Electronic Notes in Theoretical Computer Science 85 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume85.html 21 pages

Grail: a functional form for imperative mobile code

Lennart Beringer [1] Kenneth MacKenzie [1] Ian Stark [2]

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK
{lenb,kwxm,stark}@inf.ed.ac.uk

Abstract

In Robert Louis Stevenson's novel [31], Dr Jekyll is a well-regarded member of polite society, while his alter ego Mr Hyde shares the same physical form but roams abroad communing with the lowest elements. In this paper we present *Grail*, a well-behaved first-order functional language that is the target for an ML-like compiler; while also being a wholly imperative language of assignments that travels and executes as Java classfiles. We use this dual identity in the *Mobile Resource Guarantees* project, where Grail serves as proof-carrying code to provide assurances of time and space performance, thereby supporting secure and reliable global computing.

1 Introduction

A general technique for compiling functional languages onto imperative hardware is to transform the functional program into progressively more restrictive sublanguages, until arriving at a form that we can manageably translate across into some imperative assembler like C or Java bytecode. In this paper we take the process to its limit, and present a simple functional language which requires no translation at all for imperative execution. Programs in *Grail*, such as that in Figure 1, can either be evaluated functionally with a call-by-value semantics, or executed imperatively with state and `goto` — with both routes giving exactly the same result. In the functional reading "`x = 5`" is a lexically-scoped declaration; on imperative execution it updates a named storage cell.

On the theory side, these two semantics allow us to make precise some folklore correspondences between imperative and functional compilation, in

[1] Supported by the *Mobile Resource Guarantees* project (MRG) funded by the EC under the FET proactive initiative on Global Computing (IST-2001-33149).
[2] Supported by an Advanced Research Fellowship from the UK EPSRC.

```
method static int fib (int n) =
    let
        val a = 0                              // Local variable declarations
        val b = 1

        fun loop (int a, int b, int n) =       // Local function declaration
            let
                val b = add a b                // Lexically scoped variables
                val a = sub b a                // hide outer declarations
                val n = sub n 1
            in
                test(n,a,b)                    // Tail recursive function call
            end

        fun test (int n, int a, int b) =       // Another function declaration
            if n<=1 then b else loop(a,b,n)    // Conditional recursive call
    in
        test(n,a,b)                            // Main expression
    end
```

Fig. 1. Grail code for the Fibonacci number F_n. For speed, we keep track of both F_k and F_{k+1} together in accumulating parameters a and b. Appendix A shows a complete program.

the spirit of Appel's observations on SSA [3]. In particular we show results relating liveness analysis in Grail to the free variables of recursive functions, and connect dataflow information on imperative single-use registers to a functional linear type system.

More practically, the motivation for Grail comes from implementation concerns of proof-carrying code. In the *Mobile Resource Guarantees* project we plan to annotate mobile code with proofs of time and space usage. These proofs will be generated from various resource-aware type systems that have been developed for high-level functional languages [25,15,14,1,6,16]. For this we need a form of mobile code that is raw enough to execute on common platforms, yet sufficiently well-behaved to support such proofs.

The design of Grail aims to satisfy these demands: our programs execute as Java bytecode, yet the proofs address a readable ML-like form whose operational semantics we have formalized in the Isabelle theorem prover [28]. Simplifications from both the functional and imperative viewpoints combine to give code that is manageable for automated reasoning and efficient to analyze.

1.1 Proof-Carrying Code

Proof-carrying code (PCC, [26]) has emerged as an important component of trustworthy global computation. Complementing aspects of authenticity

and secrecy, it provides a mechanism for guaranteeing intensional properties of code such as adherence to security policies or resource limitations. The central idea is that when a piece of software passes from a code producer to a code consumer, it is equipped with a compact formal proof of its safety. The consumer then mechanically checks this proof before executing any code. The computationally more challenging task of creating proofs lies with the code producer and may be achieved using arbitrary means, including program analysis and programmer intervention. Security is assured without relying on networks of trust to link producer and consumer — indeed, it may not always be necessary to know the identities of code and proof providers. Any tampering with the proof or program during the transmission is either detected during proof checking or is irrelevant as far as the satisfaction of the safety policy is concerned.

Any PCC framework must fix on a representation for transmitted programs, and a logic to formulate proofs about them. Typically code will be in some machine language, with a logic tailored to its properties. However, there are pragmatic tradeoffs here, so for example *foundational* PCC takes a very general logic and within the logic itself prepares a fully formal description of the execution of a real processor [4,5].

From a global computing perspective the usage of raw machine language is undesirable – we can greatly increase code mobility by choosing some widely-deployed virtual machine such as that for Java or .NET [10,19,23]. The logic is then given in terms of a virtual machine model whose validity is either assumed or certified at lower levels of abstraction. In addition, these frameworks provide standard mechanisms for type-checking and otherwise validating mobile bytecode, which gives us an environment more amenable to formal proofs.

1.2 Mobile Resource Guarantees

Figure 2 presents a general outline of the MRG system. We have a high-level ML-family language *Camelot*, which compiles by transformation into functional Grail [22]. To support reasoning and proofs about Grail programs, we have a formal encoding in Isabelle of their behaviour and resource usage, together with a prototype Hoare logic and verification condition generator.

For transmission to the code consumer, we assemble the Grail into Java classfile binaries; proofs of resource usage may travel either separately, or wrapped with the classfile in a jar archive. On arrival the consumer extracts the original Grail and checks it against their resource policy using the proof provided. If this succeeds then the binary itself is passed to the Java VM for execution.

At the moment the consumer side is carried out by a number of distinct programs, as the system is still under development. We propose eventually to integrate these as a specialised classloader within the standard Java framework.

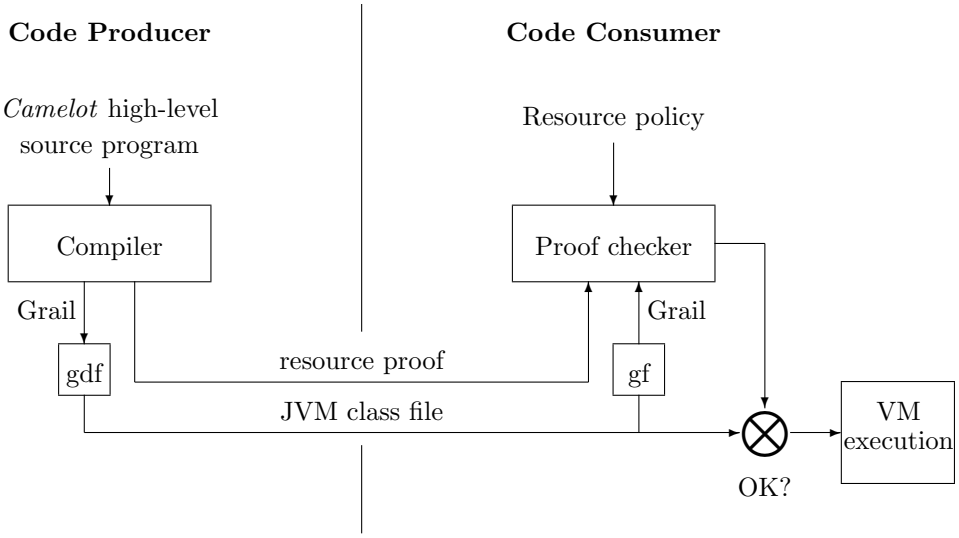


Fig. 2. Outline of the MRG framework for resource-aware proof-carrying code.

