Relational bytecode correlations

Lennart Beringer[1]

Institut für Informatik, Ludwig-Maximilians-Universität München, Oettingenstrasse 67, 80538 München, Germany Tel/Fax: ++49-(0)89-2180 9864/9338

Abstract

We present a calculus for tracking equality relationships between values through pairs of bytecode programs. The calculus may serve as a certification mechanism for non-interference, a well-known program property in the field of language-based security, and code transformations. Contrary to previous type systems for non-interference, no restrictions are imposed on the control flow structure of programs. Objects, static and virtual methods are included, and heap-local reasoning is supported by frame rules. In combination with polyvariance, the latter enable the modular verification of programs over heap-allocated data structures, which we illustrate by verifying and comparing different implementations of list copying. The material is based on a complete formalisation in Isabelle/HOL.

Key words: Non-interference, Relational proof systems, Program transformations, Proof-carrying code, Formalised program analyses

1. Introduction

Non-interference is a well-known program property in the area of language-based security [1]. In its most basic form for a simple imperative language over integers, it may be formulated by separating the program variables disjointly into public (low security) L and private (high security) variables H. The property then requires that the program preserves the relation $=_L$ between states, which is to say that the final states of two executions agree on all variables in L whenever the initial states did.

A similar property specifies the semantic validity of program transformations: two executions (now of *different* programs) commencing in *identical* states should yield *identical* states, or should at least yield return states which agree on all variables that are relevant for the ensuing program continuation.

Email addresses: `beringer@tcs.ifi.lmu.de` (Lennart Beringer)
URL: `http://www.tcs.informatik.uni-muenchen.de/~beringer` (Lennart Beringer)
[1]Present address: Department of Computer Science, Princeton University, 35 Olden Street, Princeton 08544, New Jersey

In this article, we consider a common abstraction of these two notions called (partial-correctness) *program correlations*, and present technology for certifying when programs or their executions are correlated.

In order to motivate such a unifying treatment, let us first observe that non-interference is not only robust under semantics-preserving transformations but that the two notions are in fact of equal strength: if c' is a semantically valid transformation of c and h is a variable occurring in neither c nor c', then

$$\mathit{if}\ h\ \mathit{then}\ c\ \mathit{else}\ c'$$

is non-interferent. On the other hand, if c is non-interferent and its high-security variables are among $h_1, \ldots, h_n$, then c_2 is a semantically correct transformation of c_1, where

$$\begin{array}{lcl} c_1 & := & h_1 := 0; \ldots ; h_n := 0; c; h_1 := 0; \ldots ; h_n := 0 \\ c_2 & := & h_1 := 1; \ldots ; h_n := 1; c; h_1 := 0; \ldots ; h_n := 0. \end{array}$$

The latter half of this duality is a variant of a well-known observation [2, 3] that underlies *self-composition* [4]. However, with the exception of [5] we are unaware of attempts to exploit this proximity for the development of flexible verification formalisms for non-interference.

Our verification technology concerns an idealised subset of sequential bytecode and is based on novel abstractions of states and state pairs. These abstractions approximate equality and separation between values held in a single state and correlate values across a pair of states. The verification technology itself consists of unary and relational proof systems for programs and program pairs which track the evolution of abstract states. As a consequence of the characteristics of abstract states, the proof systems capture a type of copy propagation, are applicable to structured as well as unstructured bytecode without requiring additional control flow information, admit heap-local reasoning, and are compatible with peep-hole transformations. All these characteristics separates our approach from previous non-interference type systems for bytecode [6, 7, 8]. We illustrate similarities and differences to these and other static analyses using a number of example programs, including a recursive method over heap-allocated lists with a complex non-interference specification. Motivated by proof-carrying-code (PCC, [9]) considerations we have carried out a complete formalisation of the work reported in the present article, using the Isabelle/HOL theorem prover [10]. A formalisation snapshot is available electronically [11].

1.1. Motivating example programs

We briefly demonstrate some restrictions of previous type systems and illustrate them by simple example programs. To enhance readability, the programs are given as Java code. We will later return to these examples and show how our approach admits the verification of their respective bytecode representations.

