Automatic Certification of Heap Consumption

Lennart Beringer[1], Martin Hofmann[2], Alberto Momigliano[1], and Olha Shkaravska[2]

[1] Laboratory for Foundations of Computer Science, The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ; {lenb,amomigl1}@inf.ed.ac.uk
[2] Institut für Informatik, Ludwig-Maximilians-Universität München, Oettingenstraße 67, 80538 München; {mhofmann,shkaravska}@tcs.ifi.uni-muenchen.de

Abstract. We present a program logic for verifying the heap consumption of low-level programs. The proof rules employ a uniform assertion format and have been derived from a general purpose program logic [1]. In a proof-carrying code scenario, the inference of invariants is delegated to the code provider, who employs a certifying compiler that generates a certificate from program annotations and analysis. The granularity of the proof rules matches that of the linear type system presented in [6], which enables us to perform verification by replaying typing derivations in a theorem prover, given the specifications of individual methods. The resulting verification conditions are of limited complexity, and are automatically discharged. We also outline a proof system that relaxes the linearity restrictions and relates to the type system of usage aspects presented in [2].

1 Introduction

Validating the resource consumption of a program obtained from an unknown or untrustworthy code producer is an important task of any security architecture targeting devices with limited resources. The Mobile Resource Guarantees (MRG) project [17] is developing Proof-Carrying Code (PCC) technology [14] to endow mobile code with certificates of bounded resource consumption that can be validated automatically. These certificates are generated by a compiler which, in addition to translating high-level programs into machine code, derives formal proofs based on programmer annotations and program analysis. The foundation of the validation process is a program logic that is sufficiently powerful to formulate expressive certificates. As the logic and the certificate checker are trusted components from the point of view of the program recipient, soundness of the logic with respect to an operational model of the target architecture is crucial, and should ideally be present in a machine-checkable form. In [1], we presented our general-purpose logic, including proofs of soundness and completeness, the latter relative to the ambient logic HOL. The development is completely backed up by an implementation in Isabelle/HOL, building upon, and extending, earlier work on formalised program logics by Kleymann, Nipkow and others [8, 15]. In this paper, we use this logic to justify more specialised logics for the resource *heap consumption*. We develop proof rules that allow the code producer to certify the heap consumption of a low-level program in such a way that the recipient can validate the memory behaviour prior to execution. Judgements in these heap logics arise from the base logic by restricting assertions to syntactically uniform representations and formally deriving proof rules

F. Baader and A. Voronkov (Eds.): LPAR 2004, LNAI 3452, pp. 347–362, 2005.

in the theorem prover. The assertion formats are motivated by, and closely related to, typing judgements used in a certifying compiler for inferring the memory requirements of programs at source level.

Our approach to deriving proof rules for restricted assertion formats from the base logic achieves several goals: firstly, soundness of the heap logics with respect to the operational model is obtained from the soundness of the base logic. Secondly, a method for certificate generation is achieved: the type systems infers invariants (in our case: method specifications) for the low-level code based on the strategy used for compiling high-level programs. Thirdly, a strategy is obtained that allows the program recipient to verify the validity of a proof automatically: the proof rules are set up in such a way that methods can be proved in a largely syntax-directed way, with side conditions that are of low complexity. The granularity of proof rules matches that of the type systems: sequences of low-level instructions that originate from a high-level language construct are combined in a single proof rule. Thus, the consumer-side verification can follow a validation tactic that essentially replays typing judgements, where the compiler-generated invariants eliminate the need to perform complex proof search.

In the main part of this paper, we outline this certification strategy for an (affinely) linear assertion format that interprets the type system of Hofmann and Jost [6]. Continuing our work on formalisation, the derivation of the proof system from the base logic has been implemented in Isabelle/HOL, as has the verification tactic at recipient side. However, in order to demonstrate that our approach is more widely applicable, we also outline an extension that considers assertions corresponding to the more powerful type system of [2]. Here, the linearity requirements are relaxed by distinguishing between three usage disciplines a program may obey with respect to a data structure. While the formalisation of the corresponding proof system for derived assertions in a theorem prover is under way, the syntax-directedness and the computational simplicity of the side conditions again make an automatic verification by the recipient appear feasible.

2 Components of the MRG Architecture

In this section we summarise MRG's PCC architecture. We start by introducing our representation of low-level code, the Grail language, and the program logic that forms the foundation of the certification. We then move to the high-level language, Camelot, and discuss the compilation of programs into Grail, with particular emphasis on memory management. Finally, we outline the static analysis of memory consumption that will be the basis of the proof rules in the following section. For details, see [1, 4, 6, 12].

Syntax and Semantics of Grail The target of MRG's compilation, and the language to which certificates refer, is a restricted form of Java bytecode, Grail [4]. This language retains the object and method structure of bytecode, but represents method bodies as sets of mutually tail-recursive first-order functions. The syntax comprises instructions for object creation and manipulation, method invocation and primitive operations such as integer arithmetic, as well as let-bindings to combine program fragments. The main characteristic of Grail is its dual identity: its (impure) call-by-value functional semantics coincides with an imperative interpretation of the expansion of Grail programs into

the Java Virtual Machine Language, provided that some mild syntactic conditions are met. In particular, we require that actual arguments in function calls coincide syntactically with the formal parameters of the function definitions. In [4] we showed that this discipline, together with Administrative-Normal-Form (ANF)-style normalisation of let-expressions, allows function calls to be interpreted as immediate jump instructions, and admits the definition of a code transformation that is the exact reversal of the expansion of Grail expressions into JVML. The formal syntax of expressions

$$e \in \mathit{expr} ::= \mathtt{null} \mid \mathtt{int}\ i \mid \mathtt{var}\ x \mid \mathtt{prim}\ \mathit{op}\ x\ x \mid \mathtt{new}\ C\ [\overline{t_i := x_i}] \mid x.t \mid x.t{:=}x \mid C.t \mid C.t{:=}x \mid \mathtt{let}\ x{=}e\ \mathtt{in}\ e \mid e\ ;\ e \mid \mathtt{if}\ x\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \mathtt{call}\ f \mid C.M(\overline{a})$$

$$a \in \mathit{args} ::= \mathtt{var}\ x \mid \mathsf{null} \mid i$$

is defined over mutually disjoint sets of method names, class names, function names (i.e. labels of basic blocks), (static) field names and variables, ranged over by M, C, f, t, and x, respectively. In the grammar, i ranges over integers and op denotes a primitive operation of type $\mathcal{V} \Rightarrow \mathcal{V} \Rightarrow \mathcal{V}$ such as an arithmetic or a comparison operator. Here $\mathcal{V}$ is the semantic category of values (ranged over by v), comprising integers, references r, and the special symbol $\perp$, which stands for the absence of a value. Heap references are either null or of the form $\mathsf{Ref}\ l$ where $l \in \mathcal{L}$ is a location.