Although our current implementation is on the Java virtual machine, one aim for Grail is to keep it within the intersection of Java, JavaCard [32], and the CIL bytecode of the Microsoft .NET framework [23]. We anticipate supporting any of these three formats equally well, from the same Grail sources [21].

Whatever the platform, we want mobile proofs to take advantage of the abstraction Grail provides over raw VM code. For this, the conversion between the two must be *efficient* to implement, *reversible*, and *cost-preserving* with respect to a VM resource model [9]. A brief survey of Grail's functional and imperative sides will show how we achieve this.

1.3 Functional Grail

Figure 1 shows Grail code to calculate Fibonacci numbers. The accompanying comments highlight its reading as a conventional strict first-order functional language:

- call-by-value function invocation;
- lexical scoping for variables;
- mutually recursive local function declarations;
- strict static typing.

Within this, Grail makes several simplifications appropriate to its role as a compiler target language. For example, local function declarations may not nest, only tail calls are allowed, and expressions can contain just one operation. Section 2 puts these into a grammar for Grail; later we shall see some further constraints on control and dataflow.

This Fibonacci code is the body of a single method. Above this, Grail provides precisely the class and object structure built into the Java virtual

```
method static int fib (int n) =
    let
        val a = 0                              // Initial assignment
        val b = 1                              //   to variables

        fun loop (int a, int b, int n) =       // Labelled basic block, with
            let                                //   live variable annotation
                val b = add a b
                val a = sub b a                // Sequence of assignments
                val n = sub n 1                //   updating named registers
            in
                test(n,a,b)                    // Goto, with live variable
            end                                //   annotation

        fun test (int n, int a, int b) =       // Another labelled basic block
            if n<=1 then b else loop(a,b,n)    // Conditional return or jump
    in
        test(n,a,b)                            // Initial entry label
    end
```

Fig. 3. Imperative Grail code to calculate the Fibonacci number F_n. Comments indicate semantics for execution on the Java virtual machine.

machine. Hence the basic expression operators contain not just add and sub but also primitives to create and manipulate objects. These update the Java heap, and are used in Camelot to compile algebraic datatypes.

These side-effecting operators are necessary both to capture the actual resource behaviour of the JVM, and also to implement update-in-place optimisations enabled by resource-aware type systems such as Hofmann's LFPL [6,14,16].

1.4 Imperative Grail

As an alternative view of the same code, Figure 3 shows how our Fibonacci routine can be seen as a purely imperative stream of assignment statements and jumps. Instead of local functions we have a collection of basic blocks, function calls are merely jumps, and parameter lists now track which variables are live. Despite this very different gloss, both functional and imperative semantics give the same results. This is no accident, of course, and in Section 2.3 we show that it holds for all code satisfying certain constraints.

This imperative reading gives a direct map from Grail onto Java bytecode: variables are JVM variables, and each statement expands to a short sequence of instructions, which are then composed exactly as laid out in the Grail source. For example:

val b = add a b	becomes	9 iload_1	13 iload_2	17 iload_0
val a = sub b a		10 iload_2	14 iload_1	18 iconst_1
val n = sub n 1		11 iadd	15 isub	19 isub
		12 istore_2	16 istore_1	20 istore_0

This gives bytecode that is highly stereotyped, and we can recover the original Grail simply by clustering instruction sequences. We can even identify variable names from standard JVM metadata.

One of the guides for this disassembly is that the JVM operand stack is empty between imperative Grail statements. This compares with arbitrary bytecode, where the stack may remain nonempty for any number of instructions. Similarly, Grail variables keep the same type throughout a method body, where general JVM registers need not do so. These features make Grail easier to analyze than general bytecode; for example, Grail programs immediately satisfy Leroy's conditions for efficient on-chip JavaCard verification [18]. (In fact, although the JVM in principle allows various elaborate uses of local stack and variables, Leroy observes that current Java compilers rarely take advantage of this.)

There still remains the issue of how to get functional code into a form that can be treated imperatively. We use the term *Grail normal form* to describe code where functions and methods have no free variables, and at every function invocation the actual arguments are syntactically identical to the formal parameters from the function declaration. To satisfy the first of these we need only perform standard λ-lifting. For the second, we may have to rename variables and even insert extra declarations to set them up before calls. All these are legitimate functional rearrangements, but they also correspond directly to imperative compilation techniques: namely conversion to static single-assignment form (SSA) and then elimination of Φ-functions [3].

This is an instance of a more general observation, that low-level transformations on registers and imperative variables map to functional transformations of Grail. Thus we can carry out certain optimisations like register allocation and sharing while still in the intermediate language of our compiler [35].

2 Syntax and operational semantics

In this section we give a formal treatment of Grail by defining the syntax of method bodies and presenting their operational semantics from both functional and imperative viewpoints. For the purposes of this paper we take a desugared, single-typed and slightly simplified Grail, looking only at the code within a single method declaration; for a description of the full language see [20].

We assume mutually disjoint sets *Vars* of variables (ranged over by x, y, x_i, ...), *Mnames* of method names (ranged over by $M, M_i, \ldots$) and *Fnames* of function names (ranged over by $f, f_i, \ldots$). We also assume sets *Consts* of constants c (with $0 \in \mathit{Consts}$) and P of primitive operations p. Sample

primitives are the arithmetic operations on integers, as well as primitives for object creation and manipulation, and method invocation.

The syntax of method declarations is as follows:

$$\begin{aligned} \mathit{decl} &::= \texttt{method}\ M(\vec{y}) = \mathit{body} \\ \mathit{body} &::= \texttt{fun}\ \mathit{funblock}\ \texttt{in}\ \mathit{term}\ \texttt{end} \\ \mathit{term} &::= \mathit{result} \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{result}\ \texttt{else}\ \mathit{result} \mid \\ &\qquad \texttt{let}\ x = \mathit{expr}\ \texttt{in}\ \mathit{term} \\ \mathit{funblock} &::= f(\vec{x}) = \mathit{term}\ \langle \texttt{and}\ \mathit{funblock} \rangle \\ \mathit{result} &::= x \mid f(\vec{x}) \\ \mathit{expr} &::= x \mid c \mid p(\vec{x}) \end{aligned}$$

We always assume that method parameters y_i are mutually distinct, and likewise the formal arguments x_i of function declarations. Furthermore, we assume that each called function is declared exactly once, that the arity of function calls and function declarations match and that primitive operations are applied with the correct number of arguments. We let Vars_M denote the set of variables x occurring in *decl* (including the method parameters) and use $\mathit{fv}(.)$ to denote the sets of syntactically free variables for the various phrase classes which are given as usual.

2.1 Functional semantics

The functional interpretation of a method $\texttt{method}\ M(y_1, \ldots, y_n) = \mathit{body}$ is given by a call-by-value big-step evaluation relation over the following semantic domains. *Value environments* $\mathcal{E} : \mathit{Vars}_M \rightharpoonup \mathit{Consts}$ and *closure environments* Σ associating triples $(\mathcal{E}, \vec{x}, \mathit{term})$ to function names are modelled as partial maps. We write $\mathit{dom}\ \mathcal{E}$ for the domain of $\mathcal{E}$, $[\,]$ for the empty map, $\mathcal{E}[x \mapsto c]$ for the update operation and $\mathcal{E}\,|_V$ for the restriction of $\mathcal{E}$ to the domain V, where $V \subseteq \mathit{Vars}$. Similar notation is used for closure environments and other maps. In order to cater for potential side effects of primitive operations we also include a domain of *heaps* $\mathcal{H}$ but leave their internal structure unspecified.