Copy propagation, correlated values and operations. Consider the class definition in Figure 1. Phrased in traditional terms, the verification task consists of showing that `C.m2` is non-interferent if `l` is a public argument and `h` a private argument. The code exhibits the following characteristics which prevent it (and its bytecode representation) from being verifiable using existing type systems.

```
class C {
 int A;
 int m1(int y){ this.A := y; return y }
 int m2(int l; int h){
  x := l;
  if h
  then { x := x+3; v := new C; v.A := x; return v.A }
  else { x:=3; x:=l+x; v:= new C; return v.m1(x) }
 }
}
```

Figure 1: Example: tracking constants and copies

- The value initially held in `l` is copied to the auxiliary local variable `x` which is then used in the positive branch, while the negative branch uses `l` and employs `x` for some different purpose. Copy propagation is not taken into account by previous approaches, so the equality between `l` and `x` cannot be tracked or exploited.
- The value originally held in `l` is incremented by the same value in both branches, but the branches differ slightly in the way this increment is constructed. Like copy propagation, the introduction of identical values by two execution paths, and the propagation of value correlations by arithmetic operations are not supported – but the fact that the results of the additions are identical is critical for showing the equality of the final return values.
- Both branches create local (i.e. in the conventional sense: private, due to the existence of an enclosing *high* branch condition) objects. They also assign values to the `A`-field of these objects, in case of the negative branch by means of a method invocation. Previous type systems consider "privately allocated" objects as invisible and propagate this restriction to any values that flow through fields or methods of private objects, i.e. designate any value read from a field of a private object as private.
- In order to prove that the values returned by the two branches are identical, it is necessary to track their flow through the field assignment of the positive branch and the invocation of method `m1` in the negative branch. Typically, this is not supported by existing type systems as no relationship is maintained between initial and final values of method calls.

The calculi we present do not classify variables or fields statically as private or public, do not single out code regions according to the visibility of the branch condition, lift the restrictions on object allocation, and admit the tracking of values and their copies through fields and methods.

As a small variation of `C.m2`, one may consider a method `C.m3` of return type `C` that arises from `C.m2` by replacing the return expression of the positive branch by `return v` and that of the negative branch by the sequence `x:= v.m1(x); return v`. In this case, a client of method `C.m3` should be able to exploit the fact that field `A` of the returned object may be considered low (i.e. contains correlated values in the two

```
abstract class LIST { LIST Copy () }
class NIL extends LIST { NIL Copy(){ return (new NIL) } }
class CONS extends LIST { Object HD; LIST TL;
 CONS Copy() {
   h:= this.HD; t:= this.TL; t:= t.Copy();
   z:= new CONS; z.HD := h; z.TL := t; return z }
}
```

Figure 2: Example: list copying

executions), irrespective of the execution path chosen inside `C.m3` to create the object and assign to its field.

It may be argued that some of these restrictions can be overcome by applying analyses and transformations that bring the code into a form that is acceptable to existing type systems. Indeed, `C.m2` is equivalent to

```
int m4(int l){x := l+3; v := new C; v.A:= x; return x}
```

However, such transformations need to be certified in a trustworthy system in order to ensure that only semantics-preserving manipulations have been applied. Demonstrating the usefulness of the relational approach, we will therefore validate the equivalence between m2 and m4 in Section 4.3, thus highlighting how the close relationship between non-interference and program transformations may be exploited in practise.

Heap-local reasoning. The effect of method invocations on heap objects is usually constrained along the axis of visibility (i.e. the separation public/private), irrespectively of the reachability of objects from the method parameters [6]. For example, giving a method *m* declared by

int C.m(D x){...}

the heap effect *high* indicates that at most private fields are modified. This assertion, however, is too weak for guaranteeing that private fields or objects whose existence is irrelevant for method *m* remain unchanged. This includes fields of objects that are not reachable following fields access paths from the receiver object or the method argument. Similarly, a heap effect *low* allows *any* field to be updated (again irrespective of its reachability from the method arguments), destroying any data structure invariant on whose validity the client code relies.

Programs where heap-local reasoning is of particular importance are recursive programs over algebraic data structures such as lists and trees. Figure 2 contains a representation of lists, where the two constructors are represented as subclasses of a common abstract superclass. Written in a recursive style, the method copies the spine of a list without duplicating the content elements pointed to by HD. In Section 5, we will verify that this code indeed lays out the copy in a fresh memory area. We will also compare bytecode variations resulting for the code from Figure 2 and thus provide a further application to program transformations. Finally, we will show that the code is non-interferent for lists where the visibility of the elements pointed to by HD alternates. To our knowledge, such policies would be impossible to model and hence verify in formalisms that statically associate security domains to fields unless specification-dependent code modifications are permitted. All proofs in Section 5 exploit the fact that recursive calls leave environmental objects unchanged and thus preserve the well-structuredness of list segments and the (non-)visibility of content data. The fine-grained tracking of objects necessary for such local reasoning is a characteristic of our approach that is inspired by separation logics [12]. Indeed, the proof systems we present include frame rules that enable the embedding of local judgements in contexts where additional heap objects are present. In contrast to the work of [12] however, our frame rules (like all the other proof rules) occur as part of a (domain-specific) static calculus rather than a general-purpose program logic.

