Optimisation Validation

David Aspinall[1]

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

Lennart Beringer[2]

Institut für Informatik, Ludwig-Maximilians-Universität München, Germany

Alberto Momigliano[3]

LFCS, School of Informatics, University of Edinburgh, U.K. and
DSI, University of Milan, Italy

Abstract

We introduce the idea of *optimisation validation*, which is to formally establish that an instance of an optimising transformation indeed improves with respect to some resource measure. This is related to, but in contrast with, *translation validation*, which aims to establish that a particular instance of a transformation undertaken by an optimising compiler is semantics preserving. Our main setting is a program logic for a subset of Java bytecode, which is sound and complete for a resource-annotated operational semantics. The latter employs *resource algebras* for measuring dynamic costs such as time, space and more elaborate examples. We describe examples of optimisation validation that we have formally verified in Isabelle/HOL using the logic. We also introduce a type and effect system for measuring static costs such as code size, which is proved consistent with the operational semantics.

Key words: Compiler Optimisation, Translation Validation, Program Logic, Java Virtual Machine Language, Cost Modelling, Resource Algebras, Lightweight Verification.

1 Introduction

We are interested in certifying the resource usage of mobile code for the Java platform. In previous work [3,1,6] we have described a proof-carrying code

[1] Email: `da@inf.ed.ac.uk`.

[2] Email: `beringer@tcs.ifi.lmu.de`.

[3] Email: `amomigl1@inf.ed.ac.uk`.

infrastructure which accomplishes this for memory usage. A class file is accompanied by a proof certificate which describes the resource usage of the `main` method of the program; we use a program logic with judgments of the form $\rhd e : \{P\}$, stating that expression e satisfies assertion P. For example

$$\rhd e_{\mathtt{main}} : \{r = F(h)\}$$

where h is the starting heap of the program (in particular, containing the arguments to the `main` method) and r is the memory consumption of the program expressed as a function of the size of the arguments in h.

In this paper we investigate a significant extension of this framework and a particular application. First, we generalise the form of resources so that a wider range of notions is covered, in an uniform fashion. Second, we consider *orderings* on resources, which allow us to talk about *optimisation validation*, in the sense that we can establish when one program consumes fewer resources than another.

This turns out to be of interest for the compiler community, where much research has been invested in trying to select (the order of) the best compiler transformation in the current context, given the available resources. This may be problematic:

> "However, current optimisation strategies do not always achieve the performance goals. Indeed, it is well known that optimizations may degrade performances in certain circumstances. The difficulty is that current techniques cannot always determine when it is beneficial or harmful to apply an optimization." [32]

This is where optimisation validation comes to the rescue: technically, it is inspired by the idea of *translation validation* [24], an alternative to the wholescale verification of translators and compilers. In this approach, one instead constructs a validation mechanism that, after every run of a compiler, formally confirms that the target code produced on that run is a correct translation of the source producing

> "[. . .] the same result while *(hopefully)* executing in less time or space or consuming less power." [25]

(our emphasis). Here, we take the reduction in resource usage as being the primary motivation, and therefore, what should be checked. This is appropriate in scenarios such as the safety policies considered in proof-carrying code, where resource usage may even be a more important concern than correctness, because it encompasses the security requirements of the domain.

1.1 Notions of optimisation.

To consider validating optimisations, we must first define what we mean by optimisation in our setting. We suppose that a program is given as a collection of classes, one of which includes a nominated `main` method. A simple notion of *dynamic* optimisation refers to every terminating execution of this method. Let P_1 be the program before optimisation and P_2 be the program after:

$$P_1 \longrightarrow P_2$$

We only need to consider the costs for the bodies of the main method in each program,

$$e_1 \longrightarrow e_2$$

Changes in other methods may be optimising, neutral or even non-optimising; at this point we do not study optimisations within nested program contexts. To be considered an optimisation, we want to establish that the transformation is improving with respect to a cost model. We capture the latter with the notion of *resource algebra* $\mathcal{R}$, which contains components for measuring the cost of executing each kind of instruction, along with an ordering on those costs. The overall (dynamic) cost may depend on the input of the program, and it is measured by execution in a operational semantics annotated with calculations using $\mathcal{R}$. If for all input heaps both e_1 and e_2 converge, then the resource consumption of e_2 should improve on that of e_1 [4]:

$$h \vdash e_1 \Downarrow r_1 \quad \wedge \quad h \vdash e_2 \Downarrow r_2 \implies r_2 \leq r_1$$

where the ordering $\leq$ refers to the ordering from $\mathcal{R}$. We may assume, without loss of generality, that the input pointer for the argument(s) to `main` is fixed on every execution.

1.2 Optimisation sequences.

The above defines our notion of a single-step optimisation. For several optimisations in sequence, it is enough to consider an optimisation between the initial and final program for the resource algebra of interest $\mathcal{R}$. However, we often want to decompose a sequence of optimisations into several transformations which are individually optimising. Then we can show the existence of a sequence of optimising steps:

$$P_1 \longrightarrow P_2 \longrightarrow \cdots \longrightarrow P_n$$

where each $P_i \longrightarrow P_{i+1}$ is an optimisation for some particular resource algebra $\mathcal{R}_i$. Additionally, each step in the optimisation should be non-increasing for the target cost model $\mathcal{R}$. A *proper* optimisation sequence has at least one step for which costs in $\mathcal{R}$ strictly decrease from some P_i to P_{i+1}.

[4] Cf. Sands's *Improvement Theory* [26] and his intensional notion of cost equivalence.

1.3 Validating optimisations by program logic.

To state and prove (dynamic) cost optimisations, we use a program logic that provides assertions about functions bounding the resources consumed. We must find assertions of the form:

$$ST_1 \rhd e_1 \ : \ \{F_1(h) \leq r\} \qquad\qquad ST_2 \rhd e_2 \ : \ \{r \leq F_2(h)\}$$

where the *specification tables* ST_i associate an assertion to each method and function in the program, providing the appropriate invariant. The assertions state that the resources consumed when executing P_1 are bounded from below by some function F_1 of the input heap, and that the resources consumed by P_2 are bounded from above by a function F_2. To show that P_2 is an optimisation of P_1 we must now prove that:

$$\forall h.\ F_2(h) \leq F_1(h)$$

(in particular, this holds trivially in case $F_1 = F_2$).

1.4 Static optimisations.

Static costs such as code size are commonly used as metrics for optimisation and some dynamic costs can be usefully approximated with static measurements. We cover both possibilities by introducing a notion of *static* resource algebra $\mathcal{S}$. To measure static costs, we use a type system with effects. For two function bodies e_1 and e_2 we must find a type t and effects s_1 and s_2 such that:

$$\Gamma_{\mathtt{main}} \vdash_{\Sigma_1} e_1 \ : \ t, s_1 \qquad\qquad \Gamma_{\mathtt{main}} \vdash_{\Sigma_2} e_2 \ : \ t, s_2$$

where the typing context for the body of main has the form $\mathtt{args} : \mathtt{String}[]$ and Σ_1 and Σ_2 are the resource typing signatures of programs P_1 and P_2 respectively, see Sect. 5. For P_2 to be a static optimisation of P_1 we should establish that $s_2 \leq s_1$, where the ordering $\leq$ now refers to the ordering on static costs. An ideal notion of optimisation would be w.r.t. a pair $(\mathcal{R}, \mathcal{S})$ of target dynamic and static cost models; a sequence of optimisations might alternate dynamic and static reductions as appropriate. A typical example is to use time *and* code size to validate optimisations such as loop unrolling, see Sect. 4.1. To simplify exposition here we consider the costs separately.

This paper is organized as follows. In Sect. 2 we present the dynamic semantics of our language, introduce resource algebras, and describe some typical instantiations. In Sect. 3, we present a program logic that generalizes the logic presented in [1] to arbitrary resource algebras. Sect. 4 gives example optimisation validations, including standard compiler optimisation steps, tail-call optimisation and an application specific one. Sect. 5 examines the static system, while Sect. 6 concludes with a summary and discussion of related work.