Evaluation is defined structurally: invoking M with arguments $c_1, \ldots, c_n$ in heap $\mathcal{H}$ evaluates to c if $\mathcal{H} \vdash M(c_1, \ldots, c_n) \Downarrow c, \mathcal{H}'$ is derivable for some $\mathcal{H}'$ using the rules in Figures 4 to 7. In rule F-PRIM, $\overline{p}$ represents the semantic operator corresponding to the primitive operation p – its evaluation possibly depends upon, and affects, the heap. Notice that one global environment for functions suffices as all functions are defined at the same (top) level.

2.2 Imperative semantics

The imperative semantics interprets function names as labels, treats variables imperatively and translates function calls into jumps. We define it as a small-step execution relation $\mathcal{T} \vdash (\sigma, \mathit{term}) \rightarrow (\sigma', \mathit{term}')$ between states

$$\sigma \in \mathit{State} = (\mathit{Var} \rightharpoonup \mathit{Consts}) \times \mathit{Heap}$$

$$\text{F-INV}\frac{([y_1 \mapsto c_1, \ldots, y_n \mapsto c_n], \mathcal{H}) \vdash \mathit{body} \Downarrow c, \mathcal{H}'}{\mathcal{H} \vdash M(c_1, \ldots, c_n) \Downarrow c, \mathcal{H}'}$$

$$\text{F-MB}\ \frac{(\mathcal{E}, [\,]) \vdash \mathit{funblock} \Downarrow \Sigma \quad (\mathcal{E}, \mathcal{H}, \Sigma) \vdash \mathit{term} \Downarrow c, \mathcal{H}'}{(\mathcal{E}, \mathcal{H}) \vdash \texttt{fun}\ \mathit{funblock}\ \texttt{in}\ \mathit{term}\ \texttt{end} \Downarrow c, \mathcal{H}'}$$

Fig. 4. Functional semantics: method invocation and method bodies

$$\text{F-VAR}\frac{x \in \mathit{dom}\ \mathcal{E}}{\mathcal{E}, \mathcal{H} \vdash x \Downarrow \mathcal{E}(x), \mathcal{H}} \qquad \text{F-CONST}\frac{}{\mathcal{E}, \mathcal{H} \vdash c \Downarrow c, \mathcal{H}}$$

$$\text{F-PRIM}\ \frac{\forall i. \mathcal{E}, \mathcal{H} \vdash x_i \Downarrow c_i, \mathcal{H}}{\mathcal{E}, \mathcal{H} \vdash p(x_1, \ldots, x_n) \Downarrow c, \mathcal{H}'}\ \overline{p}(\mathcal{H}, c_1, \ldots, c_n) = (c, \mathcal{H}')$$

Fig. 5. Functional semantics: phrase class *expr*

$$\text{F-CALL}\ \frac{\mathcal{E}, \mathcal{H} \vdash x_i \Downarrow c_i, \mathcal{H} \quad (\mathcal{E}_2, \mathcal{H}, \Sigma) \vdash \mathit{term} \Downarrow c, \mathcal{H}'}{(\mathcal{E}, \mathcal{H}, \Sigma) \vdash f(\vec{x}) \Downarrow c, \mathcal{H}'}\ \begin{cases} \Sigma(f) = (\mathcal{E}_1, \vec{z}, \mathit{term}) \\ \mathcal{E}_2 = \mathcal{E}_1[z_i \mapsto c_i] \end{cases}$$

$$\text{F-RET}\frac{\mathcal{E}, \mathcal{H} \vdash x \Downarrow c, \mathcal{H}}{(\mathcal{E}, \mathcal{H}, \Sigma) \vdash x \Downarrow c, \mathcal{H}}$$

$$\text{F-IFT}\ \frac{\mathcal{E}, \mathcal{H} \vdash \mathit{expr} \Downarrow 0, \mathcal{H}' \quad (\mathcal{E}, \mathcal{H}', \Sigma) \vdash \mathit{result}_1 \Downarrow c, \mathcal{H}''}{(\mathcal{E}, \mathcal{H}, \Sigma) \vdash \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{result}_1\ \texttt{else}\ \mathit{result}_2 \Downarrow c, \mathcal{H}''}$$

$$\text{F-IFF}\ \frac{\mathcal{E}, \mathcal{H} \vdash \mathit{expr} \Downarrow c_1, \mathcal{H}' \quad (\mathcal{E}, \mathcal{H}', \Sigma) \vdash \mathit{result}_2 \Downarrow c, \mathcal{H}''}{(\mathcal{E}, \mathcal{H}, \Sigma) \vdash \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{result}_1\ \texttt{else}\ \mathit{result}_2 \Downarrow c, \mathcal{H}''}\ c_1 \neq 0$$

$$\text{F-LET}\frac{\mathcal{E}, \mathcal{H} \vdash \mathit{expr} \Downarrow c_1, \mathcal{H}' \quad (\mathcal{E}[x \mapsto c_1], \mathcal{H}', \Sigma) \vdash \mathit{term} \Downarrow c, \mathcal{H}''}{(\mathcal{E}, \mathcal{H}, \Sigma) \vdash \texttt{let}\ x = \mathit{expr}\ \texttt{in}\ \mathit{term} \Downarrow c, \mathcal{H}''}$$

Fig. 6. Functional semantics: phrase classes *result* and *term*

$$\text{F-FDEC}\ \frac{}{(\mathcal{E}, \Sigma) \vdash f(\vec{x})\ = \mathit{term} \Downarrow \Sigma'}\ \begin{cases} V = \mathit{fv}(\texttt{fun}\ f(\vec{x})\ = \mathit{term}) \\ \Sigma' = \Sigma[f \mapsto (\mathcal{E}\lfloor_V, \vec{x}, \mathit{term})] \end{cases}$$

$$\text{F-FBLK}\ \frac{(\mathcal{E}, \Sigma') \vdash \mathit{funblock} \Downarrow \Sigma''}{(\mathcal{E}, \Sigma) \vdash f(\vec{x})\ = \mathit{term}\ \texttt{and}\ \mathit{funblock} \Downarrow \Sigma''}\ \begin{cases} V = \mathit{fv}(\texttt{fun}\ f(\vec{x})\ = \mathit{term}) \\ \Sigma' = \Sigma[f \mapsto (\mathcal{E}\lfloor_V, \vec{x}, \mathit{term})] \end{cases}$$

Fig. 7. Functional semantics: phrase class *funblock*

which is given by the rules in Figures 8 and 9, where $\mathcal{T} : f \rightharpoonup \mathit{term}$ maps labels to basic blocks. Notice the absence of closures and the fact that jumps ignore the arguments associated to the label.