1.2. Previous work

Throughout the paper, we will compare specific aspects of our calculi with properties of some formalisms from the literature. We therefore briefly discuss some immediately relevant pieces of previous work in the present section. Pointers to further related work will be given at the end of the article.

Relational program logics. A core component of our technology consists of a relational proof system, i.e. a calculus that exposes the two-execution-nature of correlations by judgements over pairs of program phrases. At the level of program logics, Benton [13] advocates a relational formulation for verifying transformations and interpreting existing type systems for non-interference, Amtoft et al. [14] present a logic where assertions have unary as well as relational interpretations, and Yang [15] introduces a relational form of separation logic. In contrast, the calculus introduced in the present article is purely static, i.e. trades logical precision for potential of automated proof checking. As a further difference, we consider a sequential fragment of the Java Virtual Machine Language, while the cited works consider languages of structured commands and while-loops (in the case of [13]: without objects or methods). The relationship between our work and the cited works is thus similar to that of unary type systems and non-relational Hoare-logics [16]: we expect our calculus to be embeddable in a suitable adaptation of the relational program logics to bytecode. In the absence of a formalised relational bytecode logic our development justifies the calculi by direct reference to the operational semantics, similar to the work on foundational PCC [17].

Type-systems for bytecode-level non-interference. Previous bytecode type systems for non-interference have largely been based on transferring Volpano et al.'s concept of *pc*-type [18], where the security level of branch and loop conditions determines the legality of assignments to variables and fields in the code regions dominated by the branch [6, 7, 8]. Consequently, these approaches require control flow information on whose correctness the overall soundness of the analysis relies. In the case of [7], this information is communicated by interspersing the original code with suitable pseudo-instructions. Barthe et al. [6] and Kobayashi-Shirane [8] capture similar information in meta-level functions. In the case of [6], axioms (called safe overapproximation properties – SOAP's) are presented that capture those properties on which the soundness proof relies. The relational structure of judgements separates our approach from these

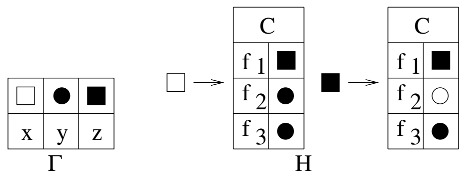

Figure 3: Illustrating an abstract state for Java

analyses, and also from approaches based on formal dependencies between variables or def/use relationships such as (at the level of Java) [19]. Indeed, our approach relaxes the above restrictions regarding assignments to fields and variables, avoids the *pc*-type, and consequently eliminates the need for additional control flow information. In particular, it is applicable to structured as well as unstructured bytecode.

We should stress that our approach concerns, in accordance with the intended compatibility with semantics-preserving transformations, *extensional* interpretations of non-interference policies, i.e. concerns only initial and final states. In contrast, some of the above-cited works apply a more intensional view where visibility restrictions of fields are to be respected throughout an execution. Furthermore, we restrict our attention to termination-insensitive interpretations, i.e. consider two executions vacuously equivalent if one of them fails to terminate.

Translation validation using symbolic execution. Viewed as a system for certifying transformations, our approach may be seen as a variant of symbolic execution as employed by Necula [20] and Tristan and Leroy [21, 22]. Indeed, our judgements relate initial and final states, capturing a similar property as symbolic expressions which specify the content of variables at the end of phrases (basic blocks) in terms of their values in the initial states. In contrast to our separation constructions, however, the cited works treat the memory as a monolithic block. As a consequence, some of the equations on exchanging the order of memory operations that are required in Tristan and Leroy's verification of list scheduling are immediately derivable in our calculi. On the other hand, we do not consider issues of inference in the present paper (as Necula does), and only consider a single (and substantially simpler) language.

1.3. Components of our approach

In order to give the reader an intuitive understanding of our approach, we summarise the main ideas, again at the level of Java.