2 Resource annotated operational semantics

We use a functional form of Java bytecode called Grail [7], although the approach would work for other languages endowed with a structural operational semantics. Grail retains the object and method structure of JVML, but represents method bodies as sets of mutually tail-recursive first-order functions. The language is built from values v, arguments a, and function body expressions e (in this paper we do not mention static fields and virtual invocation, which are accounted for elsewhere [1]):

$$\begin{aligned}
v &::= () \mid l_C \mid i \mid \mathsf{null}_C \\
a &::= v \mid x \\
e &::= a \mid \mathsf{prim}\ a\ a \mid \mathsf{new}\ C \mid x.f \mid x.f := a \mid e\ ;e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
&\quad \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{call}\ g \mid C.m(\overline{a})
\end{aligned}$$

Here, C ranges over Java class names, f over field names, m over method names, x over variables (method parameters and locals) and g over function names (which correspond to instruction addresses in bytecode). Values consist of integer constants i, typed locations l_C, the unique element $()$ of type unit and the nullary reference null_C. As in JVML, the booleans *bool* are defined as $\mathsf{true} \overset{def}{=} 1$, $\mathsf{false} \overset{def}{=} 0$.

The (impure) call-by-value functional semantics of Grail coincides with an imperative interpretation of its direct translation into JVML, provided some syntactic conditions are met. In particular, actual arguments in function calls must coincide with the formal parameters of the function definitions. Sample Grail programs are shown in Fig. 1 and 2 in their Isabelle format and as output of the Camelot compiler [3] in Fig. 3.

To model consumption of computational resources, our semantics is annotated with a resource counting mechanism based on *resource algebras*.

Definition 2.1 A *resource algebra* $\mathcal{R}$ is a partially ordered monoid $(R, 0, +, \leq)$, i.e. $(R, 0, +)$ is a monoid and $(R, \leq)$ a partially ordered set, where

(i) 0 is the minimum element: $0 \leq x$;

(ii) $+$ is order preserving on both sides: $x \leq y$ entails $x + z \leq y + z$ and $z + x \leq z + y$.

Moreover, $\mathcal{R}$ has constants in R for each expression former: $\mathcal{R}^{\mathsf{int}}$, $\mathcal{R}^{\mathsf{null}}$, $\mathcal{R}^{\mathsf{var}}$, $\mathcal{R}^{\mathsf{prim}}$, $\mathcal{R}^{\mathsf{new}}_C$, $\mathcal{R}^{\mathsf{getf}}$, $\mathcal{R}^{\mathsf{putf}}$, $\mathcal{R}^{\mathsf{comp}}$, $\mathcal{R}^{\mathsf{let}}$, $\mathcal{R}^{\mathsf{if}}$, $\mathcal{R}^{\mathsf{call}}$ and a monotone operator $\mathcal{R}^{\mathsf{meth}}_{C,m,\overline{v}} : R \to R$.

Each constant denotes the cost associated to an instruction, which are then composed via the monoidal operation. The operator $\mathcal{R}^{\mathsf{meth}}_{\dots}$ calculates a cost for method calls. For some applications, we might parameterise the constants with additional pieces of syntax, for example if we are tracking read/writes of certain variables or charge differently selected function calls and/or primitive operations. For all the resource algebras considered here, composition is commutative; however, for examples where it is not, the order of the operation in the rules is important and matches the evaluation order.

A useful operation on such algebras is the *product*, which we (under)specify simply as monoidal product; given $\mathcal{R} = (R, 0, +, \leq)$ and $\mathcal{R}' = (R', 0', +', \leq')$, define $\mathcal{R} \times \mathcal{R}'$ as $(R \times R', \langle 0, 0' \rangle, \odot, \leq^*)$, where:

(i) $\langle r_1, r_1' \rangle \odot \langle r_2, r_2' \rangle \equiv \langle r_1 + r_2, r_1' +' r_2' \rangle$;

(ii) $\leq^*$ is any partial order on $R \times R'$, provided it satisfies the conditions in Def. 2.1.

This allows us to compose various resource algebras without committing to the ordering induced by the product of posets. For instance we may want to take the ordering on $\mathcal{R} \times \mathcal{R}'$ to be lexicographic or simultaneous orderings, see for example the product algebra introduced on page 17. Conversely, we can define the *projection* $\pi_i(\mathcal{R}^n)$ of a product as expected.

The operational semantics defines a judgement

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

which relates expressions e to environments E (maps from variables to values), initial and final heaps h, h', result values v and costs $r \in R$. Heaps are partial maps from locations l to objects, where an object is represented as a class name C together with a field table (a map from field names f to values v). We use the following notations for heaps:

$h(l)$	class name of object at l;
$h(l).f$	field lookup of value at f in object at l;
$h[l.f \mapsto v]$	field update of f with v at l.

Argument evaluation in an environment E is defined by $eval_E(x) = E(x)$ and $eval_E(v) = v$, while costs are defined by

- $cost() = cost(l) = 0$;
- $cost(\mathsf{null}) = \mathcal{R}^{\mathsf{null}}$;
- $cost(i) = \mathcal{R}^{\mathsf{int}}$;
- $cost(x) = \mathcal{R}^{\mathsf{var}}$.

The function $\mathsf{fields}(C)$ returns the sequence $\overline{f}$ of fields in the class C, while $inital_{f_i}$ denotes the initial value of the field f_i. For functions and methods, we write $body_g$ and $body_{C,m}$ to stand for the definition of g and $C.m$, respectively.

The complete listing of the operational semantics rules follows:

$$\frac{a \neq l_C}{E \vdash h, a \Downarrow h, \mathit{eval}_E(a), \mathit{cost}(a)}$$

$$\frac{E \vdash h, a \Downarrow h, v_a, r_a \qquad E \vdash h, a' \Downarrow h, v'_a, r'_a}{E \vdash h, \mathsf{prim}\ a\ a' \Downarrow h, \mathsf{prim}(v_a, v'_a), r_a + r'_a + \mathcal{R}^{\mathsf{prim}}}$$

$$\frac{l = \mathsf{freshloc}(h) \qquad \mathsf{fields}(C) = \overline{f}}{E \vdash h, \mathsf{new}\ C \Downarrow h[l.f_i \mapsto \mathit{initval}_{f_i}], l, \mathcal{R}^{\mathsf{new}}_C}$$

$$\frac{E \vdash h, x \Downarrow l, h, r_x}{E \vdash h, x.f \Downarrow h, h(l).f, r_x + \mathcal{R}^{\mathsf{getf}}}$$

$$\frac{E \vdash h, x \Downarrow l, h, r_x \qquad E \vdash h, a \Downarrow v, h, r_a}{E \vdash h, x.f := a \Downarrow h[l.f \mapsto v], (), r_x + r_a + \mathcal{R}^{\mathsf{putf}}}$$

$$\frac{E \vdash h, e_1 \Downarrow h_1, (), r_1 \qquad E \vdash h_1, e_2 \Downarrow h_2, v, r_2}{E \vdash h, e_1\ ;\ e_2 \Downarrow h_2, v, r_1 + \mathcal{R}^{\mathsf{comp}} + r_2}$$

$$\frac{E \vdash h, e_1 \Downarrow h_1, v_1, r_1 \qquad E\langle x := v_1\rangle \vdash h_1, e_2 \Downarrow h_2, v, r_2}{E \vdash h, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow h_2, v, r_1 + \mathcal{R}^{\mathsf{let}} + r_2}$$

$$\frac{E \vdash h, e \Downarrow h', \mathtt{1}, r_e \qquad E \vdash h', e_1 \Downarrow h'', v, r}{E \vdash h, \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow h'', v, r_e + \mathcal{R}^{\mathsf{if}} + r}$$

$$\frac{E \vdash h, e \Downarrow h', \mathtt{0}, r_e \qquad E \vdash h', e_2 \Downarrow h'', v, r}{E \vdash h, \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow h'', v, r_e + \mathcal{R}^{\mathsf{if}} + r}$$

$$\frac{E \vdash h, \mathit{body}_g \Downarrow h', v, r}{E \vdash h, \mathsf{call}\ g \Downarrow h', v, \mathcal{R}^{\mathsf{call}} + r}$$

$$\frac{\mathit{eval}_E(a_i) = v_i \qquad \{x_i := v_i\} \vdash h, \mathit{body}_{C,m} \Downarrow h', v, r}{E \vdash h, C.m(\overline{a}) \Downarrow h', v, (\Sigma_i\ \mathit{cost}(a_i)) + \mathcal{R}^{\mathsf{meth}}_{C,m,\overline{v}}(r)}$$