Expressions represent basic blocks and are built from operators, constants, and previously computed values (names). They correspond to primitive sequences of bytecode instructions which may, as a side effect, alter the heap. For example, $x.t$ and $x.t{:=}y$ represent (non-static) field access instructions, while $C.t$ and $C.t{:=}y$ denote their static counterparts. The binding $\mathtt{let}\ x{=}e_1\ \mathtt{in}\ e_2$ is used if the evaluation of e_1 returns an integer or reference value on top of the JVM stack while $e_1\ ;\ e_2$ represents non-binding composition, used for example if e_1 is a field update. Object creation includes the initialisation of the object fields according to the argument list. Function calls follow the Grail calling convention (i.e. correspond to immediate jumps) and do not carry arguments. The instruction $C.M(\overline{a})$ represents static method invocation. While formal parameters of method invocations are variables, actual arguments can be variables, integer constants or null. Although a formal type and class system may be imposed on Grail programs, our program logic abstracts from these restrictions. We assume that all method declarations employ distinct names for identifying inner basic blocks.

A program is represented by a table *Ftable* mapping each function identifier to an expression and a list of formal arguments, and a table *Mtable* associating method parameters and the name of the initial basic block to class names and method identifiers. The formal basis of the program logic is an operational semantics that is expressed as a big-step evaluation relation $E \vdash h, e \Downarrow h', v$. For expression e, such a judgement relates an (initial) variable environment $E \in \mathcal{E}$ and an initial heap $h \in \mathcal{H}$ to a final heap $h' \in \mathcal{H}$ and the result value $v \in \mathcal{V}$. Heaps are finite maps from locations and field names to values, while environments are modelled as total maps from variable names to values. The rules for defining the operational semantics are omitted, but are available in [1].

The Core Program Logic In our program logic [1], judgements take the form $G \rhd e : P$ where e is a Grail expression, G a context used for storing verification assumptions for recursive methods and functions, and P an assertion. Deviating from both Hoare-style

and VDM-style logic [7], we combine pre- and post-conditions into single assertions: P is a predicate (in the meta-logic HOL) over the semantic components, and relates the initial and final heaps, the initial environment, and the result value: $P : \mathcal{E} \to \mathcal{H} \to \mathcal{H} \to \mathcal{V} \to \mathcal{B}$, where $\mathcal{B}$ is the set of booleans. For example, the rule for program composition

$$\frac{G \rhd e_1 : P_1 \quad G \rhd e_2 : P_2}{G \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \lambda\, E\ h\ h'\ v.\ \exists\, h_1\ w.\ (P_1\, E\, h\, h_1\, w)\ \wedge\ w \neq \bot\ \wedge\ (P_2\, (E\langle x := w\rangle)\, h_1\, h'\, v)}\ \textsc{Vlet}$$

existentially abstracts the intermediate heap and models the binding of x to the result of evaluating e_1 by interpreting P_2 in the extended environment $E\langle x := w\rangle$. Satisfaction of a specification P by program e is denoted by $\models e : P$ and asserts that $E \vdash h, e \Downarrow h', v$ implies $P\, E\, h\, h'\, v$. In [1] we proved the soundness and (relative) completeness of the program logic with respect to this (partial) interpretation, i.e. the statement $\varnothing \rhd e : P \Longleftrightarrow \models e : P$. Associations between methods and their specifications are collected in a method specification table *MST*. In order to allow the usage of a proof rule for method invocation that includes parameter adaptation, each method specification additionally also abstracts over a list of actual arguments.

Compilation of Camelot Programs The high-level language Camelot is a first-order functional language with ML-style polymorphism and algebraic datatypes [12]. The following example code introduces a data type of integer lists and functions that implement the insertion sort algorithm.

```
type L = !Nil | Cons of int * L
let ins a l = match l with Nil -> Cons(a,Nil)
                         | Cons(x,t)@_ -> if a < x
                                          then Cons(a,Cons(x,t))
                                          else Cons(x, ins a t)
let sort l =  match l with Nil -> Nil
                         | Cons(a,t)@_ -> ins a (sort t)
```

The compiler translates a program into Grail following a whole-program compilation approach with phases such as monomorphisation and let-normalisation. The resulting code contains a class InsSort comprising one (static) method for each Camelot function. For example, the code for insertion sort yields methods InsSort.ins and InsSort.sort whose (slightly pretty-printed) Grail representations are shown next:

```
method InsSort.ins(int a, D l) = call f
f: let b=prim isNull l l in
   if b then let l=null in D.make(a,l)
        else let v3=l.HD in let v2=l.TL in D.free(l) ; let b=prim less a v3 in
             if b then let l=D.make(v3,v2) in D.make(a,l)
                  else let l=InsSort.ins(a,v2) in D.make(v3,l)
method InsSort.sort(D l) = call g
g: let b=prim isNull l l in
   if b then null
        else let v3=l.HD in let v2=l.TL in D.free(l) ;
             let l=InsSort.sort(v2) in InsSort.ins(v3,l)
```

Furthermore, a class D is defined that declares sufficiently many fields for representing values of all declared types (in our case: HD and TL), plus some internal fields (TAG, FLIST, NEXT). The latter are used for discriminating between the various datatype constructors (TAG, only used for datatypes with more than one non-nullary constructor), and for implementing a freelist, i.e. a (non-cyclic) list of D-objects whose initial member is pointed to by the static field FLIST, and whose elements are linked via field NEXT. The declaration of D also contains methods for performing the operations typical of a freelist: objects can be inserted into the freelist using the method free, while the method make allocates a fresh object (method alloc) and initialises its application fields according to its method parameters (method fill). The code for these memory management methods is shown next:

```
method D.free(D nd)          = let f=D.FLIST in nd.NEXT:=f ; D.FLIST:=nd
method D.alloc()             = let f=D.FLIST in let b=prim isNull f f in
                                 if b then new D []
                                      else let t=f.NEXT in D.FLIST:=t ; f
method D.fill(D x,int v,D w) = x.HD:=v ; x.TL:=w ; x
method D.make(int v,D w)     = let x=D.alloc() in D.fill(x,v,w)
```

Notice that of all methods, alloc is the only one that contains the instruction `new`. Fresh memory can thus only be allocated through the memory management interface, and this operation is only performed if the freelist is empty. This discipline is at the heart of our verification: the interpretation of assertions in the derived program logic ensures that all requests from the freelist can be served without executing `new`.