Abstract states. In the absence of explicit classifications of program variables, object fields or code regions according to their security level, the fact that a variable or field contains identical values in two states is expressed directly, using a notion of (pairs of) abstract states. An abstract state is comprised of similar components as a concrete state, but contains abstract values ("colours") in the place of concrete runtime values. Figure 3 illustrates the idea of abstract states at the level of Java. The component to the left represents an abstract store, mapping variables x, y, and z to colours □, •,

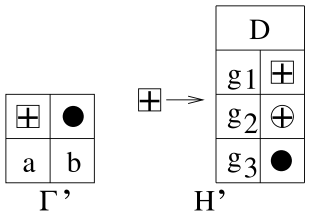

Figure 4: A further abstract state for Java

and ■. The component to the right represents an abstract heap that is comprised of two elements, at abstract locations □ and ■. Both abstract objects are of class C and contain abstract values in the fields f_1, f_2, f_3. Abstract states for bytecode will also contain an abstract operand stack, and will be complemented by a component that captures type information. The latter will, for example, indicate that □ and ■ represent reference values, and • and ◦ integer values.

Occurrences of an abstract value in different storage locations of an abstract state represents equality of the concrete values in the corresponding locations of a compatible concrete state. Thus, a state suitable for our example abstract state will contain the same (integer) value in all •-positions, and the same (reference) value in all ■-positions. On integers, different abstract values may represent the same concrete value. Thus, the runtime values for • and ◦ may in fact be identical. In contrast, different address values are always abstracted to different colours: the runtime values interpreting ■ and □ are required to be different, hence the concrete state will be guaranteed to contain two *distinct* C-objects. This property reflects the fact that each object allocation introduces a fresh address, while no such freshness condition is guaranteed to hold for the integer instructions. The resulting separation discipline enables heap-local reasoning in the style of Separation Logic [12]. Indeed, our calculus includes a frame rule similar to the one of Separation Logic.

Each abstract state thus contains statically approximative information regarding the copies of runtime values in a single concrete state. In order to capture information about the correlation of values across two executions, we use structures called *relational state descriptions* (RSD's). These contain two abstract states, together with a single type-info component of the kind mentioned above. Ignoring the latter component as well as abstract operand stacks again, Figure 4 depicts a further abstract state (Γ', H'), which together with the state (Γ, H) from above yields the RSD $\phi = ((\Gamma, H), (\Gamma', H'))$.

RSD's are interpreted over pairs of concrete states, one from each execution. In addition to requiring that each concrete state satisfies its abstract counterpart, the interpretation mandates that whenever a colour occurs jointly in both abstract states (possibly in different abstract storage locations), the runtime values interpreting such a colour in the concrete states must be indistinguishable. In particular, any colour of integer type must be interpreted identically in the two concrete states. For example, all positions containing • in Figures 3 and 4 contain the same (integer) value. This discipline generalises the relations $=_L$ mentioned above if both abstract stores contain at least the variables in L. However, our notion enables the comparison of abstract states of different – even distinct – domains, as indicated by the use of variables in Figure 4 that are distinct from those in Figure 3. Furthermore, RSD's inherit the ability to track copy propagation from abstract states and may associate different colours with a variable at different program points. In particular, they do not require a static classification of variables into public and private variables. When analysing non-interference, these features are useful when comparing states that stem from different branches of a conditional, for example when one branch introduces a further variable. With respect to the certification of program transformations, these features allows us to relate code fragments that have undergone renaming, coalescing, or splitting of variable names.

The treatment of fields is similar to that of variables, again in contrast to the static classification of field names into public and private fields in previous work. With respect to objects, the colours occurring in the domains of the abstract heaps of an RSD implicitly determine a partial bijection between addresses in the style of Banerjee & Naumann and Barthe et al. [23, 6].

Summarising, our approach tracks equality of values across two states irrespectively of their storage locations. Roughly speaking, colours that occur in both abstract states of an RSD amount to public ("correlated") values, while colours present only in one of the abstract states play the role of private values.

Proof systems. Our verification approach employs static proof systems which - like type systems - approximate when a program pair satisfies the desired runtime property but are necessarily incomplete. We envision that systems like the one we present will mostly be used as a low-level formalism in proof-carrying-code (PCC) applications. In accordance with this, the side conditions of the proof rules are computationally simple. In contrast, full program logics are more expressive but require non-trivial computations in order to discharge the side condition of the rule of consequence. As a purely syntax-directed fragment would be to weak, we include some structural rules, at the price of complicating proof search. In the present paper, we are not concerned with proof inference, which we believe will be mostly performed at higher language levels and may only be feasible for restricted subsystems. Our treatment of high-level data structures in Section 5 represents a first step in this direction and indicates how one may bridge the gap between the operational semantics of low level code and high-level invariants of application programs and data structures.