	Time	Heap	Frames	$\mathsf{MethCnts}$	$\mathsf{MethFreq}^{Id}$	$\mathsf{MethGuard}$
$\lvert\mathcal{R}\rvert$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{MS}(Id)$	$\mathcal{N} \times \mathcal{N}$	$\{\mathsf{tt}, \mathsf{ff}\}$
$\mathcal{R}^{\mathsf{int}}$	1	0	0	$\emptyset$	$(1, 0)$	tt
$\mathcal{R}^{\mathsf{null}}$	1	0	0	$\emptyset$	$(1, 0)$	tt
$\mathcal{R}^{\mathsf{var}}$	1	0	0	$\emptyset$	$(1, 0)$	tt
$\mathcal{R}^{\mathsf{prim}}$	1	0	0	$\emptyset$	$(1, 0)$	tt
$\mathcal{R}^{\mathsf{new}}_{C}$	3	$size(C)$	0	$\emptyset$	$(3, 0)$	tt
$\mathcal{R}^{\mathsf{getf}}$	2	0	0	$\emptyset$	$(2, 0)$	tt
$\mathcal{R}^{\mathsf{putf}}$	3	0	0	$\emptyset$	$(3, 0)$	tt
$\mathcal{R}^{\mathsf{comp}}$	0	0	0	$\emptyset$	$(0, 0)$	tt
$\mathcal{R}^{\mathsf{let}}$	1	0	0	$\emptyset$	$(1, 0)$	tt
$\mathcal{R}^{\mathsf{if}}$	0	0	0	$\emptyset$	$(0, 0)$	tt
$\mathcal{R}^{\mathsf{call}}$	1	0	0	$\emptyset$	$(1, 0)$	tt
$\mathcal{R}^{\mathsf{meth}}_{C,m,\overline{v}}(r)$	$\lvert\overline{v}\rvert + 2 + r$	r	$r + 1$	$r \cup_{+} \{C.m\}$	$\mathsf{Freq}_{C.m,\lvert\overline{v}\rvert}(r)$	$G_{C,m}(\overline{v}) \wedge r$
$0_{\mathcal{R}}$	0	0	0	$\emptyset$	$(0, 0)$	tt
$+_{\mathcal{R}}$	$+$	$+$	max	$\cup_{+}$	$+_{\mathsf{Freq}}$	$\wedge$
$\leq_{\mathcal{R}}$	$\leq$	$\leq$	$\leq$	$\subseteq_{+}$	$\leq_{\mathsf{Freq}}$	$\leq_{\mathsf{Guard}}$

The notation $\lvert\overline{v}\rvert$ denotes the length of the list $v_1 \dots v_n$. For method counts, $\cup_{+}$ and $\subseteq_{+}$ are multiset union and subset respectively. For frequencies, we define $\mathsf{Freq}_{Id,n}(t, p) = (0, max(t, p))$ and $\mathsf{Freq}_{C.m,n}(t, p) = (n + 2 + t, p)$ for $C.m \neq Id$. Composition in this case is $(t, p) +_{\mathsf{Freq}} (t', p') = (t + t', max(p, p'))$ and the ordering $(t, p) \leq_{\mathsf{Freq}} (t', p')$ iff $p \leq p'$. For guards, $G_{C,m}(\overline{v})$ is a boolean valued function for each C, m and $b \leq_{\mathsf{Guard}} b'$ iff $b = \mathsf{tt}$ or $b = b' = \mathsf{ff}$.

Table 1
Example resource algebras

2.1 Resource algebra examples

Some example resource algebras are shown in Table 1. The Time algebra models an instruction counter that approximates execution time; each Grail expression form is charged according to the number of JVM instructions to which it expands[5]. The Heap algebra counts the size of heap space consumed during execution (ignoring the possibility of garbage collection, which cannot be assumed for an arbitrary JVM). Only the *new* instruction consumes heap. The Frames algebra counts the maximal number of frames on the stack during execution. The MethCnts algebra traces invocations by accumulating a multiset of invoked method names.

The $\mathsf{MethFreq}^{Id}$ algebra calculates a measure of the frequency of calls to the method Id (a long identifier $C.m$), by accumulating the maximal period between successive calls; this is an example of an application specific algebra

[5] There are nil costs for the *if* instructions because they are compiled as test and branches; similarly, sequential composition has nil cost in these example algebras.

(see Sect. 4.3 for a motivating example).

Finally, the MethGuard algebra does not calculate a quantitative resource, but rather maintains a boolean monitor that checks whether arbitrary guards $G_{C,m}(\overline{v})$ are satisfied at invocations of method m in class C. If guards are considered as resource usability preconditions (for example, to check that a method parameter lies within some limits), then we may consider an optimisation to be a transformation that ensures the resource preconditions are always satisfied.

In this last case, the resource operator $\mathcal{R}^{\mathsf{meth}}_{C,m,\overline{v}}$ depends on the run-time values v_i, whereas in the other examples the function is fixed because the length of the argument list is specified by the definition of the method. In general, resource algebras that depend on runtime values such as this can collect traces along the path of computation; the resulting word may be constrained by further policies, specified for example by security automata [28] or by formulae from logics over linear structures, which can be encoded in the higher-order assertion language of our program logic, introduced next.

3 Resource-aware program logic

Our primary basis for optimisation validation is a general-purpose program logic for Grail where assertions are boolean functions over all semantic components occurring in the operational semantics, namely the input environment E and initial heap h, the post heap h', the result value v, and the resources consumed r. An assertion P thus belongs to the type $\mathcal{E} \to \mathcal{H} \to \mathcal{H} \to \mathcal{V} \to \mathcal{R} \to \mathcal{B}$.

A judgement $G \rhd e : P$ in the logic relates a Grail expression e to an assertion P, dependent on a context $G = \{(e_1, P_1), \ldots, \{e_n, P_n)\}$ that stores assumptions for recursive program structures, in the spirit of Hoare's original proof rule for procedures [17]. The program logic comprises one rule for each expression form, an axiom and a consequence rule, where the square-bracket notation $P[E, h, h', v, r]$ indicates the instantiation of a predicate:

$$\frac{(e, P) \in G}{G \rhd e : P} \qquad\qquad \frac{G \rhd e : P \quad P \longrightarrow Q}{G \rhd e : Q}$$

$$\frac{}{G \rhd a : \{h' = h \ \wedge\ v = \mathit{eval}_E(a) \ \wedge\ r = \mathit{cost}(a)\}}$$

$$\frac{}{\begin{aligned} G \rhd \mathsf{prim}\ a_1\ a_2 : \{h' = h \ \wedge\ & v = \mathsf{prim}(\mathit{eval}_E(a_1), \mathit{eval}_E(a_2)) \ \wedge \\ & r = \mathit{cost}(a_1) + \mathit{cost}(a_2) + \mathcal{R}^{\mathsf{prim}}\} \end{aligned}}$$

$$\frac{}{G \rhd \mathsf{new}\ C : \{v = \mathsf{freshloc}(h) \ \wedge\ h' = h[v.f_i \mapsto \mathit{initval}_{f_i}] \ \wedge\ r = \mathcal{R}^{\mathsf{new}}_C\}}$$

$$\frac{}{G \rhd x.f : \{h = h' \wedge (\exists l. E(x) = l \wedge v = h(l).f) \wedge r = \mathit{cost}(x) + \mathcal{R}^{\mathsf{getf}}\}}$$

$$\frac{}{\begin{array}{c} G \rhd x.f := a \;:\; \{(\exists l.\, E(x) = l \wedge h' = h[l.f \mapsto \mathit{eval}_E(a)]) \wedge v = () \wedge \\ r = \mathit{cost}(x) + \mathit{cost}(a) + \mathcal{R}^{\mathsf{putf}}\} \end{array}}$$