Two further aspects of the compilation are worth mentioning, as they concern programmer annotations in the source program. In each datatype declaration, (at most) one constructor (like `Nil` in our example code) can be equipped with the annotation `!`, thus instructing the compiler to use a heap-free representation. The effect is visible in the compiler output: conditionals corresponding to match statements w.r.t. this constructor discriminate over the condition *isNull* instead of inspecting the content of the field TAG. The second programmer annotation, `@_`, indicates that the corresponding (branch of the) pattern match may be implemented destructively, i.e. the memory cell inhabited by the value against which the match is performed is returned to the freelist after the components have been accessed. The compiler output reflects this by calling method free after de-constructing the cons-cells in methods ins and sort. The compiler verifies that both annotations are used safely: a constructor annotated with `!` must not have arguments, and pattern matching using `@_` is only admitted if the data structure may indeed be destroyed, i.e. it is not used in the continuation of the program.

Inference of Heap Space Consumption In order to analyse the memory requirements of functional programs, Hofmann and Jost [6] introduced a type system that solves the following problem. Given a program e of type, say, `bool list` $\rightarrow$ `bool list`, calculate a (linear) function f such that computing $e(L)$ for some input list L will not require more than $f(|L|)$ additional heap cells, provided that a freelist is available for storing temporarily unused cells. In the context of the Camelot compilation, the result of such an analysis can be used to ensure that the evaluation of $e(L)$ will not perform a call

to `new`, by wrapping the evaluation with code that allocates a freelist of the sufficient length $f(|L|)$ prior to calling e.

The analysis of [6] is formulated using an extended notion of types such that different portions of the input can contribute a different amount to memory consumption. For each (non-heapfree) datatype constructor, a numeric annotation indicates the amount of heap that is required for a single build operation using that constructor. For example, `L(5)` indicates the type of an integer list where each occurrence of `Cons` requires five free memory cells to be available – constructing the list, say, `[97;634;42]` thus requires fifteen additional cells to be available. Judgements in the type system are of the form $\Gamma, n \vdash e : T, m$ where T is the (extended) type of e with respect to the context Γ (which maps free variables of e to extended types), while n and m represent constants that describe memory requirements that are independent of the size of the data structures. The typing rules are defined in such a way that m and the numeric annotations in T are expressed relative to the size of the *output* data structure, while n and the annotations in Γ refer to the size of the *input* data structures. For example, evaluating a program e with typing $x : \mathtt{L(5)},\ \mathtt{4} \vdash e : \mathtt{L(2)}, \mathtt{7}$ in an environment where x is bound to the list `[97;634;42]` requires no more than $4 + (5 * 3)$ cells to be available in the freelist, and leaves a freelist of length (at least) $7 + 2 * |M|$ where M is the output list.

We present next some of the typing rules, which are motivated by this understanding. The rule for constructing a list node, CONS, requires the initial freelist to contain at least as many elements as the final freelist does, plus one cell for representing the value itself, plus the additional k cells specified in the desired return type. In order to construct lists with homogeneous memory behaviour, the type associated to the tail t in the context must also be $\mathtt{L}(k)$.

$$\frac{n \geq 1 + k + m}{\Gamma, h : \mathtt{int}, t : \mathtt{L}(k), n \vdash \mathtt{Cons}(h,t) : \mathtt{L}(k), m}\ \textsc{Cons}$$

$$\frac{\Gamma, n \vdash e_1 : A, m \qquad \Gamma, h : \mathtt{int}, t : \mathtt{L}(k), n + 1 + k \vdash e_2 : A, m}{\Gamma, x : \mathtt{L}(k), n \vdash \mathtt{match}\ x\ \mathtt{with}\ \mathtt{Nil} \Rightarrow e_1 |\ \mathtt{Cons}(h,t)@\_ \Rightarrow e_2 : A, m}\ \textsc{DM}$$

$$\frac{\Gamma, n \vdash e_1 : A, m \qquad \Gamma, h : \mathtt{int}, t : \mathtt{L}(k), n + k \vdash e_2 : A, m}{\Gamma, x : \mathtt{L}(k), n \vdash \mathtt{match}\ x\ \mathtt{with}\ \mathtt{Nil} \Rightarrow e_1 |\ \mathtt{Cons}(h,t) \Rightarrow e_2 : A, m}\ \textsc{M}$$

$$\frac{\Gamma_1, n \vdash e_1 : A, k \quad \Gamma_2, x : A, k \vdash e_2 : B, m}{\Gamma_1, \Gamma_2, n \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B, m}\ \textsc{Let} \qquad \frac{\Gamma, x : A_1, y : A_2, n \vdash e : A, m}{\Gamma, z : A_1 \oplus A_2, n \vdash e[z/x, z/y] : A, m}\ \textsc{Share}$$

The effect of a pattern match on the freelist depends on whether the match is performed destructively or not. In both cases, the branch executed in the case of an empty list has exactly the same memory behaviour as the composite expression. In the case of a non-empty list, the freelist available for the continuation grows at least by the amount k "stored in" the list node that is taken apart. In case of a destructive match, the cell inhabited by this node becomes available as well, which explains the additional $+1$ in rule DM that is not present in the rule for a non-destructive match, M. The rule for program composition, LET, reflects the above-mentioned interpretation of typing judgements, in particular the fact that the result type and the right-hand-side annotation m of a judgement refer to the size of the result of the computation, as the typing of the composite

statement may be obtained compositionally from the typing of the sub-terms, glued together by the freelist-constant k, and the type A, that occur in both hypothesis. Finally, the rule SHARE allows us to split resources between different variables representing the same data structure – operation $\oplus$ recursively descends through the type structure and adds the annotations in the leafs. As this contraction results in the data structure being aliased, the soundness of this rule relies on the semantic condition of *benign sharing*: whenever a cell l is returned to the freelist during a destructive pattern match, the program continuation will not access l through any aliasing access path. Various static approximations to this conditions can be considered. Of these, imposing a *linear* typing discipline (i.e. considering the type system without rule SHARE, and interpreting the context split in rule LET to be a disjoint partitioning) is rather restrictive, but only moderately complex to implement, and is therefore chosen in the formal interpretation of assertions in the next section. However, as many programs cannot be typed under such a discipline, it is desirable to have alternative means at hand. The generalisation of our assertion format in Sect. 6 is a step in that direction, as it corresponds to the more permissive type system of usage aspects presented in [2].

The inference process for the type system consists of two stages. First, a skeleton type derivation is constructed where the numerical annotations $n,k,m,\ldots$ are interpreted as (rational) variables, constrained by side conditions such as the one in rule CONS. These side conditions are collected, and in a second step handed over to a linear programming solver. Any feasible solution to the linear program corresponds to a possible typing derivation. This inference process has been implemented for the language considered in [6] and for Camelot, and scales well even to programs where skeleton derivations contain thousands of variables or constraints.

In the context of certificate generation, the solution inferred by the analysis (if existing) is presented as a signature that contains one (extended) typing for each Camelot function. In the case of our example program, one such signature is

$$\{\texttt{ins} : 1, \texttt{int} \times \mathtt{L}(0) \multimap \mathtt{L}(0), 0,\ \texttt{sort} : 0, \mathtt{L}(0) \multimap \mathtt{L}(0), 0.\}$$

For both functions, this signature asserts that the heap consumption does not depend on the size of the input: `ins` consumes one heap cell, while `sort` executes in-place: the cell that is required in the call to `ins` has previously been gained in the pattern match.