Invoking M with arguments $c_1, \ldots, c_n$ in $\mathcal{H}$ results in c if the judgement

$$\text{I-CONST}\frac{}{(s,\mathcal{H}) \vdash c \downarrow c, \mathcal{H}} \quad \text{I-VAR}\frac{}{(s,\mathcal{H}) \vdash x \downarrow s(x), \mathcal{H}}\ x \in dom\ s$$

$$\text{I-PRIM}\ \frac{}{(s,\mathcal{H}) \vdash p(x_1,\ldots,x_n) \downarrow c, \mathcal{H}'}\ \begin{cases} \forall i. x_i \in dom\ s \\ \overline{p}(\mathcal{H}, s(x_1),\ldots,s(x_n)) = (c,\mathcal{H}') \end{cases}$$

Fig. 8. Imperative semantics: phrase class *expr*

$$\text{I-LET}\frac{(s,\mathcal{H}) \vdash expr \downarrow c, \mathcal{H}'}{\mathcal{T} \vdash ((s,\mathcal{H}), \texttt{let}\ x = expr\ \texttt{in}\ term) \to ((s[x \mapsto c], \mathcal{H}'), term)}$$

$$\text{I-IFT}\ \frac{(s,\mathcal{H}) \vdash expr \downarrow 0, \mathcal{H}'}{\mathcal{T} \vdash ((s,\mathcal{H}), \texttt{if}\ expr\ \texttt{then}\ result_1\ \texttt{else}\ result_2) \to ((s,\mathcal{H}'), result_1)}$$

$$\text{I-IFF}\ \frac{(s,\mathcal{H}) \vdash expr \downarrow c, \mathcal{H}' \qquad c \neq 0}{\mathcal{T} \vdash ((s,\mathcal{H}), \texttt{if}\ expr\ \texttt{then}\ result_1\ \texttt{else}\ result_2) \to ((s,\mathcal{H}'), result_2)}$$

$$\text{I-JMP}\frac{}{\mathcal{T} \vdash (\sigma, f(\vec{x})) \to (\sigma, \mathcal{T}(f))}\ f \in dom\ \mathcal{T}$$

Fig. 9. Imperative semantics: phrase classes *result* and *term*

$\mathcal{H} \vdash M(c_1,\ldots,c_n) \downarrow c, \mathcal{H}'$ is derivable for some $\mathcal{H}'$ using the rules

$$\text{I-INV}\ \frac{([y_1 \mapsto c_1,\ldots,y_n \mapsto c_n], \mathcal{H}) \vdash body \downarrow c, \mathcal{H}'}{\mathcal{H} \vdash M(c_1,\ldots,c_n) \downarrow c, \mathcal{H}'}$$

$$\text{I-MB}\ \frac{\mathcal{T}_{funblock} \vdash (\sigma, term) \to^* ((s,\mathcal{H}), x)}{\sigma \vdash \texttt{fun}\ funblock\ \texttt{in}\ term\ \texttt{end} \downarrow s(x), \mathcal{H}}\ x \in dom\ s$$

where

$$\mathcal{T}_{funblock} = \begin{cases} [f \mapsto term] & \text{if}\ \ funblock \equiv f(\vec{x}) = term \\ \mathcal{T}_{funblock_1}[f \mapsto term] & \text{if}\ \ funblock \equiv f(\vec{x}) = term\ \texttt{and}\ funblock_1 \end{cases}$$

and $\to^*$ is the reflexive transitive closure of $\to$.

2.3 Coincidence of functional and imperative semantics

As promised, we demonstrate that the functional and imperative operational semantics coincide, but only for a particular class of well-formed programs.

Definition 2.1 A declaration `method` $M(\vec{y})$ = `fun` *funblock* `in` *term* `end` is in *Grail normal form* (GNF) if

- $fv(funblock) = \emptyset$ (local functions are closed),
- $fv(term) \subseteq \{y_1,\ldots,y_n\}$ (all free variables are amongst the method parameters) and
- for each call $f(\vec{x})$, the declaration of f is exactly $f(\vec{x}) = \ldots$

The first condition requires all functions to be fully λ-lifted. As a consequence, value environments in closures will always be empty, so that dynamic binding and static binding coincide. The third condition ensures that

the calling functions deposit the values "in the right variables" - indeed, rule F-CALL simplifies to

$$\text{F-CALL1}\frac{(\mathcal{E}\downarrow_{\{x_1,\ldots,x_n\}}, \mathcal{H}, \Sigma) \vdash \mathit{term} \Downarrow c, \mathcal{H}'}{(\mathcal{E}, \mathcal{H}, \Sigma) \vdash f(\vec{x}) \Downarrow c, \mathcal{H}'} \begin{cases} \Sigma(f) = ([\,], \vec{x}, \mathit{term}) \\ \{x_1, \ldots, x_n\} \subseteq \mathit{dom}\ \mathcal{E} \end{cases}$$

and no copying of values between registers is necessary. Finally, the first and second condition in combination ensure that the side-condition $\{x_1, \ldots, x_n\} \subseteq \mathit{dom}\ \mathcal{E}$ here is always fulfilled, so even the inclusion of surplus variables in a list of function parameters will not lead to a difference in behaviour.

This is enough for the functional and imperative semantics to match.

Theorem 2.2 *Let* `method` $M(y_1, \ldots, y_n) = \mathit{body}$ *be a method declaration in GNF. Then* $\mathcal{H} \vdash M(c_1, \ldots, c_n) \Downarrow c, \mathcal{H}'$ *if and only if* $\mathcal{H} \vdash M(c_1, \ldots, c_n) \downarrow c, \mathcal{H}'$.

The proof proceeds by first stating a lemma on the coincidence of expression evaluation, followed by an induction on the structure of derivations.

The correspondence between functional and imperative semantics extends to a precise matching of resource usage: for example, the stern condition on function parameters means that when function calls are mapped to imperative jumps, there is no hidden cost of register rearrangement. This is naturally relevant for our PCC application, where the systems we have available for inferring high-level information about time and space usage of programs work with strongly-typed functional languages [1,6,14,15,16,25]. Accordingly, Camelot is such a language, and we compile it by successive transformations into functional Grail. Of course, the code is then transmitted and executed as imperative Java bytecode; but the close semantic correspondence means that resource guarantees remain valid.

3 Liveness analysis and function parameters

In this section, we show that the formal parameters of local functions are intimately linked to the imperative concept of liveness. This is relevant for the assertions of our Hoare logic and the generation of verification conditions.

Following the framework of [27], we present liveness analysis as a dataflow analysis where elementary statements carry labels $\ell, \ell_i, \ldots$ from an infinite set **Lab**. Formally, labelled methods arise from ordinary methods if we replace the phrase class *term* by

$$\begin{aligned} \mathit{term} ::= {} & [\mathit{result}]^\ell \mid \texttt{if}\ [\mathit{expr}]^\ell\ \texttt{then}\ [\mathit{result}]^\ell\ \texttt{else}\ [\mathit{result}]^\ell \mid \\ & \texttt{let}\ [x = \mathit{expr}]^\ell\ \texttt{in}\ \mathit{term} \end{aligned}$$

For the remainder of this paper, let $\mathit{decl} \equiv$ `method` $M(\vec{y}) = \mathit{body}$ be a fixed labelled method declaration with $\mathit{body} \equiv$ `fun` *funblock* `in` *term* `end` and $\mathcal{T} = \mathcal{T}_{\mathit{funblock}}$, such that the labelling is a bijection between the subset $\mathbf{Lab}_M$ of labels occurring in *body* and the labelled phrases.