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

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

$$\frac{G \rhd e_1 : P_1 \qquad G \rhd e_2 : P_2 \qquad G \rhd e_3 : P_3}{\begin{array}{l} G \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \{\exists\, h_1\, v_1\, r_1\, r_2.\; P_1[E, h, h_1, v_1, r_1] \wedge \\ \quad (v_1 = \mathsf{1} \Longrightarrow P_2[E, h_1, h', v, r_2]) \wedge \\ \quad (v_1 = \mathsf{0} \Longrightarrow P_3[E, h_1, h', v, r_2]) \;\wedge\; r = r_1 + \mathcal{R}^{\mathsf{if}} + r_2\} \end{array}}$$

$$\frac{G \cup \{(\mathsf{call}\ g, P)\} \rhd \mathit{body}_g : P[E, h, h', v, \mathcal{R}^{\mathsf{call}} + r]}{G \rhd \mathsf{call}\ g : P[E, h, h', v, r]}$$

$$\frac{G \cup \{(C.m(\overline{a}), P)\} \rhd \mathit{body}_{C,m} : P[\{x_i := \mathit{eval}_E(a_i)\}, h, h', v, \mathcal{R}^{\mathsf{meth}}_{C,m,\mathit{eval}_E(\overline{a})}(r)]}{G \rhd C.m(\overline{a}) : P[E, h, h', v, r]}$$

Before demonstrating how the program logic is used to verify the resource consumption of programs, we summarise some basic meta-theoretical properties. These have been formally proven by representing the operational semantics and the program logic in the proof assistant Isabelle/HOL; for more details, see [1,2]; the results here are a generalisation with resource algebras of those presented there. First, semantic validity, which has a partial correctness interpretation:

Definition 3.1 An assertion P is *valid* for expression e, written $\models e : P$, if for all E, h, h', v and r $E \vdash h, e \Downarrow h', v, r$ implies that the assertion $(P\ E\ h\ h'\ v\ r)$ holds. A context G is *valid*, $\models G$, if for all pairs (e, P) in G, it holds that $\models e : P$. Assertion P is *valid for e in context* G, if $\models G$ implies $\models e : P$.

Indeed, the proof system is sound with respect to the operational semantics:

Theorem 3.2 (Soundness) *If $G \rhd e : P$ then $G \models e : P$.*

The proof of Theorem 3.2 proceeds by induction on the height of derivations, employing suitably relativised notions of (context) validity.

Given the adopted partial correctness interpretation, it is clear that non-terminating programs satisfy their specifications vacuously. To verify resource consumption of such programs, we have developed an auxiliary termination logic, see [2].

The treatment of logical completeness, as well as the actual proving methodology, benefits from some admissible rules concerning the proof context. Beyond the usual weakening rule(s), other rules allow one to discharge the proof context, i.e. to derive judgements in the absence of contextual assumptions. This is done using a *specification table* ST, which maps function and method calls into assertions. We says that a context G *respects* the specification table ST, notation $ST \models G$, if all entries in it consist of a function or method' call together with its assertion in the table; moreover their bodies satisfy a corresponding assertion. See [2] for the formal definition.

$$\frac{ST \models G \quad (e, P) \in G}{\emptyset \rhd e : P} \quad (\textsc{spectable})$$

$$\frac{ST \models G \qquad (C.m(\overline{a}), ST(C, m, \overline{a})) \in G}{\emptyset \rhd C.m(\overline{b}) : ST(C, m, \overline{b})} \quad (\textsc{adapt})$$

The spectable rule accounts for the verification of (possibly mutually recursive) program fragments using the specification table, while adapt may be used to adjust the actual arguments when extracting method specifications.

To prove relative completeness, we define a context G_{strong} that associates to each function and method call its strongest specification.

Definition 3.3 The strongest specification for e is

$$SSpec(e) \equiv \{E \vdash h, e \Downarrow h', v, r\}$$

Lemma 3.4 *For any e, $G_{strong} \rhd e : SSpec(e)$.*

Furthermore, G_{strong} satisfies $ST_{strong} \models G_{strong}$, where ST_{strong} is the specification table defined by $(\lambda g.\ SSpec(\mathsf{call}\ g), \lambda C m \overline{a}.\ SSpec(C.m(\overline{a})))$. From this, we obtain:

Theorem 3.5 (Completeness) *For any e and P, $\models e : P$ implies $\emptyset \rhd e : P$.*

The completeness result means that we can conceivably derive any provable assertion using the rules of the program logic, following the structure of the program.

4 Validated optimisations

The program logic presented in the previous section can be used to justify program transformations that are routinely applied in optimising compilers [20], *provided* they are in fact improving. In this section we give some example optimisations and sketch the proofs of their validation. While the transformations and the examples we consider in this paper are fairly simple, they serve the purpose of demonstrating our methodology.

4.1 Standard low-level optimisations

We first consider the motivating program of [25],

```
i <- 0; x <- 1; y <- 2;
WHILE i < 24 DO {i <- i + x + y ; g <- 2 * i}
EXIT
```

Our formal verification refers to a translation of this code into (the Isabelle representation of) our language. The result of this (manual) translation is the method $\mathtt{R.calc}_0$:

```
method static int R.calc$_0$() =
   let $i = 0$ in let $x = 1$ in let $y = 2$ in let $g = 0$ in call $f$
   fun f(int i, int x, int y, int g) = if $i < 24$ then call $h$ else var $g$
   fun h(int i, int x, int y, int g) =
      let $j = i + x$ in let $i = j + y$ in let $g = 2 * i$ in call $f$
```

which differs from the original code only in minor ways: we extended the loop prelude by an assignment to variable g, converted the loop into two functions which represent basic blocks, and turned the EXIT statement into a return statement of the final value of g.

Using the resource algebra Time defined previously, we now outline our Isabelle proof of the judgement

$$\emptyset \rhd \mathtt{R.calc}_0([\,]) : \{r = 213\} \tag{1}$$

which states that an invocation of $\texttt{R.calc}_0$ requires 213 units of time. We first define two auxiliary (semantic) functions

$$\begin{aligned} \mathit{cost}_f(n) &= 24 * n + 10 \\ \mathit{cost}_h(n) &= 24 * n + 1 \end{aligned}$$

that describe the costs of evaluating functions f and h, respectively, where n is the number of loop iterations. Next, we define a specification table ST_0 for $\texttt{calc}_0$ and its local functions f and h.

$$\left[\begin{array}{l} \texttt{R.calc}_0 \mapsto \{r = 11 + \mathit{cost}_f(8)\} \\ \quad \textsf{call}\ f \mapsto \{\forall\ J.\ (E(x) = 1 \wedge E(y) = 2 \wedge E(i) = 3 * J \wedge J \leq 8) \longrightarrow \\ \qquad\qquad\qquad r = \mathit{cost}_f(8 - J)\} \\ \quad \textsf{call}\ h \mapsto \{\forall\ J.\ (E(x) = 1 \wedge E(y) = 2 \wedge E(i) = 3 * J \wedge J \leq 7) \longrightarrow \\ \qquad\qquad\qquad r = \mathit{cost}_h(8 - J)\} \end{array}\right]$$

The first line defines the specification of $\texttt{R.calc}_0$ in terms of the auxiliary function cost_f, while the entries for $\textsf{call}\ f$ and $\textsf{call}\ h$ ensure that the auxiliary functions correctly model the costs of the executing the local functions. In both cases, the specifications depend on the value of the variable i; intuitively, the universally quantified variable J represents the number of loop iterations that have already been performed.

Next, we define a context, G_0 that associates the specification table entries to the relevant function and method calls, e.g.

$$G_0 = \{(\texttt{R.calc}_0([\,]),\ ST_0\ \texttt{R.calc}_0), (\textsf{call}\ f, ST_0\ \textsf{call}\ f), (\textsf{call}\ h, ST_0\ \textsf{call}\ h)\}$$

The core of the verification consists in establishing $ST_0 \models G_0$. From that, it is just a matter of calling the SPECTABLE rule to conclude the proof of (1). The former, in turn, requires us to prove that each entry in G_0 is justified: for each entry $(\textsf{call}\ f, P)$ – and similarly for method entries – we need to show that the body body_f satisfies $G_0 \rhd \mathit{body}_f : P[E, h, h', v, \mathcal{R}^{\textsf{call}} + r]$. Using the rules of our program logic, these proofs proceed syntax-directed and in a VC-Gen fashion, leaving side conditions involving numeric constraints. Ideally, we would delegate the solution of those verification conditions to a fully automated (external) solvers. Currently, instead, the proof assistant often needs directions when facing large case-splits and quantifier instantiations beyond decision procedures.