Our main judgement form associates a pair of bytecode phrases with a pre-RSD and a post-RSD, as in $\vdash c, c' : \phi \to \psi$. In accordance with our initial observation, such a judgement may be read as concerning either a program transformation or non-interference. The former reading expresses the semantic soundness of transforming c into c', where ϕ captures correlations between the input states of the two executions and ψ captures correlations between the terminal states. The second reading asserts non-interference of c if c and c' are identical, where again ϕ captures correlations between the input states of the two executions and ψ captures correlations between the terminal states. Cases where c and c' differ arise in proof trees of such judgements, for example if the top-level command is a conditional and c and c' are the respective branches.

Judgements are interpreted in a partial-correctness style, which in the case of non-interference amounts to termination-insensitivity. A special case of the interpretation asserts that the final states of two terminating executions commencing in states related by the pre-RSD are related by the post-RSD. However, the formal interpretation of judgements is more general, as it extends this guarantee to all separated state extensions of the abstract states mentioned in the judgement. This interpretation simplifies the proof of the frame rule, and also yields a stronger guarantee than previous type systems, as non-interference is guaranteed to hold in contexts containing additional objects: paraphrasing and slightly simplifying the formal development to make it applicable to Java, we will define a property

$$\mathit{Safe}_{c,c'}(\phi,\psi) := \forall\, s\; s'\; t\; t'.\; s =_{\phi} s' \wedge s, c \downarrow t \wedge s', c' \downarrow t' \Longrightarrow t =_{\psi} t'$$

where $s =_{\phi} s'$ denotes indistinguishability of s and s' w.r.t. RSD ϕ, and $s, c \downarrow t$ represents the operational semantics. The interpretation of a relational judgement $\vdash c, c' : \phi \rightarrow \psi$ will then be given by

$$\mathit{Interprete}_{c,c'}(\phi,\psi) := \forall\, \phi'\; \psi'\; \xi.\; \phi' = \phi * \xi \wedge \psi' = \psi * \xi \Longrightarrow \mathit{Safe}_{c,c'}(\phi',\psi'),$$

where $*$ denotes the separating conjunction of RSD's, which in particular requires the domains of the abstract heaps to be distinct. By choosing ξ to be the empty RSD this interpretation specialises to $\mathit{Safe}_{c,c'}(\phi,\psi)$. Choosing non-trivial ξ yields the guarantee that non-interference (or correctness of a transformation) will be satisfied by concrete states $s, \ldots, t'$ that satisfy the extended RSD's, i.e. contain additional objects for the colours in the abstract heaps in ξ. These states will satisfy $t =_{\psi'} t'$ whenever $s =_{\phi'} s'$, $s, c \downarrow t$, and $s', c' \downarrow t'$ hold. The situation may be compared to notions of observational equivalence: there, phrases are examined with respect to their behaviour in program contexts, while our notion compares their behaviour in terms of state contexts[2].

Our proof system is governed by a distinction between correlated and non-correlated events. Correlated events occur when the two program runs execute instructions with "corresponding" effects. In our case, correlations may exist between pairs of instructions that introduce values to states (i.e. the instruction forms const i, new C, arithmetic operations, and method invocations), and between pairs of conditionals. *All* instruction forms may occur uncorrelated, in which case they only affect one of the two executions. The latter class in particular includes operations that transfer existing values between the components of states (load/store, simple stack operations, field access), and method invocations and conditionals that occur only in one execution.

In accordance with this dichotomy, the proof system is comprised of rules for correlated instructions and rules for non-correlated instructions. Those of the latter kind are isolated as a separate proof system of unary judgements. Roughly speaking, uncorrelated segments correspond to high code regions in extant type systems, but in the absence of a formal separation into high and low code regions this notion is only phenomenologically observed rather than actively enforced. In particular, instruction forms that transfer abstract values between the components of an abstract state (for

[2]Having completed the work described in this article, we learnt that similarly extended interpretations of Separation Logic judgements have been employed by Birkedal and Yang for modelling the semantics of higher-order separation logics, and have been called *resource Kripke semantics* by these authors [24].

