}) : \Sigma(f)}\ (\text{DAPP}) \qquad \frac{}{\rhd \texttt{nil} : D(0)}\ (\text{DNIL}) \qquad \frac{}{\rhd \texttt{cons}(x_1,x_2) : D(1)}\ (\text{DCONS})$$

[2]We do not model garbage collection here, so the size of the heap always increases.

$$\frac{\rhd e_1 : D(m) \quad \rhd e_2 : D(n)}{\rhd \mathtt{let}\ x{=}e_1\ \mathtt{in}\ e_2 : D(m+n)}\ \text{(DLET)} \qquad \frac{\rhd e_1 : D(n) \quad \rhd e_2 : D(n)}{\rhd \mathtt{match}\ x\ \mathtt{with}\ \mathtt{nil} \Rightarrow e_1 ; \mathtt{cons}(x_1,x_2) \Rightarrow e_2 : D(n)}\ \text{(DMATCH)}$$

All these proof rules for derived assertions have been formalised and proven in Isabelle.

We can now automatically construct a proof of bounded heap consumption, by replaying the type derivation for the high-level type system $\vdash_H$, and using the corresponding rules in the derived logic. The side-conditions coming out of this proof will be only the inequality side-conditions used in the derived logic. No reasoning about the heaps is necessary at all at this level; this has been covered already in the soundness proof of the derived logic w.r.t. the core program logic.

5.3. Beyond Cachera-Jensen

The destructive pattern match of Camelot allows one to replenish the freelist and thus to make an unbounded number of allocations yet get by with a bounded total heap size.

We can generalise our type system to encompass this situation by assigning a type of the form $\Sigma(f) = (m,n)$ with $m,n \in \mathbb{N}$ to functions and, correspondingly, a typing judgement of the format $\vdash_\Sigma e : (m,n)$. The corresponding derived assertion $D(m,n)$ asserting that if in the pre-heap the static field FL points to a freelist of length greater or equal to m then the static field FL points to a freelist of length greater or equal n in the post heap. Moreover, the size of the post-heap equals the size of the pre-heap.

Of course, we assume now that allocations are fed from the freelist, that is part of the heap, as long as that freelist is nonempty. If we know that, say, $e : (5,3)$ then we can execute e after filling the freelist with 5 freshly allocated cells.

The typing rules for this extended system are as follows; corresponding derived rules are provable in the program logic.

$$\frac{\vdash_H e : (m,n) \quad m' \geq m+q \quad n' \leq n+q}{\vdash_H e : (m',n')}\ \text{(WEAK)} \qquad \frac{}{\vdash_H k : (0,0)}\ \text{(CONST)} \qquad \frac{}{\vdash_H x : (0,0)}\ \text{(VAR)}$$

$$\frac{}{\vdash_H f(x_1,\ldots,x_{n_f}) : \Sigma(f)}\ \text{(APP)} \qquad \frac{}{\vdash_H \mathtt{nil} : (0,0)}\ \text{(NIL)} \qquad \frac{}{\vdash_H \mathtt{cons}(x_1,x_2) : (1,0)}\ \text{(CONS)}$$

$$\frac{\vdash_H e_1 : (m,n) \quad \vdash_H e_2 : (n,k)}{\vdash_H \mathtt{let}\ x{=}e_1\ \mathtt{in}\ e_2 : (m,k)}\ \text{(LET)} \qquad \frac{\vdash_H e_1 : (m,n) \quad \vdash_H e_2 : (m+1,n)}{\vdash_H \mathtt{match}\ x\ \mathtt{with}\ \mathtt{nil} \Rightarrow e_1 ; \mathtt{cons}(x_1,x_2)@\_ \Rightarrow e_2 : (m,n)}\ \text{(MATCH)}$$

Notice that this type system does not prevent deallocation of live cells. Doing so would compromise functional correctness of the code but not the validity of the derived assertions that merely speak about freelist size.