In the same fashion, we have established Isabelle proofs of specifications $\emptyset \rhd \texttt{R.calc}_i([\,]) : \{r = r_i\}$ for methods $\texttt{R.calc}_1 \ldots \texttt{R.calc}_7$ which arise from applying the code transformations described in [25] to $\texttt{R.calc}_0$. The resulting code is shown in Fig. 1, while Table 2 summarises the costs r_i obtained for each transformation step.

```
class R {
...
method static int calc1()  = let i = 0 in let x = 1 in let y = 2 in let g = 0 in call f
  fun f(int i, int x, int y, int g)  = if i < 24 then call h else var g
  fun h(int i, int x, int y, int g)  = let i = i + 3 in let g = 2 * i in call f

method static int calc2()  = let i = 0 in let g = 0 in call f
  fun f(int i, int g)  = if i < 24 then call h else var g
  fun h(int i, int g)  = let i = i + 3 in let g = 2 * i in call f

method static int calc3()  = let i = 0 in let g = 0 in call h
  fun h(int i, int g)  = let i = i + 3 in let g = 2 * i in if i < 24 then call h else var g

method static int calc4()  = let g = 0 in call h
  fun h(int g)  = let g = g + 6 in if g < 48 then call h else var g

method static int calc5()  = let g = 0 in call h
  fun f(int g)  = let g = g + 6 in if g < 48 then call h else var g
  fun h(int g)  = let g = g + 6 in if g < 48 then call f else var g

method static int calc6()  = let g = 0 in call h
  fun h(int g)  = let g = g + 6 in let g = g + 6 in if g < 48 then call h else var g

method static int calc7()  = let g = 0 in call h
  fun h(int g)  = let g = g + 12 in if g < 48 then call h else var g}
```

Figure 1. A sequence of low level transformations

In general, proofs of functional correctness of arbitrary code fragments may be required to verify statements about resource consumption in this case-study. However, this is not the case for the specific transformations we considered: none of the specifications involved constrains the result values v. Furthermore, none of the transformations increases the dynamic resources consumed. Indeed, except for loop unrolling (the conversion calc$_4$ → calc$_5$), all transformations reduce the costs [6].

[6] The loop unrolling performed in [25] actually *increases* the dynamic costs, because it jumps to a shared code block instead of duplicating the continuation code. Using our

i	t_i	Transformation
0	213	
1	197	Constant propagation and constant folding
2	193	Dead assignment elimination
3	176	Branch movement, inlining, redundant test elimination
4	126	Induction variable elimination
5	126	Loop unrolling *without* code sharing
6	82	Dead code elimination
7	66	Expression folding

Table 2
Costs associated to low level transformations

4.2 Tail-call optimisation

Next, we consider a recursive program involving heap structures. Figure 2 defines the class REV with method $\mathtt{App}$ for appending an element to a list, and methods $\mathtt{Rev}_1, \ldots, \mathtt{Rev}_3$ for reversing a list. We assume that objects of class LIST contain fields HD and TL of type $\mathtt{int}$ and LIST, respectively.

Concentrating our attention on the required height of the frame stack, we observe that method $\mathtt{Rev}_1$ is formulated using method recursion and employs the auxiliary method $\mathtt{App}$. As all its recursive invocations are nested, $\mathtt{Rev}_1$ requires a frame stack of a height that depends linearly on the length of the input list. To express this dependency we define a predicate $h, v \models_X n$ that specifies when a reference value v represents a non-cyclic (integer) list of length n in a heap region $h\lfloor_X$.

$$\begin{aligned} h, v \models_\emptyset 0 \;&\equiv\; v = \mathsf{null} \\ h, v \models_{v \uplus Y} (n+1) \;&\equiv\; v \in \mathit{dom}(h) \wedge h(v) = \mathsf{LIST} \wedge h(v).\mathsf{TL} = t \wedge h, t \models_Y n \end{aligned}$$

We can now prove a specification that relates the length of the list to the stack depth. The quantitative specification we will prove also indicates that the runtime, the number of jumps, and the number of method invocations all grow quadratically with the length of the input list:

$$\rhd\, \mathsf{REV}.\mathtt{Rev1}([a]) : \{\forall\, n\; X.\; h, E(a) \models_X n \longrightarrow r_{\mathsf{Frames}} = n + 1\}$$

Method $\mathtt{Rev}_2$ arises from $\mathtt{Rev}_1$ by introducing an accumulator that eliminates

formalism of static resources we could characterise this as a static optimisation instead, namely reducing code size.

```
class REV {
method static LIST App(LIST l, int i) = call app
   fun app(LIST l, int i) = if l = null then call app_0 else call app_1
   fun app_0(int i) = let l = null in let x = new LIST in
                          x.HD:=i ; x.TL:=l ; var x
   fun app_1(LIST l, int i) = let h = l.HD in let t = l.TL in
                                  let t = REV.App(t, i) in l.TL:=t ; var l

method static LIST Rev_1(LIST l) =  call rev1
   fun rev1(LIST l) = if l = null then null else call rev1_1
   fun rev1_1(LIST l) = let h = l.HD in let t = l.TL in
                            let t = REV.Rev_1(t) in REV.App(t, h)

method static LIST Rev_2(LIST l, LIST acc) =  call rev2
   fun rev2(LIST l, LIST acc) = if l = null then var acc else call rev2_1
   fun rev2_1(LIST l, LIST acc) = let h = l.HD in let t = l.TL in
                                      l.TL:=acc ; REV.Rev_2(t, l)

method static LIST Rev_3(LIST l, LIST acc) =  call rev3
   fun rev3(LIST l, LIST acc) = if l = null then var acc else call rev3_1
   fun rev3_1(LIST l, LIST acc) = let t = l.TL in l.TL:=acc ;
                                      let acc = var l in let l = var t in call rev3}
```

Figure 2. Class REV

the invocation to App and formulates the recursion as tail recursion. Its specification imposes some well-structuredness conditions on both arguments: pointers must represent lists, which moreover should be *non-overlapping* in the heap. The frame depth depends only on the length of the first argument, which gives the same overall depth cost as $\texttt{Rev}_1$ (but considerable saving in allocated space):

$$\rhd \mathsf{REV}.\texttt{Rev}_2([a,b]) : \{\forall\ n\ X\ m\ Y.\ h, E(a) \models_X n \wedge h, E(b) \models_Y m \wedge X \cap Y = \emptyset \\ \longrightarrow r_{\mathsf{Frames}} = n + 1\}$$

In Rev_3, the method-level tail recursion is converted into a method-internal loop and the redundant field is eliminated, resulting in a program whose exe-

$$\left[\begin{array}{rl}
\mathsf{REV}.\mathtt{App}(\overline{a}) \mapsto & \{\forall\ x\ y\ n\ X. \\
& \quad (\overline{a} = [x, y] \wedge h, E(x) \models_X n) \longrightarrow \\
& \qquad (h', v \models_{\{\mathsf{freshloc}(h)\} \cup X} n + 1 \wedge h =_{dom(h) \setminus X} h' \wedge r = r_{\mathtt{App}}(n))\} \\
\mathsf{REV}.\mathtt{Rev}_1(\overline{a}) \mapsto & \{\forall\ x\ n\ X. \\
& \quad (\overline{a} = [x] \wedge h, E(x) \models_X n) \longrightarrow (\exists\ Y.\ h', v \models_Y n \wedge r = r_{\mathtt{Rev1}}(n))\} \\
\mathsf{REV}.\mathtt{Rev}_2(\overline{a}) \mapsto & \{\forall\ x\ n\ X\ y\ m\ Y. \\
& \quad (\overline{a} = [x, y] \wedge h, E(x) \models_X n \wedge h, E(y) \models_Y m \wedge X \cap Y = \emptyset) \longrightarrow \\
& \qquad (\exists Z.\ h', v \models_Z n + m \wedge r = r_{\mathtt{Rev2}}(n))\} \\
\mathsf{REV}.\mathtt{Rev}_3(\overline{a}) \mapsto & \{\forall\ x\ n\ X\ y\ m\ Y. \\
& \quad (\overline{a} = [x, y] \wedge h, E(x) \models_X n \wedge h, E(y) \models_Y m \wedge X \cap Y = \emptyset) \longrightarrow \\
& \qquad (\exists Z. h', r \models_Z n + m \wedge r = r_{\mathtt{Rev4}}(n))\} \\
\mathsf{call}\ \mathtt{rev3} \mapsto & \{\forall\ n\ X\ m\ Y. \\
& \quad (h, E(\mathtt{l}) \models_X n \wedge h, E(\mathsf{acc}) \models_Y m \wedge X \cap Y = \emptyset) \longrightarrow \\
& \qquad (\exists Z. h', v \models_Z n + m \wedge r = r_{\mathtt{Rev4}}(n))\}
\end{array}\right]$$

Table 3
Specification table for class REV.

cution only requires a single frame.

$$\triangleright \mathsf{REV}.\mathtt{Rev}_3([a, b]) : \{\forall\ n\ X\ m\ Y.\ h, E(a) \models_X n \wedge h, E(b) \models_Y m \wedge X \cap Y = \emptyset \longrightarrow r_{\mathsf{Frames}} = 1\}$$

The verification of the three specifications follows the same strategy as before. This is an overall optimisation for the resource algebra Frames, but we verified the intermediate steps using a more informative *product* resource algebra, Frames × MethCntsAll × Time × Heap , where MethCntsAll is similar to the MethCnts algebra shown in Table 1, except that only the *size* of the multisets is considered (thus we sum calls to all methods), and we stipulate that size(LIST) $= 1$. We obtain the following resource tuples, which show the costs in terms of of the length n of the (first) input list.