Initial labels and the set of flow edges in M are given by

phrase ph	$init(ph)$	$flow(ph)$
`method` $M(\vec{y}) = body$	$init(body)$	$flow(body)$
`fun` *funblock* `in` *term* `end`	$init(term)$	$flow(funblock) \cup flow(term)$
$[x]^\ell$	ℓ	$\emptyset$
$[f(\vec{x})]^\ell$	ℓ	$\{(\ell, init(\mathcal{T}(f)))\}$
`let` $[x = expr]^\ell$ `in` $term_1$	ℓ	$\{(\ell, init(term_1))\} \cup flow(term_1)$
`if` $[expr]^\ell$ `then` $[result_1]^{\ell_1}$ `else` $[result_2]^{\ell_2}$	ℓ	$\cup_{i=1}^{2}(flow([result_i]^{\ell_i}) \cup \{(\ell, \ell_i)\})$

where $flow(funblock) = \cup_{f \in dom\ \mathcal{T}} flow(\mathcal{T}(f))$. The *reverse flow* of M is given by $rflow(M) = \{(\ell', \ell) \mid (\ell, \ell') \in flow(decl)\}$ and the set of *final labels* of M is given by $final(M) = \{\ell \mid \nexists \ell'.(\ell, \ell') \in flow(decl)\}$.

To each label from $\mathbf{Lab}_M$ we associate two sets $kill(\ell), gen(\ell) \subseteq Vars_M$

Occurrence of ℓ	$kill(\ell)$	$gen(\ell)$
$[x]^\ell$	$\emptyset$	$\{x\}$
$[f(\vec{x})]^\ell$	$\emptyset$	$\emptyset$
`let` $[x = expr]^\ell$ `in` ...	$\{x\}$	$fv(expr)$
`if` $[expr]^\ell$ `then` ...	$\emptyset$	$fv(expr)$

Notice that the arguments of function calls do not generate liveness, in accordance with the standard definition of $gen(\ell)$ in an analysis for an imperative language.

Definition 3.1 A pair $\mathsf{lv} = (\mathsf{lv}_{entry}, \mathsf{lv}_{exit})$ of functions $\mathsf{lv}_{entry}, \mathsf{lv}_{exit} : \mathbf{Lab}_M \to 2^{Vars_M}$ is called a *liveness solution* for m if the constraints

$$\mathsf{LV}_{exit}(\ell) \supseteq \begin{cases} \emptyset & \text{if } \ell \in final(M) \\ \cup_{(\ell',\ell) \in rflow(M)} \mathsf{LV}_{entry}(\ell') & \text{otherwise} \end{cases} \quad (1)$$

$$\mathsf{LV}_{entry}(\ell) \supseteq (\mathsf{LV}_{exit}(\ell) \setminus kill(\ell)) \cup gen(\ell) \quad (2)$$

are satisfied for the substitution $[\mathsf{lv}_{entry}/\mathsf{LV}_{entry}, \mathsf{lv}_{exit}/\mathsf{LV}_{exit}]$

For methods in GNF, the method parameters and the formal parameters of local functions correspond to liveness at the associated program points.

Theorem 3.2 *If decl is in GNF then there is a solution* $\mathsf{lv} = (\mathsf{lv}_{entry}, \mathsf{lv}_{exit})$ *for* M *such that* $\mathsf{lv}_{entry}(init(body)) = \{y_1, \ldots, y_n\}$ *and for all* $f \in dom\ \mathcal{T}$, $\mathsf{lv}_{entry}(init(\mathcal{T}(f))) = \{x_1, \ldots, x_n\}$ *where* `fun` $f(x_1, \ldots, x_n) = \ldots$ *is the declaration for* f *in funblock.*

Conversely, each liveness solution determines a transformation of *decl* where the formal parameters of functions are exactly the live-in variables.

Theorem 3.3 *Let decl be in GNF and* lv *a solution for* M *which fulfils* $\mathsf{lv}_{entry}(init(body)) = \{y_1, \ldots, y_n\}$. *For each* $\ell \in \mathbf{Lab}_M$ *choose an enumera-*

tion Entry(ℓ) *of* $\mathsf{lv}_{entry}(\ell)$*. Define*

$$decl' \equiv \texttt{method}\ M'(\vec{y})\ =\ \texttt{fun}\ funblock'\ \texttt{in}\ term'$$

where for each $f \in dom\ \mathcal{T}$*, the formal parameters of* f*'s declaration are modified to Entry*$(init(\mathcal{T}(f)))$ *and function calls are updated accordingly. Then* $decl'$ *is a method declaration in GNF and is functionally equivalent to decl, i.e.* $\mathcal{H} \vdash M(c_1, \ldots, c_n) \Downarrow c, \mathcal{H}'$ *holds exactly if* $\mathcal{H} \vdash M'(c_1, \ldots, c_n) \Downarrow c, \mathcal{H}'$*.*

In the PCC setting, these results allow the code consumer to recover the formal parameters by performing a liveness analysis. Alternatively, we may communicate the formal arguments in the meta-information or as part of the proof and only verify their consistency with the free variables. The latter task does not require a fixed point iteration and is thus conceptually and computationally simpler.

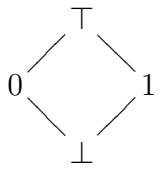
4 Read-once variables and linear typing

We can extend the close relationship between functional variables and imperative registers to more fine-grained usage properties. In this section we summarise a generalisation of liveness analysis which detects when a value written to a register is accessed exactly once. This has applications to memory management and operand forwarding in asynchronous processors; for full details of the analysis and its motivation, see Beringer's thesis [8]. We then outline a linear type system for achieving the same task in the functional interpretation. While the notion of linearity differs slightly from that usually considered in the literature, the correspondence between dataflow analysis and type system may again be made formally precise: each typing derivation gives imperative read-once information, and each solution to the dataflow system determines a typing derivation.

4.1 Imperative read-once analysis

The analysis targets the use of variables in the first component (*store*) of imperative states $(s, \mathcal{H})$. A variable x is called to be *read-once* if any read access to x may be implemented destructively, *i.e.* result in the removal of the entry for x in s.

An appropriate (conservative) dataflow analysis for identifying read-once variables was presented in [8], and generalises the liveness analysis. Instead of associating elements from 2^{Vars_M} to labels, we use the lattice $(\mathcal{S}, \bot, \sqsubseteq, \sqcup)$ illustrated below:



$\oplus$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	1	$\top$
0	$\bot$	0	1	$\top$
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

equipped with a commutative and monotone operation $\oplus$. The $\bot$ element represents unknown information, $\top$ indicates contradictory usage in different branches or existence of more than one use, and elements 0 and 1 represent exactly none and exactly one use, respectively.

Variable usage in expressions is captured by functions $use(expr) : Var_M \to \mathcal{S}$

$$\begin{aligned} use(c)(x) &= 0 \\ use(y)(x) &= \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \\ use(p(x_1,\ldots,x_n))(x) &= \begin{cases} \oplus_{i=1}^{n} use(x_i)(x) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases} \end{aligned} \tag{3}$$

and the usage in elementary blocks is defined by $usage : \mathbf{Lab}_M \to Vars_M \to \mathcal{S}$

Occurrence of ℓ	$usage(\ell)(x)$
$[y]^\ell$	$use(y)(x)$
$[f(\vec{x})]^\ell$	0
`let` $[x = expr]^\ell$ `in ...`	$use(expr)(x)$
`if` $[expr]^\ell$ `then ...`	$use(expr)(x)$

The dataflow equations for $\mathsf{RO}_{exit}, \mathsf{RO}_{entry} : \mathbf{Lab}_M \to Vars_M \to \mathcal{S}$ are given pointwise by

$$\mathsf{RO}_{exit}(\ell)(x) = \begin{cases} 0 & \text{if } \ell \in final(M) \\ \bigsqcup_{(\ell',\ell)\in rflow(M)} \mathsf{RO}_{entry}(\ell')(x) & \text{otherwise} \end{cases} \tag{4}$$

$$\mathsf{RO}_{entry}(\ell)(x) = \begin{cases} usage(\ell)(x) & \text{if } x \in kill(\ell) \\ \mathsf{RO}_{exit}(\ell)(x) \oplus usage(\ell)(x) & \text{otherwise} \end{cases} \tag{5}$$

Definition 4.1 A pair $\mathsf{ro} = (\mathsf{ro}_{entry}, \mathsf{ro}_{exit})$ of functions

$$\mathsf{ro}_{exit}, \mathsf{ro}_{entry} : \mathbf{Lab}_M \to Vars_M \to \mathcal{S}$$

is a *solution* for M if (4) and (5) are fulfilled for $[\mathsf{ro}_{exit}/\mathsf{RO}_{exit}, \mathsf{ro}_{entry}/\mathsf{RO}_{entry}]$, and for all $x \in Vars_M$, $\mathsf{ro}_{entry}(init(body))(x) \sqsupseteq 1$ implies $x \in \vec{y}$.