3 Format and Interpretation of Assertions

In this section we introduce a class of assertions that interpret judgements of the high-level type system in the program logic. These assertions have a uniform syntactic form, and their interpretation expands to a predicate over the semantic components (environment, pre-heap, post-heap and return value), as is required of specifications by the core logic. This syntactic form,

$$[\![U, n, \Gamma \blacktriangleright T, m]\!]$$

comprises components similar to the type system:

$n, m \in \mathbb{N}$ represent the numerical results from the analysis. In the interpretation these numbers will relate to the initial and final length of the freelist, respectively.

Γ is the typing context, a partial map from program variables to extended types.
U (a finite set of program variables) is used to enforce the linear typing discipline.
T indicates the type of an expression e that satisfies the assertion.

In this paper, we only consider the data type of integer list. In the grammar

$$T \in \mathbf{T} ::= \mathbf{1} \mid \mathbf{I} \mid \mathbf{L}(k)$$

the constructors represent respectively the unit (void) type, the integer type, and the type of lists where each occurrence of the `Cons` constructor is equipped with $k \in \mathbb{N}$ additional free heap cells and the `Nil` constructor does not reserve any space. In [3] we consider additional types, for representing e.g. integer trees.

The interpretation of assertions, and the proof rules that will be presented in the following section, are formulated in such a way that the set U is inferred during the verification condition generation, and coincides with the free variables of e. Thus, the restricted context $\Gamma\lfloor_U$ amounts to the minimal context in which an expression e may be typed.

Before giving the semantic definition of an assertion, we introduce some auxiliary predicates. Given a value v of type T and a heap h, the predicate $v,h \models_T R,n$ computes the region R inhabited by v, and the number n of free heap cells associated with it according to the numerical annotations in T.

$$\frac{}{\bot,h \models_{\mathbf{1}} \varnothing,0}\ \textsc{regU} \qquad \frac{}{i,h \models_{\mathbf{I}} \varnothing,0}\ \textsc{regI} \qquad \frac{\mathit{LIST}(n,r,R,h)}{r,h \models_{\mathbf{L}(k)} R,k*n}\ \textsc{regL}$$

In rule REGU, we abuse the earlier notation slightly and let the symbol $\bot$ also denote the canonical value of type $\mathbf{1}$. In rule REGL, the list predicate $\mathit{LIST}(n,r,R,h)$ is satisfied if reference r in heap h points to a (cycle-free) linked list of length n, whose cells inhabit exactly locations R.

$$\frac{}{\mathit{LIST}(0,\mathsf{null},\varnothing,h)}\ \textsc{Nil} \qquad \frac{h(l).\mathsf{HD} = i \quad h(l).\mathsf{TL} = r \quad \mathit{LIST}(n,r,R,h)}{\mathit{LIST}(n+1,\mathsf{Ref}\,l,R \uplus l,h)}\ \textsc{Cons}$$

The definition directly reflects the layout of data values implemented by the Camelot compiler – the disjoint sum notation $\uplus$ indicates the implicit side condition $\{l\} \cap R = \varnothing$.

Next, we define a predicate $\Gamma,U \models^E_h R,n$ that computes the amount n of free heap associated to the variables in $\Gamma\lfloor_U$ and the heap region R inhabited by the corresponding data structures.

$$\frac{}{\Gamma,\varnothing \models^E_h \varnothing,0}\ \textsc{HeapE} \qquad \frac{E\langle x\rangle,h \models_{\Gamma(x)} R_1,n \quad \Gamma,U \models^E_h R_2,m}{\Gamma,U \uplus x \models^E_h R_1 \uplus R_2,n+m}\ \textsc{HeapV}$$

Rule HEAPV, in combination with the above definition of the datatype representation predicates, enforces a strict separation both between and within data structures. We will relax some of these separation conditions in Sect. 6.

A further auxiliary predicate, $\mathit{freelist}(h,F,N)$, is defined by $\mathit{FL}(N,h\langle\mathsf{D.FLIST}\rangle,F,h)$ and expresses the fact that in heap h, the static field D.FLIST points to a (non-cyclic) list of length N, where the cells collectively inhabit locations F and are linked via field NEXT. The predicate $\mathit{FL}(\_,\_,\_,\_)$ is defined analogously to the predicate $\mathit{LIST}(\_,\_,\_,\_)$.

Finally, the predicate $\mathit{footprint}(R,h,h') \equiv \forall l \in \mathit{dom}\, h \setminus R.\ h(l) = h'(l)$ bounds the set of locations on which two heaps may differ.

The interpretation $[\![U,n,\Gamma \blacktriangleright T,m]\!]$ is now defined by

$$[\![U,n,\Gamma \blacktriangleright T,m]\!] \equiv\ \forall\, q\, F\, R.$$
$$\left(\begin{array}{l} \exists\, N\, K.\ \mathit{freelist}(h,F,N) \wedge \\ \qquad \Gamma,U \models^{E}_{h} R,K \wedge \\ \qquad R \cap F = \varnothing \wedge \\ \qquad n+K+q \leq N \end{array}\right) \longrightarrow \left(\begin{array}{l} \exists\, Q\, S\, M\, H.\ v,h' \models_T Q,S \wedge \mathit{freelist}(h',H,M) \wedge \\ \qquad Q \cap H = \varnothing \wedge (Q \cup H) \subseteq (R \cup F) \wedge \\ \qquad \mathit{footprint}(F \cup R,h,h') \wedge \\ \qquad m+S+q \leq M \wedge \mathit{dom}\, h = \mathit{dom}\, h' \end{array}\right)$$

where the free variables E, h, h' and v are implicitly abstracted over. A judgement $G \rhd e : [\![U,n,\Gamma \blacktriangleright T,m]\!]$ thus asserts that, whenever

- the initial heap h contains a freelist of length N, inhabiting locations F,
- the region R inhabited by the data structures $\Gamma\lfloor_U$ is disjoint from F, and
- the length N of the freelist is at least the amount K of heap owned by $\Gamma\lfloor_U$, plus the additionally required size n and some constant q,

and the evaluation of e terminates, then there are M and S and regions Q and G s. t.

- the result v (according to the type T) inhabits region Q (in the final heap h') and contributes S cells to the (final) freelist,
- the final heap contains a freelist of length M inhabiting region G,
- the result and the final freelist do not overlap,
- both G and Q are contained in the initial freelist region F, extended by the locations reachable (in the initial heap) by the variables in $\Gamma\lfloor_U$,
- locations that are neither part of the freelist nor reachable from variables from $\Gamma\lfloor_U$ remain unchanged, i.e. $F \cup R$ is an approximation of the locations touched,
- the final length M of the freelist is at least the amount S contributed by the result, plus the analysis number m and the constant q, and
- no new objects have been allocated.