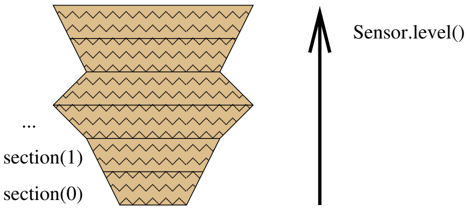
	r_{Frames}	$r_{\mathsf{MethCntsAll}}$	r_{Times}	r_{Heap}
$r_{\mathtt{App}}(n) \equiv$	$(n + 1,$	$n + 1,$	$22n + 22,$	$1)$
$r_{\mathtt{Rev1}}(n) \equiv$	$(n + 1,$	$n(n + 1)/2,$	$11n^2 + 29n + 11,$	$n)$
$r_{\mathtt{Rev2}}(n) \equiv$	$(n + 1,$	$n + 1,$	$20n + 11,$	$0)$
$r_{\mathtt{Rev3}}(n) \equiv$	$(1,$	$1,$	$18n + 11,$	$0)$

Under the lexicographic order for the product algebra, both steps are optimising. To preserve datatype representation conditions across method calls, we need stronger invariants in the specifications than shown above. The full specification table is given in Table 3, which also contains an invariant of the loop represented by the function call `rev3`.

4.3 Optimisation of method call frequency

As well as standard optimisations, our framework can be used to validate optimisations that are custom specified for a particular application.

Consider the hypothetical scenario of a process-control application where there is an irregularly shaped chemical tank (illustrated opposite) whose contents must be carefully monitored to ensure sufficient reagent. When the amount reaches a critical low point, the reaction must be temporarily halted while the tank refills.



An embedded controller runs a program that which monitors the level gauge. A suitable notion of optimisation in this setting would be to transform the program into one which checks the tank level more frequently, so reducing the latency between noticing a tank empty condition and triggering the refill cycle. Thus the frequency of invoking the `Sensor.level()` method is a suitable resource measure. Using the same methodology as previously, and with the resource algebra MethFreq, we can validate the transformation of a naive implementation of a program which calculates the amount of reagent into the tank into a better one which checks the level more frequently.

The full listing of the example program (naive version) is in Fig. 3. The method `calc(n)` calculates the amount of reagent left in `n` sections of the tank. It is invoked from the `runloop` method, which is supposed to be the process control loop; in reality, this loop would be run indefinitely and involve other tasks besides level calculation.

Numerous optimisations are possible in this example to increase the rate of testing the sensor level. For example, we might sum up the section sizes only until we find out that the critical level has been safely exceeded. Or (supposing the dimensions of the tank are fixed during the run of the process), we may calculate the sums for the container sections in advance to avoid looping over the `section` array each time we test the sensor level. We have not yet undertaken the formal verification of this example, as it goes slightly beyond our formal presentation of the logic as it makes use of arrays; however, the extension is straightforward.

```
class ChemCalc {
   field static int alarm
   field static int[] section
   field static int critical_amount
   method static void runloop() =
   let
     val n = 1000
     fun raise_alarm () = putstatic <int ChemCalc.alarm> 1
     fun loop_check(int n) = if n>0 then loop(n) else ()
     fun loop(int n) =
     let
       val chem_level = invokestatic <int Sensor.level()> ()
       val chem_amount = invokestatic <int ChemCalc.calc(int)> (chem_level)
       val n = sub n 1
       val critical = getstatic <int ChemCalc.critical_amount>
     in
       if chem_amount < critical then raise_alarm() else loop_check(n)
     end
   in loop_check(n) end
   method static int calc(int n) =
   let
     val a = 0
     fun sumup(int n, int a) =
     let
         val cs = getstatic <int[] ChemCalc.section>
         val x = get cs n
         val a = add x a
         val n = sub n 1
      in
         sumup_check (n,a)
      end
      fun sumup_check(int n, int a) =
          if n>0 then sumup(n,a)
                 else a
   in sumup_check(n,a) end
```

Figure 3. Grail code for an embedded controller

5 Static semantics

A static resource algebra $\mathcal{S}$ is defined exactly as in Def. 2.1, except that the resource constructors depend on the typing context only; in particular, the method operator does not depend on the values of its arguments. For a fixed signature the judgment $\Gamma \vdash e : t, s$ assigns type t and effect s to Grail expression e in a straightforward way; an example is the rule for *if* expressions:

$$\frac{\Gamma \vdash e \ : \ \text{bool},\ s_e \qquad \Gamma \vdash e_1 \ : \ t,\ s_1 \qquad \Gamma \vdash e_2 \ : \ t,\ s_2}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \ : \ t,\ s_e + \mathcal{S}^{\mathsf{if}} + s_1 + s_2}$$

Notice that this is different from usual type and effect systems, where the effect on both branches would be the same. See Fig. 4 for the full listing, where argument typing and static cost are defined as follows:

$$\begin{array}{llll}
\mathit{type}_\Gamma(x) &= \Gamma(x) & \mathit{scost}(x) &= \mathcal{S}^{\mathsf{var}} \\
\mathit{type}_\Gamma(i) &= \mathrm{int} & \mathit{scost}(i) &= \mathcal{S}^{\mathsf{int}} \\
\mathit{type}_\Gamma() &= \mathrm{unit} & \mathit{scost}() &= 0 \\
\mathit{type}_\Gamma(l_C) &= C & \mathit{scost}(l_C) &= 0 \\
\mathit{type}_\Gamma(\mathsf{null}_C) &= C & \mathit{scost}(\mathsf{null}_C) &= \mathcal{S}^{\mathsf{null}}
\end{array}$$

Many of the standard properties of type and effect systems hold, culminating in subject reduction. All proofs are standard and hence omitted. First, it is immediate to show that our system is conservative w.r.t. the effect-free system defined, as usual, by erasure. Further, a canonical forms property holds.

Fact 5.1 (Canonical forms) *Assume that* $\Gamma \vdash v \,:\, t,\ \ s$*; then:*

- *if* $t = \mathrm{int}$ *then* $v = i$ *and* $s = \mathcal{S}^{\mathsf{int}}$.
- *if* $t = \mathrm{unit}$ *then* $v = ()$ *and* $s = 0$.
- *if* $t = C$ *then either* $v = \mathsf{null}_C$ *and* $s = \mathcal{S}^{\mathsf{null}}$ *or* $v = l_C$ *and* $s = 0$.

From this we observe that every value has *constant* effect.

Weakening is admissible as well as a specialised form of substitution, in which arguments play the role of variables and the effect is increased accordingly. This generalises *value* substitution in type and effects systems, which holds because values are *pure* (i.e. have nil effect).

We now introduce a generalisation of the well-known relation between static effects and dynamic traces [31], which is key in the statement and proof of subject reduction.

Definition 5.2 Given two resource algebras $\mathcal{S}$ and $\mathcal{R}$, an *approximation* is a relation $\sqsubseteq$ included in $S \times R$, which reads “static effect s approximates dynamic resource r”, with the following properties:

(i) $0^{\mathcal{S}} \sqsubseteq 0^{\mathcal{R}}$ and for every constructor c, it holds $\mathcal{S}^c \sqsubseteq \mathcal{R}^c$.
(ii) If $s \sqsubseteq r$, then $\mathcal{R}^{\mathsf{meth}}_{C,m,\overline{a}}(s) \sqsubseteq \mathcal{R}^{\mathsf{meth}}_{C,m,\overline{a}}(r)$.
(iii) $s_1 \sqsubseteq r_1 \wedge s_2 \sqsubseteq r_2$ entails $s_1 +^{\mathcal{S}} s_2 \sqsubseteq r_1 +^{\mathcal{R}} r_2$;
(iv) $s \sqsubseteq r$ entails $s +^{\mathcal{S}} s' \sqsubseteq r$.