This is a true generalisation of liveness, in that each read-once solution ro determines a liveness solution $\mathsf{lv}^{\mathsf{ro}}$.

Proposition 4.2 *For a solution* ro *for* M *define* $\mathsf{lv}^{\mathsf{ro}}_{entry}$ *and* $\mathsf{lv}^{\mathsf{ro}}_{exit}$ *pointwise by*

$$\begin{aligned} \mathsf{lv}^{\mathsf{ro}}_{exit}(\ell) &= \{x \in Var_M \mid \mathsf{ro}_{exit}(\ell)(x) \sqsupseteq 1\} \\ \mathsf{lv}^{\mathsf{ro}}_{entry}(\ell) &= \{x \in Var_M \mid \mathsf{ro}_{entry}(\ell)(x) \sqsupseteq 1\}. \end{aligned}$$

Then $\mathsf{lv}^{\mathsf{ro}} = (\mathsf{lv}^{\mathsf{ro}}_{entry}, \mathsf{lv}^{\mathsf{ro}}_{exit})$ *is a liveness solution for* M.

Read-once usage of a variable x can be read off from the ro_{exit} component of a solution in the same way as usefulness can be read off from the lv_{exit} component of a liveness solution.

Definition 4.3 A variable $x \in \mathit{Vars}_M$ is *read-once for solution* ro if

(i) for all $\ell \in \mathbf{Lab}_M$, $x \in \mathit{kill}(\ell)$ implies $\mathsf{ro}_{exit}(\ell)(x) = 1$,

(ii) $\mathsf{ro}_{entry}(\mathit{init}(\mathit{decl}))(x) \neq \top$, and

(iii) $x \in \{y_1, \ldots, y_n\}$ iff $\mathsf{ro}_{entry}(\mathit{init}(\mathit{decl}))(x) = 1$.

A variable x is *read-once* if there is some solution ro for M for which x is read-once.

To show the analysis sound it is enough to follow [8] and give an extended dynamic semantics which marks read-once variables as being unavailable after the first read access.

4.2 Linear usage typing

We formalise the corresponding functional analysis as a linear type system [13]. To do this we refine the context into a purely linear and an intuitionistic component, and also split the set of variables into two syntactic categories: *linear variables* $a, b, \ldots \in \mathit{LVars}$ and intuitionistic variables $r, t, \ldots \in \mathit{IVars}$. This is similar to some other linear type systems [7,33]. However, there are certain differences to these systems, as we aim to track usage of registers rather than the values within them.

As our set of constants is unrefined, we use a single type, $\circ$. Contexts are partial functions from variables to types, and can always be split into the form $\Theta; \Gamma$ where $\mathit{dom}\ \Theta \subseteq \mathit{IVars}$ and $\mathit{dom}\ \Gamma \subseteq \mathit{LVars}$. We use $-$ to denote empty context components, and stipulate that juxtaposition $\Gamma_1\Gamma_2$ implicitly requires the domains of Γ_1 and Γ_2 to be disjoint. The type system is given by the following rules.

$$\text{T-CONST}\frac{}{\Theta; - \vdash c : \circ} \qquad \text{T-IVAR}\ \frac{}{r : \circ, \Theta; - \vdash r : \circ} \qquad \text{T-LVAR}\ \frac{}{\Theta; a : \circ \vdash a : \circ}$$

$$\text{T-PRIM}\ \frac{\forall i. \Theta; \Gamma_i \vdash x_i : \circ}{\Theta; \Gamma_1 \ldots \Gamma_n \vdash p(x_1, \ldots, x_n) : \circ} \qquad \text{T-RES}\ \frac{\Theta; \Gamma \vdash x : \circ}{\Theta; \Gamma \vdash [x]^\ell : \circ}$$

$$\text{T-CALL}\ \frac{\forall i.\ \Theta; \Gamma_i \vdash x_i : \circ}{\Theta; \Gamma_1 \ldots \Gamma_n \vdash [f(x_1, \ldots, x_n)]^\ell : \circ}$$

$$\text{T-ILET}\ \frac{\Theta; \Gamma_1 \vdash \mathit{expr} : \circ \quad r : \circ, \Theta; \Gamma_2 \vdash \mathit{term} : \circ}{\Theta; \Gamma_1\Gamma_2 \vdash \texttt{let}\ [r = \mathit{expr}]^\ell\ \texttt{in}\ \mathit{term} : \circ}$$

$$\text{T-LLET}\ \frac{\Theta; \Gamma_1 \vdash \mathit{expr} : \circ \quad \Theta; \Gamma_2, a : \circ \vdash \mathit{term} : \circ}{\Theta; \Gamma_1\Gamma_2 \vdash \texttt{let}\ [a = \mathit{expr}]^\ell\ \texttt{in}\ \mathit{term} : \circ}$$

$$\text{T-IF}\ \frac{\Theta; \Gamma_1 \vdash \mathit{expr} : \circ \quad \Theta; \Gamma_2 \vdash [\mathit{result}_1]^{\ell_1} : \circ \quad \Theta; \Gamma_2 \vdash [\mathit{result}_2]^{\ell_2} : \circ}{\Theta; \Gamma_1\Gamma_2 \vdash \texttt{if}\ [\mathit{expr}]^\ell\ \texttt{then}\ [\mathit{result}_1]^{\ell_1}\ \texttt{else}\ [\mathit{result}_2]^{\ell_2} : \circ}$$

$$\text{T-FBLK}\ \frac{[x_1 : \circ, \ldots, x_n : \circ] \vdash \mathit{term} : \circ \quad \langle \vdash \mathit{funblock} \rangle}{\vdash f(x_1, \ldots, x_n) = \mathit{term}\ \langle \texttt{and}\ \mathit{funblock} \rangle}$$

$$\text{T-MBODY}\ \frac{\vdash \mathit{funblock} \quad \Theta; \Gamma \vdash \mathit{term} : \circ}{\Theta; \Gamma \vdash \texttt{fun}\ \mathit{funblock}\ \texttt{in}\ \mathit{term}\ \texttt{end}\ : \circ}$$

$$\text{T-MDECL}\ \frac{[y_1 : \circ, \ldots, y_n : \circ] \vdash \mathit{body} : \circ}{\vdash \texttt{method}\ M(y_1, \ldots, y_n) = \mathit{body} : \circ}$$

Unlike some other systems, although a linear variable may be not be copied, its contents can be *moved* to an intuitionistic variable, as in

```
let a = 5 in let r = a in let s = r + r in s.
```

The bodies of local functions are typed in the context given by their formal parameters. At call sites, the context in which a function is invoked may contain additional intuitionistic variables (which are then discarded), but all linear variables must be accessed. Read as a type system for non-restricted functional programs, we thus admit a function declared by `fun f(r,a,t) = a` to be called by `...in f(b,s,s)`, but not by `...in f(b,b,s)`. For methods in GNF the syntactic restrictions of course ensure that any call has the form `...in f(r,a,t)`, and the correspondence to dataflow solutions indeed holds only for programs in GNF.