Thus, data structures represented by variables in $\Gamma\lfloor_U$ are potentially destroyed. Corresponding locations may have been recycled during the evaluation of e, may have been inserted into the freelist, or have become unreachable. In contrast, locations not reachable from variables in $\Gamma\lfloor_U$ remain unchanged.

4 Proof Rules

Having introduced the assertion format, we can derive proof rules for various Grail phrases by unfolding the interpretation. The design of the proof rules was guided by the aim of minimising the complexity of verification conditions that arise from side conditions, and to mirror the high-level typing rules. Indeed, the granularity of the proof rules corresponds to that of the typing system: match statements and constructor applications are verified as single entities, i.e. only the soundness proof of the rules inspects the constituent instructions of the corresponding methods.

We first present the rules for basic syntactic constructs of Grail. There are no proof rules for object creation and (virtual or static) field access instructions, since these operations are only performed inside the memory management methods. The rules for function calls and method invocations are the rules of the base logic.

$$\frac{m \le n}{G \rhd \texttt{null} : [\![\varnothing, n, \Gamma \blacktriangleright \mathbf{L}(k), m]\!]}\ \textsc{Null} \qquad \frac{m \le n}{G \rhd \texttt{int}\ i : [\![\varnothing, n, \Gamma \blacktriangleright \mathbf{I}, m]\!]}\ \textsc{Int}$$

$$\frac{m \le n \quad \Gamma(x) = T}{G \rhd \texttt{var}\ x : [\![\{x\}, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{Var} \qquad \frac{\{x,y\} \subseteq dom\,\Gamma \quad m \le n}{G \rhd \texttt{prim}\ op\ x\ y : [\![\{x,y\}, n, \Gamma \blacktriangleright \mathbf{I}, m]\!]}\ \textsc{Prim}$$

$$\frac{G \rhd e_1 : [\![U_1, n, \Gamma \blacktriangleright \mathbf{1}, m]\!] \quad G \rhd e_2 : [\![U_2, m, \Gamma \blacktriangleright T, k]\!]}{G \rhd e_1\ ;\ e_2 : [\![U_1 \uplus U_2, n, \Gamma \blacktriangleright T, k]\!]}\ \textsc{Comp}$$

$$\frac{G \rhd e_1 : [\![U_1, n, \Gamma \blacktriangleright S, l]\!] \quad G \rhd e_2 : [\![U_2, l, (\Gamma, x : S) \blacktriangleright T, m]\!] \quad S \neq \mathbf{1}}{G \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : [\![U_1 \uplus (U_2 \setminus \{x\}), n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{Let}$$

$$\frac{G \rhd e_1 : [\![U_1, n, \Gamma \blacktriangleright T, m]\!] \quad G \rhd e_2 : [\![U_2, n, \Gamma \blacktriangleright T, m]\!]}{G \rhd \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : [\![U_1 \cup U_2, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{If}$$

$$\frac{(G \cup \{(\texttt{call}\, f, P\}) \rhd Ftable\, f : P}{G \rhd \texttt{call}\, f : P}\ \textsc{Call}$$

$$\frac{\begin{array}{l}(G \cup \{(C.M(\overline{a}), P\}) \rhd \\ \quad Mtable\ C\ M : \lambda\, E\ h\ h'\ v.\ \forall\, E'.\ E = frame\ (params\ C\ M)\ \overline{a}\ E' \longrightarrow P\, E'\, h\, h'\, v\end{array}}{G \rhd C.M(\overline{a}) : P}\ \textsc{InvS}$$

Next, we present the rules for non-destructive and destructive match operations, and for constructor `Cons`. Treating the freelist management operations atomically reflects the fact that the states at intermediate program points of these composite statements do not satisfy formulae of the restricted form – they contain dangling pointers and incompletely built data structures. In rule DMATCH, the additional side condition $x \neq t$ is needed to avoid the insertion of t into the freelist by instruction $\mathsf{D}.\texttt{free}(x)$.

$$\frac{\Gamma(x) = \mathbf{L}(k) \quad h \notin \{x,t\} \quad G \rhd e : [\![U, n+k, (\Gamma, h : \mathbf{I}, t : \mathbf{L}(k)) \blacktriangleright T, m]\!]}{G \rhd \texttt{let}\ h = x.\mathsf{HD}\ \texttt{in let}\ t = x.\mathsf{TL}\ \texttt{in}\ e : [\![(U \setminus \{h,t\}) \uplus x, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{Match}$$

$$\frac{\begin{array}{c}\Gamma(x) = \mathbf{L}(k) \qquad h \notin \{x,t\} \qquad x \neq t \\ G \rhd e : [\![U, n+k+1, (\Gamma, h : \mathbf{I}, t : \mathbf{L}(k)) \blacktriangleright T, m]\!]\end{array}}{\begin{array}{l}G \rhd \texttt{let}\ h = x.\mathsf{HD}\ \texttt{in let}\ t = x.\mathsf{TL}\ \texttt{in}\ \mathsf{D}.\texttt{free}(x)\ ;\ e : \\ \qquad\qquad [\![(U \setminus \{h,t\}) \uplus x, n, \Gamma \blacktriangleright T, m]\!]\end{array}}\ \textsc{DMatch}$$

$$\frac{\Gamma(y) = \mathbf{L}(k) \quad \Gamma(x) = \mathbf{I}}{G \rhd \mathsf{D}.\texttt{make}(x,y) : [\![\{x,y\}, m+k+1, \Gamma \blacktriangleright \mathbf{L}(k), m]\!]}\ \textsc{Make}$$

Finally, we give some structural rules.We will comment on their role in verification condition generation in the next section.

$$\frac{G \rhd e : [\![U, n, \Gamma \blacktriangleright T, m]\!] \quad n \leq n' \quad m' \leq m}{G \rhd e : [\![U, n', \Gamma \blacktriangleright T, m']\!]}\ \textsc{Relax}$$

$$\frac{G \rhd e : [\![V, n, \Gamma \blacktriangleright T, m]\!] \quad V \subseteq U}{G \rhd e : [\![U, n, \Gamma \blacktriangleright T, m]\!]}\ \textsc{Gen} \qquad \frac{G \rhd e : [\![U, n, \Gamma \blacktriangleright T, m]\!]}{G \rhd e : [\![U, n+k, \Gamma \blacktriangleright T, m+k]\!]}\ \textsc{Shift}$$

$$\frac{G \rhd e : [\![U, n, \Gamma \blacktriangleright T, m]\!] \quad \forall\, x \in U.\ \Delta(x) = \Gamma(x)}{G \rhd e : [\![U, n, \Delta \blacktriangleright T, m]\!]}\ \textsc{Ctxt}$$

Theorem 1. *All proof rules presented in this section are derivable in HOL from the core logic.*

The proof rules enforce benign sharing in a way that corresponds to linearity in the type system. The rules COMP and LET combine the U-sets using the disjoint union operator $\uplus$. From the point of view of surrounding code, linearity is also observed in the rules MATCH and DMATCH, despite variable x occurring repeatedly in the program text.