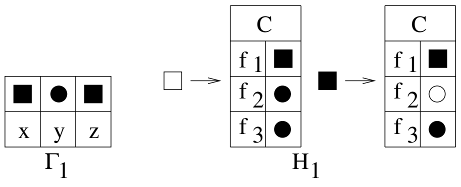

Figure 5: Illustrating the effect of applying $x := x.f_1$ on the state from Figure 3

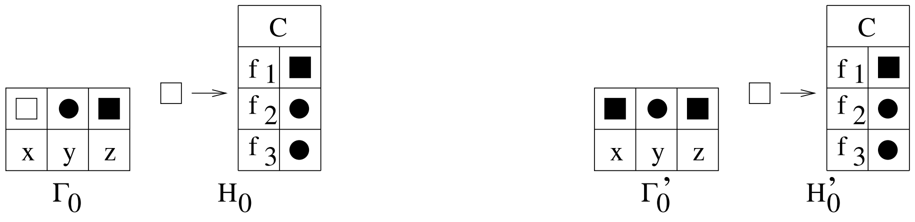

Figure 6: Local-reasoning transition for field access

example field access instructions) are verified using unary rules, irrespectively of the existence or visibility of enclosing branch conditions. For example, the effect of executing (the sequence of bytecode instructions corresponding to) the instruction $x := x.f_1$ on the abstract state of Figure 3 yields the abstract state (Γ_1, H_1) shown in Figure 5.

The main system of relational proof rules includes the unary proof rules by virtue of appropriate injection rules. Thus, carrying out the above field access operation to the RSD ϕ yields the RSD $((\Gamma_1, H_1), (\Gamma', H'))$. In particular, the effect is independent of the visibility of the object or the field involved in the access operation, in contrast to the above-mentioned restrictions on field access operations in previous type systems. Similar comments apply to method invocations that are applied only to one of the components.

The relational proof system also contains rules for correlated events. These affect both components and resemble low proof rules of traditional type systems. For example, we have rules for correlated occurrences of conditionals and method invocations, and rules for synchronised value creation, for integer values as well as references (i.e. object allocation). The latter rules introduce fresh colours to both abstract states. In contrast to traditional low proof rules, however, our rules do not require the correlated instructions to occur at the same program point.

The frame rules embed judgements into contexts with additional objects or colours and thus support heap-modular reasoning. Indeed, the transition from Figure 3 to Figure 5 may be justified by framing the object ■ onto the transition from (Γ_0, H_0) to (Γ_0', H_0') given in Figure 6.

Furthermore, there are axiom rules for extracting assumptions from appropriate (polyvariant) unary and relational proof contexts, and further structural rules.

The declarative style of abstract states ensures that the proof systems are flowinsensitive [25], with respect to variables as well as fields.

In order to maintain the copy propagation information embodied in abstract states across method invocations, judgements are required to preserve colours and abstract addresses. This means that the administrative component of the post-RSD must contain the administrative component of the pre-RSD, and that each abstract object in the abstract heap of an initial abstract state must be present (albeit with potentially different abstract values in the abstract fields) in the corresponding abstract heap of the final RSD. Thus, while the interpretation of object colours of a *single* RSD amounts to a partial bijection between addresses (essentially the partial bijection constructed in the work of [23] and [6]), the resulting discipline regarding objects *across a judgement* is slightly different from [23] and [6]. The interpretation in the latter works requires the existence of a partial bijection on the post-heaps that extends the partial bijection on the pre-heaps. Both partial bijections capture correlated ("low") objects but ignore non-correlated ones. Our interpretation explicitly relates pre- and post-objects, irrespective of their visibility level. This preservation of colours and objects is also imposed on the unary proof system. As a result, one may specify restricted functional correctness policies which, for example, guarantee that the result of a method points to a freshly allocated object, or that it coincides with the value passed in a certain argument position. The latter property is useful for tracking copies through method calls.

Abstract representation predicates. In order to specify non-interference and transformation relationships in the presence of heap-allocated data-structures, we introduce representation predicates that construct or characterise components of RSD's. Their definition proceeds by induction on meta-level data structures and makes use of separation operators for abstract states. Indeed, these predicates resemble datatype representation formulae in separation logic that specify concrete states. Together with polyvariance of the proof system, and the frame rules, they enable the structured formulation and heap-modular verification of complex unary and relational properties of recursive methods over inductive data types.

1.4. Outline

Section 2 introduces syntax and operational semantics of our chosen language fragment. The derivation system for unary judgements is then presented in Section 3, followed by the relational system in Section 4. In both cases, example verifications illustrate core properties of the system, complementing the more theoretical developments. In Section 5, we describe the verification of code over heap-allocated data structures, using abstract representation predicates and illustrated by the code for list copying. We conclude by discussing further related and future work in Section 6.