4.3 From linear typing to read-once information

Usage information for variables can be read off from the judgements associated to labelled expressions and then related to the dataflow equations.

Definition 4.4 For $\ell \in \mathbf{Lab}_M$ and derivation $\mathcal{C}$ of $\vdash \mathit{decl} : \circ$, we let $\mathcal{C}_\ell$ denote the sub-derivation of $\mathcal{C}$ for the phrase labelled ℓ.

Lemma 4.5 *Let* $\Theta; \Gamma \vdash\colon \mathit{expr} : \circ$ *and* $a \in \mathit{LVars}$. *Then*

$$\mathit{use}(\mathit{expr})(a) = \begin{cases} 1 & \text{if } a \in \mathit{dom}\ \Gamma \\ 0 & \text{if } a \notin \mathit{dom}\ \Gamma. \end{cases}$$

Proof. Induction on the derivation for $\Theta; \Gamma \vdash\colon \mathit{expr} : \circ$. □

We use this to show how each typing derivation determines a solution for equations (4) and (5), restricting our attention first to variables from *LVars*.

Definition 4.6 Let $\mathit{decl} \equiv \texttt{method}\ M(\vec{\mathrm{y}})\ = \mathit{body}$ and let $\mathcal{C}$ be a derivation for $\vdash \mathit{decl} : \circ$. For all $\ell \in \mathbf{Lab}_M$ and $a \in \mathit{LVars}_M$ define $f^{\mathcal{C}} = (f^{\mathcal{C}}_{\mathit{entry}}, f^{\mathcal{C}}_{\mathit{exit}})$ by

$$f^{\mathcal{C}}_{\mathit{exit}}(\ell)(a) = \begin{cases} 0 & \text{if } \ell \in \mathit{final}(M) \\ \sqcup_{(\ell', \ell) \in \mathit{rflow}(M)} f^{\mathcal{C}}_{\mathit{entry}}(\ell')(a) & \text{otherwise} \end{cases}$$

$$f^{\mathcal{C}}_{\mathit{entry}}(\ell)(a) = \begin{cases} 1 & \text{if } a \in \mathit{dom}\ \Delta_\ell \\ 0 & \text{if } a \notin \mathit{dom}\ \Delta_\ell \end{cases}$$

where Δ_ℓ denotes the linear component of the final sequent of $\mathcal{C}_\ell$.

Proposition 4.7 *If decl is in GNF then $f^{\mathcal{C}}$ as defined above fulfils equations* (4) *and* (5) *for all $a \in LVars_M$.*

Proof. Satisfaction of equation (4) follows directly from the definition of $f^{\mathcal{C}}_{exit}$, while the proof of equation (5) depends on whether $a \in kill(\ell)$ and the form of the phrase labelled by ℓ. □

We can extend $f^{\mathcal{C}}$ to a solution for M by combining it with any solution for the non-linear variables.

Theorem 4.8 *Let $\mathcal{C}$ be a derivation for $\vdash decl : \circ$, decl in GNF and $a \in LVars_M$. Then a is read-once.*

Proof. We have to show that there is a solution $\mathsf{ro} = (\mathsf{ro}_{entry}, \mathsf{ro}_{exit})$ for M such that a is read-once for ro. For the minimal solution $f^{min} = (f^{min}_{entry}, f^{min}_{exit})$ for M, we define ro by

$$\mathsf{ro}_{exit}(\ell)(x) = \begin{cases} f^{min}_{exit}(\ell)(x) & \text{if } x \in IVars_M \\ f^{\mathcal{C}}_{exit}(\ell)(x) & \text{if } x \in LVars_M \end{cases}$$

$$\mathsf{ro}_{entry}(\ell)(x) = \begin{cases} f^{min}_{entry}(\ell)(x) & \text{if } x \in IVars_M \\ f^{\mathcal{C}}_{entry}(\ell)(x) & \text{if } x \in LVars_M \end{cases}$$

Then the fact that a is read-once for ro follows by analysis of $\mathcal{C}_\ell$. □

4.4 From read-once to linear typing

The reverse result is that we can globally replace intuitionistic variables r which have been identified to be read-once in typing derivations by fresh linear variables. More precisely, this requires that r is not a "spurious" function argument, in that it is in fact mentioned in the minimal parameter set generated under Theorem 3.3 from the liveness solution induced by ro. The proof proceeds by detailing the correspondence between the lattice of usage values and certain properties of typing derivations.

Theorem 4.9 *Let decl be in GNF, $\vdash decl : \circ$ and ro be a solution for M. If r is read-once for ro and is not spurious, then $\vdash decl[a/r] : \circ$*

5 Discussion

In building a PCC framework with Java classfiles as the transport format, the natural question is: why not just use Java bytecode as the base language? The results presented here give the answer: Grail *is* Java bytecode, but with a stern discipline over the flow of control and data that makes it efficient and straightforward to analyze.

Readers familiar with the history of compilers for functional languages will have already recognised many of the techniques used in Grail. These include ideas from A-normal form and CPS form, types in compilation, and typed

target languages like HOAL and TAL [2,11,12,24,34]. We have taken particular inspiration from λ-JVM, a functional language for expressing general JVM programs [17]. The novelty of Grail, by comparison with these other schemes, lies in the fact that it is strict enough to support a reversible translation to bytecode which preserves execution costs. Indeed, the fact that the Java virtual machine does have enough typing and structure to support all this might be taken as evidence of the influence earlier research has had in selling the benefits of decent languages at even the lowest level.

Current work in MRG is focused on equipping Grail with a specialised logic for reasoning about resource usage. At the moment our prototype infrastructure encodes a resource-aware operational semantics for Grail into Isabelle, and proofs require the full support of its inference engine [28]. This is a useful testbed, but is impractical for wider use. We are therefore developing a suitable Hoare logic, with strong auxiliary variables and elements of separation logic for describing heap usage [29,30].

Working upwards, we are considering a more relaxed version of functional Grail for use in proof generation. Features like nested declarations and full α-conversion could give more "elbow room" during reasoning, even when final assertions are about strict Grail code.

A separate project is to bring arbitrary JVM code under Grail's discipline. Leroy reports success in transforming substantial Java libraries into his constrained form of JavaCard, with no significant change in code size [18]. We are investigating how far this applies to the tighter strictures of Grail.

The software described in this paper is available from the MRG website [25]. This includes the camelot compiler as well as an assembler gdf and disassembler gf to convert between Grail text and Java binaries [22].

Acknowledgements

This work was performed as part of the *Mobile Resource Guarantees* project, funded by the European Commission under the Fifth Framework's proactive initiative on Global Computing, IST-2001-33149. In addition, Ian Stark is funded by an EPSRC Advanced Research Fellowship in *Mathematical Models for Concurrent and Mobile Computation*, GR/R76950/01. We would like to thank all MRG members for the numerous discussions on Grail and Nicholas Wolverson for his help with the implementation of the compilers.

References

[1] Amadio, R., *Max-plus quasi-interpretations*, in: *Typed Lambda Calculi and Applications: Proceedings of TLCA 2003*, Lecture Notes in Computer Science (2003).