5 Verification

We now return to our example program, insertion sort, and outline the verification process. As was remarked earlier, compiling the Camelot code for insertion sort results in two class declarations: the class D with fields for the representation of data types and the (pre-verified) memory management methods, plus a class InsSort containing the application methods ins and sort. In addition, the compiler generates a certificate that contains the result of the program analysis in a form that can be automatically verified. The certificate contains the method specification table, the definition of a proof context G, and calls to a predefined Isabelle tactic. Before describing the global verification strategy, we first outline how this tactic verifies an individual method body.

For verifying that method body *Mtable C M* satisfies a specification of the restricted form, i.e. that $G \rhd \mathit{Mtable}\ C\ M : [\![U, n, \Gamma \blacktriangleright T, m]\!]$ holds, we have implemented an Isabelle tactic ($\approx$ 150 lines of ML) that starts by applying the GEN rule, then applies the syntax-directed and memory management proof rules discharging the side conditions locally, and finally verifies that the initial side condition of GEN, $V \subseteq U$, holds for the inferred set V. The tactic maintains a stack of open goals that ensures that only ground conditions arise. Inspecting the proof rules shows that apart from numerical comparisons, set inclusions, and context look-ups, no advanced simplification nor decision procedures are required. The tactic is applied with a specification table *MST* that contains entries representing the result of the type analysis. Assertions are formulated from the perspective of the method body, i.e. the chosen variable names are the formal parameters. Method invocations are verified using a variation of INVS that incorporates the effect of rules SHIFT and CTXT, and a notion of variable renaming for assertions that is needed to handle the passing from actual arguments to formal parameters. In our example program, the specification table contains two entries that correspond to

$$\mathit{Ins\_Spec} \equiv [\![\{a, l\}, 1, [a : \mathbf{I}, l : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!]$$
$$\mathit{Sort\_Spec} \equiv [\![\{l\}, 0, [l : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!]$$

Note that we have made no effort to employ efficient data structures and we rely on naive representation of contexts and sets as provided by Isabelle/HOL. However, we have implemented a technique that allows us to verify each function body only once, based on compiler-generated merge point information. For some details see [3].

Global verification is based on the rule

$$\frac{\mathit{goodContext}\ \mathit{MST}\ G \quad \mathit{finite}(G) \quad (C.M(\overline{a}), \mathit{MST}\ M\ \overline{a}) \in G}{\varnothing \rhd C.M(\overline{b}) : \mathit{MST}\ M\ \overline{b}}\ \textsc{VAdaptS}$$

which derives $\varnothing \rhd C.M(\overline{b}) : B$ (notice the empty context), provided the existence of a context G that fulfils property *goodContext* and contains an entry $(C.M(\overline{a}), A)$ where A and B arise by instantiating the method specification table entry for M with the method arguments $\overline{a}$ and $\overline{b}$, respectively. The generated certificate contains the definition of such a context G, consisting of one entry $(C.M(\overline{a}), \mathit{MST}\ M\ \overline{a})$ for each method invocation occurring in the program. In our example program, the context G is given by

$$G \equiv \left\{ \begin{array}{l} (\mathsf{InsSort.ins}(a, v_2),\ \mathit{MST}\ \mathsf{ins}\ [a, v_2]),\ (\mathsf{InsSort.ins}(v_3, l),\ \mathit{MST}\ \mathsf{ins}\ [v_3, l]), \\ (\mathsf{InsSort.sort}(v_2),\ \mathit{MST}\ \mathsf{sort}\ [v_2]) \end{array} \right\}.$$

The definition of *goodContext* (see [1] for details) requires each such entry to satisfy $G \rhd \mathit{Mtable}\ C\ M : \varphi(\mathit{MST}\ M)$, where φ models the passing of method arguments to the formal parameters. As the result of applying φ is of the form $[\![U, n, \Gamma \blacktriangleright T, m]\!]$, discharging the condition of the *goodContext* predicate may be performed by the tactic discussed above. Our verification script verifies first each method body individually, before combining the resulting local correctness statements. We thus obtain correctness of the sort method for arbitrary method arguments

Theorem 2. *We have* $\varnothing \rhd \mathsf{InsSort.sort}(x) : \mathit{MST}\ \mathsf{sort}\ [x]$

for arbitrary x using a strategy that verifies each method body only once, despite the existence of two entries for ins in G.

6 Usage Aspects

As we pointed out earlier, the interpretation of assertions $[\![U, n, \Gamma \blacktriangleright T, n]\!]$ corresponds to a linear type system at the Camelot level. Although guaranteeing benign sharing, this discipline is overly restrictive, as may be illustrated by the expression `Cons(length(x), x)`, where `x` is used as an argument for `length`, but also in the surrounding code. Motivated by similar examples involving nontrivial sharing of heap cells, Aspinall and Hofmann [2] introduced a less restrictive type system that distinguishes three different usages a program can make of a variable. These *aspects* are ordered in increasing order of permissiveness:

1. modifying use, e.g. l in `sort` l, or the destroyed parameter in in-place list append;
2. non-modifying use, but shared with result, e.g. the second argument in append;
3. non-modifying use, and not shared with result, e.g. l in `length` l.

Based on aspects we can allow duplication of variables in certain cases while preserving benign sharing. For instance the nonlinear expression $\texttt{let x} = e_1(\texttt{y}^3)\ \texttt{in}\ e_2(\texttt{x}^i, \texttt{y}^1)$ will be allowed, where variables are annotated with their relative usage aspects. In the remainder of this section, we outline an assertion format for the type system of [2], suitably adapted to the setting of Camelot compilation. To increase readability we omit numerical annotations as they have the same format and meaning as in the linear system.