For example, the static approximation $\mathsf{SG} = \langle \mathsf{S}, \{\mathsf{tt}\}, \cup, \leq_{\mathsf{SG}} \rangle$ of the $\mathsf{MethGuard}$ algebra, for $\mathsf{S} \equiv \{\{\mathsf{tt}\}, \{\mathsf{tt}, \mathsf{ff}\}\}$ and where the ordering is defined as $b \leq_{\mathsf{SG}} b'$ iff $b = \{\mathsf{tt}\}$ or $b = b' = \{\mathsf{tt}, \mathsf{ff}\}$. The approximation relation is $\in^{-1}$, which trivially satisfies the above conditions.

$$\frac{}{\Gamma \vdash a \ : \ \mathit{type}_\Gamma(a),\ \ \mathit{scost}(a)} \qquad\qquad \frac{\Gamma \vdash e \ : \ t,\ s \qquad s \leq s'}{\Gamma \vdash e \ : \ t,\ s'}$$

$$\frac{\Gamma \vdash a_1 \ : \ t_1,\ s_1 \qquad \Gamma \vdash a_2 \ : \ t_2,\ s_2 \qquad \Sigma(\mathsf{prim}) = t_1 \times t_2 \to t_3}{\Gamma \vdash \mathsf{prim}\ a_1\ a_2 \ : \ t_3,\ \ s_1 + s_2 + \mathcal{S}^{\mathsf{prim}}}$$

$$\frac{}{\Gamma \vdash \mathsf{new}\ C \ : \ C,\ \ \mathcal{S}^{\mathsf{new}}_C} \qquad\qquad \frac{\Gamma \vdash x \ : \ C,\ s \qquad \Sigma(C.f) = t}{\Gamma \vdash x.f \ : \ t,\ \ s + \mathcal{S}^{\mathsf{getf}}}$$

$$\frac{\Gamma \vdash x \ : \ C,\ s_x \qquad \Gamma \vdash a \ : \ t,\ s_a \qquad \Sigma(C.f) = t}{\Gamma \vdash x.f := a \ : \ \mathrm{unit},\ \ s_x + s_a + \mathcal{S}^{\mathsf{putf}}}$$

$$\frac{\Gamma \vdash e_1 \ : \ \mathrm{unit},\ s_1 \qquad \Gamma \vdash e_2 \ : \ t_2,\ s_2}{\Gamma \vdash e_1\ ;\ e_2 \ : \ t_2,\ \ s_1 + \mathcal{S}^{\mathsf{comp}} + s_2}$$

$$\frac{\Gamma \vdash e_1 \ : \ t_1,\ s_1 \qquad \Gamma, x : t_1 \vdash e_2 \ : \ t_2,\ s_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \ : \ t_2,\ \ s_1 + \mathcal{S}^{\mathsf{let}} + s_2}$$

$$\frac{\Gamma \vdash e \ : \ \mathrm{bool},\ s_e \qquad \Gamma \vdash e_1 \ : \ t,\ s_1 \qquad \Gamma \vdash e_2 \ : \ t,\ s_2}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \ : \ t,\ \ s_e + \mathcal{S}^{\mathsf{if}} + s_1 + s_2}$$

$$\frac{\Sigma(g) = t_1 \times \cdots \times t_n \to t, s}{\Gamma \vdash \mathsf{call}\ g : t, \mathcal{S}^{\mathsf{call}} + s}$$

$$\frac{\Gamma \vdash a_i \ : \ t_i,\ s_i \qquad \Sigma(m) = t_1 \times \cdots \times t_n \to t, s}{\Gamma \vdash C.m(\overline{a}) \ : \ t,\ \ \Sigma_i s_i + \mathcal{S}^{\mathsf{meth}}_{C,m,\overline{a}}(s)}$$

Figure 4. Typing rules

Let $E : \Gamma$ be the usual correspondence between environment and typing. Further say that a heap h is well-typed if $\Sigma(C.f) = t$ implies $h(l_C).f \ : \ t$. Finally, we say that a signature is *well-typed* if for all $g, m \in \Sigma$ it holds

$$\Sigma(g) = t_1 \times \cdots \times t_n \to t, s \Longrightarrow x_1 : t_1 \ldots x_n : t_n \vdash \mathit{body}_g : t, s$$
$$\Sigma(C.m) = t_1 \times \cdots \times t_n \to t, s \Longrightarrow x_1 : t_1 \ldots x_n : t_n \vdash \mathit{body}_{C,m} : t, s$$

Theorem 5.3 (Subject reduction) *Assume a well-typed signature, $\mathcal{S}$ and $\mathcal{R}$ as above. Suppose further that $\Gamma \vdash e \ : \ t, \ s$ and for a well typed heap h it holds $E \vdash h, e \Downarrow h', v, r$ and $E : \Gamma$; then $\Gamma \vdash v \ : \ t, \ \mathit{scost}(v)$ and $s \sqsubseteq r$.*

The above result ensures the consistency of the operational semantics with the type system; it is a basis for approximating dynamic measurements using type checking instead of theorem proving.

6 Conclusions

We have presented a framework and methodology for *optimisation validation*, based on generic forms of dynamic and static resource costs. We have formalised most of the setting in Isabelle/HOL, particularly including the soundness and completeness of the program logic, which was applied to validate the specific optimisations in Sect. 4.

One can argue against our approach in various ways. For example, validating optimisation with disregard of behavioural equivalence seems pointless (often, the empty program is the ultimate optimisation). Yet, we see resource improvement validation as orthogonal to translation validation; in some settings one may check both things. Optimising compilers usually use heuristics to decide what to do with code, but in some cases a sequence of transformations may not actually result in improvement even if correctness is preserved; the optimisation is then pointless[32]. In others settings, such as our PCC application, the safety policy that we care most about is captured by our resource consumption notion and so resource usage preservation is more crucial than functional equivalence.

6.1 Related Work.

By now there is an extensive literature on verifying compiler correctness and optimisations (e.g. [10,11,20]), but as far as we know, no previous work on *formal* and *static* methods for verifying that optimisations in fact improve resource usage. The closest are an early formal approach to performance estimation and monitoring for space and time complexity w.r.t. OO programs [27] and, on the other hand, a framework for *predicting* the impact of optimizations, via models for the latter as well for code and resources [32]. This is empirically tested and used to select the right combination and application strategy of given optimizations.

Specific instances of machine checked correctness proofs have also been pursued: to name a few w.r.t. dead code elimination [8], tokenization and componentization transformations [13].

One of the most well-developed approach is David Sands' *Improvement Theory* [26], a specialisation of the standard theory and reasoning principles of *observational* equivalence, in which basic observations include some intensional information about computational cost. This is extended to space improvements for effects-free call-by-need languages in [15]. "Paper and pencil" proofs are mostly equational and require considerable ingenuity even in the simplest cases. Our direction is inspired by *translation validation* (TV) [24], mainly implemented in the automatic TVOC tool [4]. It addresses both *reordering transformation* such as loop fusion and *structure preserving* ones, such as constant folding, where statements can be inserted or deleted. In both cases sound rules generate verification conditions entailing a bisimulation between source and target w.r.t. observable variables. Those VC's are fed to a theorem prover, in particular *CVC*. TV subsumes Rinard's *credible compilation* [25], which instead requires full code instrumentation. Necula [23] demonstrates an approach to TV based on symbolic execution in the context of the GNU C compiler intermediate language. Similarly to Rinard, he uses simulation of execution paths, but instead of compiler annotation, a constraint-based algorithm heuristically tries to infer a simulation. The system is robust enough to allow the author to verify structure preserving optimisations in gcc itself.

Several other researchers have considered *program logics* going beyond traditional functional correctness specification. For example [29] presents a generic Hoare calculus for reasoning about computational monads and is formalises in *HasCasl*. With a similar aim as us, Denney and Fischer [12] introduce a framework for safety policies: given a semantically defined safety property (such as "no division by zero") and an operational semantics, the aim is to derive specialised Hoare rules to enforce the property; however, for "stateful" properties, such as memory writes limits, the approach becomes technically quite involved.