2. Syntax and operational semantics

We consider an idealised subset of the JVML where programs are formulated over the disjoint sets $\mathcal{X}$ of (local) variables, C of class names, $\mathcal{M}$ of method names, and $\mathcal{F}$ of field names, ranged over, respectively, by x, C, m, and f and similar letters. Program labels $\ell \in C \times \mathcal{M} \times \mathbf{N}$ comprise a class name, a method name, and an instruction counter l. Programs map labels to instructions of the grammar

$$\begin{array}{rcl} \iota & ::= & \mathsf{const}\ i \mid \mathsf{dup} \mid \mathsf{pop} \mid \mathsf{swap} \mid \mathsf{load}\ x \mid \mathsf{store}\ x \mid \mathsf{binop} \oplus \mid \mathsf{new}\ C \mid \mathsf{getf}\ C.f \\ & & \mid \mathsf{putf}\ C.f \mid \mathsf{invStat}\ C.m \mid \mathsf{invVirt}\ C.m \mid \mathsf{goto}\ \ell \mid \mathsf{ifeq}\ \ell \mid \mathsf{vreturn} \end{array}$$

The category $\mathcal{V}$ of values (ranged over by v) comprises integers i and addresses $a \in \mathcal{A}$, while $\oplus$ ranges over binary integer operators such as $\mathsf{add}, \mathsf{mul}, \ldots$. Compared to the full sequential fragment of JVML the most significant difference is the omission of exceptions, null references and arrays. The treatment of these features is a topic for future research (see Section 6).

We write $\mathcal{P}(\ell) = \iota$ to identify the instruction at label ℓ. Throughout the paper, we require that all jumps in the body of a method $C.m$ have targets in $C.m$ and that instructions in each body are numbered consecutively starting from 0. To simplify notation, we write $\ell + 1$ for $(C, m, l + 1)$ where $\ell = (C, m, l)$. We denote the subclass relation by $C \leq C'$, require that overriding method declarations use the same formal lists of parameters as the overridden declarations, and denote the parameter list of a method by $params(C, m)$.

The operational semantics is given over states $s = (O, \sigma, h)$ comprising an operand stack $O \in \mathcal{V}^*$, a store $\sigma \in \mathcal{X} \rightharpoonup_{fin} \mathcal{V}$, and a heap $h \in \mathcal{H}$. Heaps are modelled as finite maps from locations to objects, where an object comprises a class identifier (the dynamic class) and a field table: $\mathcal{H} = \mathcal{A} \rightharpoonup_{fin} (C \times (\mathcal{F} \rightharpoonup_{fin} \mathcal{V}))$. To fix some notation, $|L|$ denotes the length of some list L, $L!n$ the item at position n (where $0 \leq n < |L|$), $[\ ]$ the empty list, :: the cons operation, and $cod\ L$ the set of elements contained in L. $A \rightharpoonup_{fin} B$ is the type of finite partial maps from A to B, with lookup, update, containment, union, and delete operations $. \downarrow .$, $.[. \mapsto .]$, $. \subseteq .$, $. \cup .$ and $. - .$, $[]$ the empty map, and dom . and cod . defined as usual.

The dynamic semantics is given by two mutually recursive judgement forms. The big-step judgement form $\mathcal{P} \vdash s, \ell \Downarrow h, v$ is defined by the rules

$$\textsc{Vret}\ \frac{\mathcal{P}(\ell) = \mathsf{vreturn}}{\mathcal{P} \vdash (v :: O, \sigma, h), \ell \Downarrow h, v} \quad \text{and} \quad \textsc{Run}\ \frac{\mathcal{P} \vdash s, \ell \rightarrow t, \ell' \qquad \mathcal{P} \vdash t, \ell' \Downarrow h, v}{\mathcal{P} \vdash s, \ell \Downarrow h, v}$$

and models the (terminating) execution from label ℓ until the end of the current method invocation frame. Rule Run involves a small-step judgement $\mathcal{P} \vdash \ell, s \rightarrow \ell', t$ as a hypothesis. Small-step judgements model the execution of single instructions. The appropriate rules are given in Figure 7. Note that the cases for method invocations (rules InvS and InvV) refer in turn to big-step judgements regarding the method bodies. A similarly structured operational semantics is used in [22].