Usage Aspects for the Source Language We define a notion of usage-aspect aware contexts Γ in which variables are decorated with their usage aspects; for instance $x^i : A$ if $x : A$ is used with aspect $i \in \{1, 2, 3\}$. If $x^i : A \in \Gamma$, we write $\Gamma(x) = A$ and $\Gamma[x] = i$. The "committed to i" context Δ^{i} is the same as Δ, but each declaration $x^2 : A$ is replaced with $x^i : A$. If we have two contexts Δ_1, Δ_2 which only differ on usage aspects, we define the context $\Delta = \Delta_1 \wedge \Delta_2$, to have the same domain and typing, but such that $\Delta[x] = min(\Delta_1[x], \Delta_2[x])$. Some of the typing rules are:

$$\frac{\Gamma, x^i : A \vdash e : B \quad j \leq i}{\Gamma, x^j : A \vdash e : B}\ \text{LDROP} \qquad \frac{\Gamma \vdash e_1 : B \quad \Gamma \vdash e_2 : B}{\Gamma, x^3 : \mathbf{I} \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : B}\ \text{LIF}$$

$$\frac{}{x^2 : A \vdash x : A}\ \text{LVAR} \qquad \frac{\Gamma, \Delta_1 \vdash e_1 : A \quad \Theta, \Delta_2, x^i : A \vdash e_2 : B \quad \Phi(i)}{\Gamma^i, \Theta, \Delta_1^{\mathsf{i}} \wedge \Delta_2 \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B}\ \text{LLET}$$

The LVAR rule has default aspect 2, although variables can be raised (not shown here) if heap-free or weakened (LDROP) to a more destructive usage. The most complex rule is LLET: first, the context is split into parts according to variables specific to e_1 (or e_2), that is Γ (or Θ), and common variables, possibly used with different aspects Δ_1, Δ_2. A variable whose region overlaps with the result of e_1 (i.e. of a variable that is of aspect 2 in Γ, Δ_1) inherits the aspect of x in e_2 - this is why Γ^i and Δ_1^{i} appear in the succedent. Additionally, for a variable occurring in both contexts, the resulting usage aspect should not supersede its aspects in the two antecedents. The additional side condition $\Phi(i)$ prevents any common variable from being modified in e_1 or e_2 before being referenced in e_2: namely, $\Delta_1[z] = 1$ is never allowed, $\Delta_1[z] = 3$ is always allowed, $\Delta_1[z] = 2$ is only allowed, provided neither $i = 1$ nor $\Delta_2[z] = 1$. Further, we exclude $\Delta_1[z] = \Delta_2[z] = 2$. For more details, please see [2].

Derived Assertions for Usage Aspects In preparation of the definition of derived assertions, we extend the previous auxiliary judgements by a (boolean) *separation* flag p; the relation $v, h \models_{\mathbf{L}(A)} R, p$ means that v points to a well formed list occupying a region R in a heap h. If the separation flag p is set to false then the regions occupied by elements of the list are allowed to overlap (internal sharing). Otherwise, these regions must be located in separated parts of the heap.

This definition is generalised to a relation over environments, heaps, contexts and regions in the following way:

$$\frac{E\langle x\rangle, h \models_{\Gamma(x)} R_1, \mathsf{true} \quad \Gamma, U \models^E_{h,1} R_2}{\Gamma, U \uplus x \models^E_{h,1} R_1 \uplus R_2} \qquad \frac{E\langle x\rangle, h \models_{\Gamma(x)} R_1, \mathsf{true} \quad \Gamma, U \models^E_{h,2} R_2, \mathsf{true}}{\Gamma, U \uplus x \models^E_{h,2} R_1 \uplus R_2, \mathsf{true}}$$

$$\frac{E\langle x\rangle, h \models_{\Gamma(x)} R_1, \mathsf{false} \quad \Gamma, U \models^E_{h,2} R_2, \mathsf{false}}{\Gamma, U \uplus x \models^E_{h,2} R_1 \cup R_2, \mathsf{false}} \qquad \frac{E\langle x\rangle, h \models_{\Gamma(x)} R_1, \mathsf{false} \quad \Gamma, U \models^E_{h,3} R_2}{\Gamma, U \uplus x \models^E_{h,3} R_1 \cup R_2}$$

The interpretation of aspect-aware assertions mirrors the correctness theorem in [2], extended with a freelist, but not including any reasoning about the freelist's length:

$$[\![U_1,U_2,U_3,\Gamma \blacktriangleright T]\!] \equiv \forall\, F\, R_1\, R_2\, R_3$$
$$\left(\begin{array}{l} U_1 \uplus U_2 \uplus U_3 \subseteq dom\ \Gamma \wedge \\ freelist(h,F) \wedge \\ \Gamma, U_1 \models^E_{h,1} R_1 \wedge \\ \Gamma, U_2 \models^E_{h,2} R_2, \mathsf{false} \wedge \\ \Gamma, U_3 \models^E_{h,3} R_3 \wedge \\ R_1 \cap (R_2 \cup R_3) = \varnothing \wedge \\ F \cap (R_1 \cup R_2 \cup R_3) = \varnothing \end{array}\right) \longrightarrow \left(\begin{array}{l} \exists Q H.\ v,h' \models_T Q, \mathsf{false} \wedge \\ \quad freelist(h',H) \wedge Q \cap H = \varnothing \wedge \\ \quad footprint(F \cup R_1, h, h') \wedge \\ \quad Q \subseteq (F \cup R_1 \cup R_2) \wedge \\ \quad H \subseteq (F \cup R_1) \wedge \\ \quad dom\ h = dom\ h' \wedge \\ \quad \Gamma, U_2 \models^E_{h,2} R_2, \mathsf{true} \longrightarrow v,h' \models_T Q, \mathsf{true} \end{array}\right)$$

A judgement $G \rhd e : [\![U_1,U_2,U_3,\Gamma \blacktriangleright T]\!]$ thus asserts that, whenever

- variables in U_i point in the initial heap h to sets of locations R_i according to their type in Γ and usage aspect i, with internal sharing allowed when $i \geq 2$,
- the heap regions associated with variables of aspect 1 do not overlap with heap regions related to other aspects, i.e. $R_1 \cap (R_2 \cup R_3) = \varnothing$,
- the initial heap contains a freelist inhabiting region F, which does not overlap with any region pointed to by a variable in $U_1 \cup U_2 \cup U_3$,

and the evaluation of e terminates, then there exist regions Q and H such that:

- in h', result v according to type T inhabits region Q, possibly with internal sharing,
- the final heap contains a freelist in region H, not overlapping with Q,
- data structures pointed to by variables outside U_1 and F remain unchanged,
- the result region consists of locations from the initial freelist F, locations from R_1 (corresponding to destroyed data substructures, whose space has been recycled) and from R_2, which may overlap with the result region Q,
- the final freelist region consists of locations of the initial freelist and R_1,
- no new objects are allocated,
- if region R_2 does not contain any shared (sub)structures, then neither does Q.

For example, in-place list append admits a typing that corresponds to the assertion $\rhd \mathtt{append}(l_1,l_2) : [\![\{l_1\},\{l_2\},\emptyset,(l_1 : \mathbf{L}(A), l_2 : \mathbf{L}(A)) \blacktriangleright \mathbf{L}(A)]\!]$ where $l_1(l_2)$ is the destroyed (aliased) argument.