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

[3] Appel, A. W., *SSA is functional programming*, ACM SIGPLAN Notices **33** (1998), pp. 17–20.

[4] Appel, A. W., *Foundational proof-carrying code*, in: *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science* (2001), pp. 247–258.

[5] Appel, A. W. and A. P. Felty, *A semantic model of types and machine instructions for proof-carrying code*, in: *Conference Record of POPL '00: 27th ACM Symposium on Principles of Programming Languages* (2000), pp. 243–253.

[6] Aspinall, D. and M. Hofmann, *Another type system for in-place update*, in: *Programming Languages and Systems: Proceedings of the 11th European Symposium on Programming, ESOP 2002*, Lecture Notes in Computer Science **2305** (2002), pp. 36–52.

[7] Barber, A., *Dual intuitionistic linear logic*, Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Science, University of Edinburgh (1996).

[8] Beringer, L., "Asynchronous Queue Machines with Explicit Forwarding," Ph.D. thesis, University of Edinburgh (2002).

[9] Beringer, L., *Cost model*, Laboratory for Foundations of Computer Science, University of Edinburgh (2002), `http://www.lfcs.ed.ac.uk/mrg/publications`.

[10] ECMA International, *Common language infrastructure (CLI), 2nd edition*, Standard ECMA-335 (2002), `http://www.ecma-international.org`.

[11] Flanagan, C., A. Sabry, B. F. Duba and M. Felleisen, *The essence of compiling with continuations*, in: *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 28(6) (1993), pp. 237–247.

[12] Flanagan, C., A. Sabry, B. F. Duba and M. Felleisen, *Retrospective on "The essence of compiling with continuations"*, in: *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*, ACM Press, 2003 .

[13] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **46** (1986), pp. 1–102.

[14] Hofmann, M., *A type system for bounded space and functional in-place update*, Nordic Journal of Computing **7** (2000), pp. 258–289.

[15] Hofmann, M. and S. Jost, *Static prediction of heap space usage for first-order functional programs*, in: *Conference Record of POPL '03: 30th ACM Symposium on Principles of Programming Languages*, 2003, pp. 185–197.

[16] Konečný, M., *Typing with conditions and guarantees in LFPL*, in: *Types for Proofs and Programs: Proceedings of the International Workshop TYPES 2002*, Lecture Notes in Computer Science **2646** (2002), pp. 182–199.

[17] League, C., V. Trifonov and Z. Shao, *Functional Java bytecode*, in: *Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics*, Workshop on Intermediate Representation Engineering for the Java Virtual Machine (2001).
URL `http://flint.cs.yale.edu/flint/publications/lamjvm.html`

[18] Leroy, X., *Bytecode verification for Java smart cards*, Software Practice & Experience **32** (2002), pp. 319–340.

[19] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," The Java Series, Addison-Wesley, 1997.

[20] MacKenzie, K., *Grail: A functional intermediate language for resource-bounded computation, version 1.2*, Laboratory for Foundations of Computer Science, University of Edinburgh (2002), `http://www.lfcs.ed.ac.uk/mrg/publications/Grail-manual.pdf`.

[21] MacKenzie, K., *JVML and .NET*, Laboratory for Foundations of Computer Science, University of Edinburgh (2002), `http://www.lfcs.ed.ac.uk/mrg/publications/`.

[22] MacKenzie, K., *From Camelot to Grail: Compiling a high-level language*, Laboratory for Foundations of Computer Science, University of Edinburgh (2003), `http://www.lfcs.ed.ac.uk/mrg/publications`.

[23] Microsoft, *Overview of the .NET framework*, in: *.NET Framework Developer's Guide*, `http://msdn.microsoft.com` .

[24] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 527–568.

[25] MRG, *Mobile resource guarantees* (2002–2004), `http://www.lfcs.ed.ac.uk/mrg`.

[26] Necula, G. C., *Proof-carrying code*, in: *Conference Record of POPL '97: 24th ACM Symposium on Principles of Programming Languages* (1997), pp. 106–119.

[27] Nielson, F., H. R. Nielson and C. Hankin, "Principles of Program Analysis," Springer, 1999.

[28] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL — A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science **2283**, Springer, 2002.

[29] Oheimb, D. v. and T. Nipkow, *Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited*, in: *Formal Methods Europe 2002*, Lecture Notes in Computer Science **2391** (2002), pp. 89–105.

[30] Reynolds, J., *Separation logic: A logic for shared mutable data structures*, in: *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science* (2002), pp. 55–74.

[31] Stevenson, R. L., "Strange Case of Dr. Jekyll and Mr. Hyde," Longmans, Green, London, 1886.

[32] Sun Microsystems, "Java Card 2.2 Platform Specification," (2003), available online at `http://java.sun.com/products/javacard/specs.html`.

[33] Turner, D. N. and P. Wadler, *Operational interpretations of linear logic*, Theoretical Computer Science **227** (1999), pp. 231–248.

[34] Wand, M., *Correctness of procedure representations in higher-order assembly language*, in: *Mathematical Foundations of Programming Semantics '91: Proceedings of the 7th International Conference*, number 598 in Lecture Notes in Computer Science (1992), pp. 294–311.

[35] Wolverson, N., *Optimisation and resource bounds in Camelot compilation*, Laboratory for Foundations of Computer Science, University of Edinburgh (2003), `http://homepages.inf.ed.ac.uk/s9904010/camelot`.

A Complete Grail program example

Here is a complete program for calculating Fibonacci numbers requested on the command line and printing them to standard output, all coded in Grail. A sample session with it is as follows:

```
$ gdf Fib.gr
Compiled Fib.gr
$ java Fib 5 3 24
fib(5) = 5
fib(3) = 2
fib(24) = 46368
$
```

For a detailed Grail grammar, see [20].

```
class Fib {

    // Calculate the Fibonacci number, tracking two at once for speed
    method static int fib (int n) =
    let val a = 0
        val b = 1
        fun loop (int a, int b, int n) =
            let val b = add a b
                val a = sub b a
                val n = sub n 1
            in
                test(a,b,n)
            end
        fun test (int a, int b, int n) =
            if n<=1 then b else loop(a,b,n)
```

```
    in
        test(a,b,n)
    end

    // Main method; scan arguments, convert to integers, act on each
    method public static void main (java.lang.String[] args) =
    let
        val j = 0
        val n = 0

        fun test (java.lang.String[] args, int j) =
        let val l = length args
        in
            if j >= l then () else print(args,j)
        end

        fun print (java.lang.String[] args, int j) =
        let
            val s = get args j
            val n = invokestatic
                    <int java.lang.Integer.parseInt(java.lang.String)> (s)
            val m = invokestatic <int Fib.fib(int)> (n)
            val () = invokestatic <void Fib.print(int,int)> (n,m)
            val j = add j 1
        in
            test(args,j)
        end
    in
        test(args,j)
    end

    // Output method; take two integers and print a message about them
    method public static void print (int n, int m) =
    let
        val o = getstatic <java.io.PrintStream java.lang.System.out>
        val () = invokevirtual o
                 <void java.io.PrintStream.print(java.lang.String)> ("fib(")
        val () = invokevirtual o <void java.io.PrintStream.print(int)> (n)
        val () = invokevirtual o
                 <void java.io.PrintStream.print(java.lang.String)> (")␣=␣")
    in
         invokevirtual o <void java.io.PrintStream.println(int)> (m)
    end
}
```