In [5] we extend the type system even further by allowing for input-dependent freelist size using an amortised approach. Here it is crucial to rule out "rogue programs" that deallocate live data. There are a number of type systems capable of doing precisely that; among them we choose the admittedly rather restrictive linear typing that requires single use of each variable.

To obtain working experience with our PCC infrastructure and the program logics, a small set of *exercises* has been made available online and is available at: `http://lionel.tcs.ifi.lmu.de/mrg/pcc4/exercises.html`. The software required for running the exercises can be downloaded from the following address: `http://groups.inf.ed.ac.uk/mrg/camelot/programs/MRG-infra-0805.tgz`.

References

[1] M. Abadi. Logic in Access Control. In *Proceedings of the Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS03)*, pages 228–233, 2003.

[2] A.W. Appel. Foundational proof-carrying code. In *LICS'01 — 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE Computer Society, June 2001.

[3] D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A Program Logic for Resources. *Theoretical Computer Science*, July 2005. Submitted.

[4] S. Berghofer and T. Nipkow. Proof Terms for Simply Typed Higher Order Logic. In *TPHOL'02 — Theorem Proving in Higher Order Logics*, volume 1869 of *LNCS*, pages 38–52. Springer-Verlag, 2000.

[5] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In Andrei Voronkov Franz Baader, editor, *LPAR 2004 — Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *LNCS*, pages 347–362, Montevideo, Uruguay, March 14–18, Feb 2005. Springer.

[6] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *FM'05 — Intl Symp on Formal Methods*, LNCS, pages 91–106, Newcastle, UK, July 18–22, 2005. Springer.

[7] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978. see Corrigendum in SIAM J. Comput. 10, 612.

[8] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00 — Symposium on Principles of Programming Languages*, Boston, MA, USA, Jan 19–21, 2000.

[9] G. A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, University of Toronto, 1975.

[10] C. Hankin and D. Le Métayer. A Type-based Framework for Program Analysis. In B. Le Charlier, editor, *SAS'94 — Static Analysis Symposium*, volume 864 of *LNCS*, pages 380–394, Namur, Belgium, September 1994. Springer-Verlag.

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

[12] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*, pages 185–197, New Orleans, LA, USA, January 2003. ACM Press.

[13] A. Igarashi and N. Kobayashi. Resource Usage Analysis. In *POPL'02 — Symposium on Principles of Programming Languages*, pages 331–342, January 2002. Expanded version to appear in ACM TOPLAS.

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

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

[16] T-M. Kuo and P. Mishra. Strictness Analysis: a New Perspective Based on Type Inference. In *FPCA'89 — Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, Imperial College, London, UK, September 11–13, 1989. ACM Press.

[17] D. Leivant. Implicit Computational Complexity for Higher Type Functionals. In *CSL*, volume 2471 of *LNCS*, pages 367–381. Springer, 2002.

[18] X. Leroy. Bytecode Verification for Java Smart Cards. *Software Practice and Experience*, 32(4):319–340, April 2002.

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

[20] J-Y. Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183(1):2–18, 2003.

[21] J. Strother Moore. Inductive assertions and operational semantics. In *CHARME'03 — Correct Hardware Design and Verification Methods*, volume 2860 of *LNCS*, pages 289–303, L'Aquila, Italy, Oct 21–24, 2003. Springer.

[22] G.C. Necula. Proof-carrying Code. In *POPL'97 — Symposium on Principles of Programming Languages*, pages 106–116, Paris, France, Jan 15–17, 1997.

[23] G.C. Necula. *Logical Aspects of Secure Computer Systems*, chapter Proof-Carrying Code. IOS Press, 2005.

[24] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 2005. ISBN 3-540-65410-0.

[25] T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[26] B.C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[27] C. Pierik and F.S. de Boer. Modularity and the Rule of Adaptation. In C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST*, volume 3116 of *LNCS*, pages 394–408. Springer, 2004.

[28] D. Walker. A Type System for Expressive Security Policies. In *POPL'00 — Symposium on Principles of Programming Languages*, pages 254–267, Boston, MA, USA, Jan 19–21, January 2000.