Since we consider optimisation from one program to another, a natural approach suggests itself, namely using a logic which relates two programs at once. In Benton's relational Hoare logic [5] judgements $\{R\}\ c_1 \sim c_2\ \{S\}$ refer to the execution of two (possibly) different programs c_1 and c_2 while the pre- and post-conditions are relations (rather than predicates) over states. As a special case, the program logic contains a proof system in which (functional) equivalence of programs can be directly verified, in contrast to our approach where separate judgements are needed. A similar logic is used in Rinard's report [25]. In both cases, proofs of soundness are included while completeness is not examined.

Elsewhere, forms of *cost algebras* (monads) and *partial orders* similar to ours are considered for analysis of resource consumptions, e.g., [19,14] and optimisation [21]. General static analysis techniques which have similarities with the setup of our type and effect system include [16,30]. There is also considerable work on specific static analyses for different notions of resource usage: to name one, the use of abstract interpretation for certification of bound memory usage in Java byte code [9], but this wold lead us well beyond the scope of this overview.

6.2 Future work.

There are several avenues for pursuing this work. First, by considering finer-grained transformations individually, perhaps by generalising Improvement Theory to resource algebras. Second, it would be awesome if our static analysis was able to validate optimisations directly and avoid the need for the program logic: this is in fact possible in restricted (e.g. boolean) domains, but further assumptions are needed in the general case. To scale our techniques to routine application we would need either an automatic technique based on the type system or better automatic assistance for using the program logic. Endowing relational Hoare logics with a notion of resource algebra seems also a swift way to combine semantics preservation with optimisation validation.

Finally, the considerable generality of resource algebras allows examples that are less directly related to optimisation, but useful for validating other safety properties (including correspondence properties in protocols, or resource usage analysis in the sense of [18]), and we would like to apply our general techniques to those examples too.

The sources for the core logic (and much more) are available at:

`http://www.tcs.ifi.lmu.de/~hwloidl/mrg/MRG-infra-0805.tgz`

The examples presented in this paper can be downloaded from:

`http://homepages.inf.ed.ac.uk/amomigl1/papers/cocv06.tar`

Acknowledgement

This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein.

References

[1] Aspinall, D., L. Beringer, M. Hofmann, H.-W. Loidl and A. Momigliano, *A program logic for resource verification*, in: K. Slind, A. Bunker and G. Gopalakrishnan, editors, *TPHOLs2004*, LNCS **3223** (2004), pp. 34–49.

[2] Aspinall, D., L. Beringer, M. Hofmann, H.-W. Loidl and A. Momigliano, *A program logic for resources* (2005), to appear in Theoretical Computer Science.

[3] Aspinall, D., S. Gilmore, M. Hofmann, D. Sannella and I. Stark, *Mobile resource guarantees for smart devices*, in: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet and T. Muntean, editors, *CASSIS 2004*, LNCS **3362** (2005), pp. 1–26.

[4] Barrett, C. W., Y. Fang, B. Goldberg, Y. Hu, A. Pnueli and L. D. Zuck, *TVOC: A Translation Validator for Optimizing Compilers*, in: K. Etessami and S. K. Rajamani, editors, *CAV*, LNCS **3576** (2005), pp. 291–295.

[5] Benton, N., *Simple relational correctness proofs for static analyses and program transformations*, in: N. D. Jones and X. Leroy, editors, *POPL* (2004), pp. 14–25.

[6] Beringer, L., M. Hofmann, A. Momigliano and O. Shkaravska, *Automatic certification of heap consumption*, in: A. V. Franz Baader, editor, *LPAR 2004*, LNCS **3425** (2005), pp. 347–362.

[7] Beringer, L., K. MacKenzie and I. Stark, *Grail: a functional form for imperative mobile code*, in: *Foundations of Global Computing*, number 85.1 in ENTCS (2003), pp. 1–21.

[8] Blech, J. O., L. Gesellensetter and S. Glesner, *Formal verification of dead code elimination in Isabelle/HOL*, in: B. K. Aichernig and B. Beckert, editors, *SEFM* (2005), pp. 200–209.

[9] Cachera, D., T. Jensen, D. Pichardie and G. Schneider, *Certified memory usage analysis*, in: J. Fitzgerald, I. J. Hayes and A. Tarlecki, editors, *FM'05*, LNCS **3582** (2005), pp. 91–106.

[10] Cousot, P. and R. Cousot, *Systematic design of program transformation frameworks by abstract interpretation*, in: *POPL*, 2002, pp. 178–190.

[11] Dave, M. A., *Compiler verification: a bibliography*, SIGSOFT Softw. Eng. Notes **28** (2003), pp. 1–4.

[12] Denney, E. and B. Fischer, *Correctness of source-level safety policies*, in: K. Araki, S. Gnesi and D. Mandrioli, editors, *FME 2003*, LNCS **2805** (2003), pp. 894–913.

[13] Genet, T., T. P. Jensen, V. Kodati and D. Pichardie, *A Java Card CAP converter in PVS*, ENTCS **82** (2003).

[14] Grobauer, B., *Cost recurrences for DML programs*, in: *ICFP'01* (2001), pp. 253–264.

[15] Gustavsson, J. and D. Sands, *Possibilities and limitations of call-by-need space improvement*, in: *ICFP'01* (2001), pp. 265–276.

[16] Hankin, C. and D. L. Métayer, *A type-based framework for program analysis*, in: *SAS*, 1994, pp. 380–394.

[17] Hoare, C. A. R., *Procedures and parameters: An axiomatic approach*, in: E. Engeler, editor, *Symp. Semantics of Algorithmic Languages,*, Notes in Mathematics 188 (1971), pp. 102–116.

[18] Igarashi, A. and N. Kobayashi, *Resource usage analysis*, ACM SIGPLAN Notices **37** (2002), pp. 331–342.

[19] Jay, C. B., M. Cole, M. Sekanina and P. Steckler, *A monadic calculus for parallel costing of a functional language of arrays*, in: C. Lengauer, M. Griebl and S. Gorlatch, editors, *Euro-Par*, LNCS **1300** (1997), pp. 650–661.

[20] Kennedy, K. and J. R. Allen, "Optimizing compilers for modern architectures: a dependence-based approach," Morgan Kaufmann Publishers Inc., 2002.

[21] Knoop, J., O. Rüthing and B. Steffen, *Optimal code motion: Theory and practice*, ACM TOPLAS **16** (1994), pp. 1117–1155.

[22] MacKenzie, K. and N. Wolverson, *Camelot and Grail: resource-aware functional programming on the JVM*, , **4** (2004), pp. 29–46.

[23] Necula, G. C., *Translation validation for an optimizing compiler*, in: *PLDI '00* (2000), pp. 83–94.

[24] Pnueli, A., M. Siegel and E. Singerman, *Translation validation*, in: B. Steffen, editor, *TACAS*, LNCS **1384** (1998), pp. 151–166.

[25] Rinard, M., *Credible compilation*, Technical Report MIT-LCS-TR-776, MIT Laboratory for Computer Science (1999).

[26] Sands, D., *Improvement theory and its applications*, in: A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, Cambridge University Press, 1998 pp. 275–306.

[27] Schmidt, H. W. and W. Zimmermann, *A complexity calculus for object-oriented programs*, Hournal of Object-Oriented Systems **1** (1994), pp. 117–147.

[28] Schneider, F. B., *Enforceable security policies*, ACM Transactions on Information and System Security **3** (2000), pp. 30–50.

[29] Schröder, L. and T. Mossakowski, *Monad-independent Hoare logic in HasCASL*, in: M. Pezze, editor, *FASE*, LNCS **2621** (2003), pp. 261–277.

[30] Skalka, C. and S. F. Smith, *History effects and verification*, in: W.-N. Chin, editor, *APLAS*, LNCS **3302** (2004), pp. 107–128.

[31] Wadler, P. and P. Thiemann, *The marriage of effects and monads*, ACM Trans. Comput. Log. **4** (2003), pp. 1–32.

[32] Zhao, M., B. R. Childers and M. L. Soffa, *Predicting the impact of optimizations for embedded systems*, in: *LCTES* (2003), pp. 1–11.