Proof Rules for Usage Aspects We now introduce some of the derived rules.

$$\frac{\Gamma(x) = T}{G \rhd \mathtt{var}\ x : [\![\varnothing,\{x\},\varnothing,\Gamma \blacktriangleright T]\!]}\ \textsc{DUvar} \qquad \frac{G \rhd e : [\![U_1,U_2 \uplus x,U_3,\Gamma \blacktriangleright T]\!]}{G \rhd e : [\![U_1 \uplus x,U_2,U_3,\Gamma \blacktriangleright T]\!]}\ \textsc{DUdrop21}$$

$$\frac{G \rhd e_1 : [\![U_1,U_2,U_3,\Gamma \blacktriangleright T]\!] \quad G \rhd e_1 : [\![U_1,U_2,U_3,\Gamma \blacktriangleright T]\!]}{G \rhd \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : [\![U_1,U_2,U_3 \uplus x,(\Gamma, x : \mathbf{I}) \blacktriangleright T]\!]}\ \textsc{DUif}$$

$$\frac{\begin{array}{ll} G \rhd e_1 : [\![U_{11},U_{12},U_{13},\Gamma \blacktriangleright S]\!] & x \in U_{2i} \\ G \rhd e_2 : [\![U_{21},U_{22},U_{23},(\Gamma, x : S) \blacktriangleright T]\!] & \Psi(i) \end{array}}{G \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : [\![U_1,U_2,U_3,\Gamma \blacktriangleright T]\!]}\ \textsc{DUlet}$$

These rules are direct counterparts of the typing rules. In particular, in the DULET rule the real work is done by the side condition $\Psi(i)$, which statically approximates benign sharing. For instance, for $i = 2$ the side condition $\Psi(2)$ specialises to the conjunction of static assumptions $U_1 = U_{11} \cup U_{21}$, $U_2 = (U_{12} \setminus U_{21}) \cup (U_{22} \setminus \{x\})$, $U_3 = \big(U_{13} \setminus (U_{21} \cup U_{22})\big) \cup \big(U_{23} \setminus (U_{11} \cup U_{12})\big)$, $U_{11} \cap (U_{21} \cup U_{22} \cup U_{23}) = \varnothing$, $U_{12} \cap (U_{21} \cup U_{22} \cup U_{23}) \subseteq U_{23}$.

7 Conclusion and Related Work

In this paper, we have described a logic for derived assertions that allows the results of [6]'s analysis to be verified in Grail's bytecode logic. Although we have presented the logic for a specific datatype, our approach applies to algebraic datatypes in general. Because the MRG project aims to verify the consumption of a variety of resources, we have employed a general purpose logic as the basis of our formalisation. Our work is thus best compared to other work on mechanical or at least formal verification of pointer programs using variants of traditional (general purpose) Hoare logic. Historically, some of the first formal verification of pointer programs in [11] (and later [10]) used a model where the store is incorporated in the assertion logic. More recent is the verification of several algorithms, including list manipulating programs and the Schorr-Waite graph-marking algorithm, by Bornat [5] using the Jape system. This approach employs a Hoare logic for a while-language with components that are semantically modelled as pointer-indexed arrays. Separation conditions are expressed as predicates on (object) pointers. Mehta and Nipkow [13] employ the same semantic model of the heap for reasoning about pointer programs in higher-order logics. This effort extends earlier work by Nipkow et al. [15] on formalised proofs in HOL of soundness and (relative) completeness of program logics.

Proving heap-related properties has also been the topic of Separation Logic [16]. Indeed, the primitives of Separation Logic appear well suited to express the mutual separation of data structures, and their separation from the freelist more succinctly. An Isabelle/HOL implementation is presented in [18], although the author reports proofs (typically in-place reversal) to be slightly more complicated than in [13]. Furthermore, little support for automation is currently available, both for proof search and for generating invariants. Finally, properties such as heap preservation in our predicate *footprint* are more intensional than is usually the case in (Hoare-style) Separation Logic. Differently from Hoare-style logics in general, the style of our logic allows us to relate pre and post states without the use of auxiliary variables.

The contribution of the present paper is the translation of typing assertions to statements in the base logic and the formulation of derived rules which allow for automatic construction of proofs. We have indicated how the linearity restrictions may be overcome by considering the more generous sharing and separation systems induced by usage aspects. This could be pushed further toward Konečný's [9] system for layered sharing. Comparing the verification of the example programs with the verification of similar programs in the core bytecode logic demonstrates the general benefit of a proof system of derived assertions, concerning both the proof complexity and automation. Indeed, while verification in the bytecode logic appears to depend on the machinery of a general purpose theorem prover and manual intervention, a logic of derived assertions

may be implementable in a stand alone prover with access to fairly straightforward simplification capabilities.

Acknowledgements This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing. We would like to thank all our colleagues and in particular David Aspinall for this role in implementing the certificate generation tactic.

References

1. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In K. Slind, A. Bunker, and G. C. Gopalakrishnan, editors, *Proceedings of TPHOLs'04*, volume 3223 of *LNCS*, pages 34–49. Springer, Sept. 2004.
2. D. Aspinall and M. Hofmann. Another type system for in-place update. In D. L. Métayer, editor, *Proceedings of ESOP'02*, volume 2305 of *LNCS*, pages 36–52. Springer, Apr. 2002.
3. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Towards certificate generation for linear heap consumption. In *Proceedings of LRPP'04*, July 2004.
4. L. Beringer, K. MacKenzie, and I. Stark. Grail: a Functional Form for Imperative Mobile Code. In *Proceedings FGC'03*, volume 85(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, June 2003.
5. R. Bornat. Proving Pointer Programs in Hoare Logic. In R. Backhouse and J. Nuno Oliveira, editors, *Proceedings of MPC'00*, volume 1837 of *LNCS*, pages 102–126, July 2000.
6. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL'03*, pages 185–197. ACM Press, Jan. 2003.
7. C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
8. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.
9. M. Konečný. Functional in-place update with layered datatype sharing. In M. Hofmann, editor, *Proceedings of TLCA'03*, volume 2701 of *LNCS*, pages 195–210. Springer, June 2003.
10. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.
11. D. C. Luckham and N. Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, Oct. 1979.
12. K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware Functional Programming on the JVM. In S. Gilmore, editor, *Proceedings of TFP'03*, pages 29–46. intellect, 2003.
13. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Proceedings of CADE-19*, volume 2741 of *LNCS/LNAI*, pages 121–135. Springer, Aug. 2003.
14. G. C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
15. T. Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In J. Bradfield, editor, *Proceedings of CSL'02*, volume 2471 of *LNCS*, pages 103–119, Sept. 2002.
16. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of LICS'02*. IEEE Computer Society, July 2002.
17. D. Sannella and M. Hofmann. Mobile Resource Guarantees. EU Project IST-2001-33149, 2002–2004. http://groups/inf.ed.ac.uk/mrg/.
18. T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Proceedings of CSL'04*, volume 3210 of *LNCS*, pages 250–264. Springer, Sept. 2004.