3. Unary proof system

We now present the unary proof system. In addition to occurring as a subsystem of the relational proof system, this system is of significance in its own right, as it is able to certify basic functional correctness properties that involve identity relationships

Name	$\mathcal{P}(\ell)$	O	ℓ'	t	*Side conditions*
Const	const i	O	$\ell+1$	$(i :: O, \sigma, h)$	
Load	load x	O	$\ell+1$	$(v :: O, \sigma, h)$	$\sigma{\downarrow}x = v$
Store	store x	$v :: O'$	$\ell+1$	$(O', \sigma[x \mapsto v], h)$	
New	new C	O	$\ell+1$	$(a :: O, \sigma, k)$	$\begin{cases} a \notin dom\ h \\ k = h[a \mapsto (C, [\,])] \end{cases}$
Get	getf $C.f$	$a :: O'$	$\ell+1$	$(v :: O', \sigma, h)$	$\begin{cases} h{\downarrow}a = (C', A) \\ C' \le C \\ A{\downarrow}f = v \end{cases}$
Put	putf $C.f$	$v :: a :: O'$	$\ell+1$	(O', σ, k)	$\begin{cases} h{\downarrow}a = (C', A) \\ C' \le C \\ A' = A[f \mapsto v] \\ k = h[a \mapsto (C', A')] \end{cases}$
IfT	ifeq ℓ_1	$0 :: O'$	ℓ_1	(O', σ, h)	
IfF	ifeq ℓ_1	$i :: O'$	$\ell+1$	(O', σ, h)	$i \neq 0$

$$\textsc{Instr}\ \frac{\textit{see table}}{\mathcal{P} \vdash (O, \sigma, h), \ell \to t, \ell'}$$

$$\textsc{InvS}\ \frac{\begin{array}{c}\mathcal{P}(\ell) = \mathsf{invStat}\ C.m \quad params(C, m) = [x_1, \ldots, x_n] \\ \mathcal{P} \vdash ([], [x_i \mapsto v_i], h), (C, m, 0) \Downarrow k, v\end{array}}{\mathcal{P} \vdash ([v_1, \ldots, v_n]@O, \sigma, h), \ell \to (v :: O, \sigma, k), \ell + 1}$$

$$\textsc{InvV}\ \frac{\begin{array}{c}\mathcal{P}(\ell) = \mathsf{invVirt}\ C.m \qquad h{\downarrow}a = (C', A) \\ params(C', m) = [x_1, \ldots, x_n] \qquad C' \le C \\ \mathcal{P} \vdash ([], [x_i \mapsto v_i, \mathsf{this} \mapsto a], h), (C', m, 0) \Downarrow k, v\end{array}}{\mathcal{P} \vdash ([v_1, \ldots, v_n, a]@O, \sigma, h), \ell \to (v :: O, \sigma, k), \ell + 1}$$

Figure 7: Dynamic semantics (rules for dup, pop, swap, goto ℓ', and binop $\oplus$ omitted)

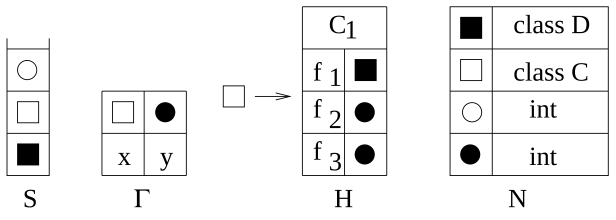

Figure 8: An abstract state

between initial and final values, and freshness conditions for objects allocated by a phrase.

For the remainder of the paper, let $\mathfrak{C}$ be an infinite set of identifiers ("colours"), ranged over by γ, δ and similar letters. Moreover, we denote by $\mathcal{T}$ the set of types, whose constituents (typically ranged over by *tp*) are int and terms of the form $\mathsf{class}(C)$. To simplify the notation, we also suppress explicit references to $\mathcal{P}$ from most definitions.

3.1. Abstract states

Definition 1. *(Abstract states) An abstract state is a triple* $\Sigma = (S, \Gamma, H)$ *where* $S \in \mathfrak{C}^*$ *is called an abstract operand stack,* $\Gamma \in \mathcal{X} \rightharpoonup_{fin} \mathfrak{C}$ *an abstract store, and* $H \in \mathfrak{C} \rightharpoonup_{fin} (C \times (\mathcal{F} \rightharpoonup_{fin} \mathfrak{C}))$ *an abstract heap. The domain of* Σ *is defined by* $Dom(\Sigma) = cod\ S \cup cod\ \Gamma \cup dom\ H \cup \bigcup_{(C,F) \in cod\ H} cod\ F$*, i.e. the set of colours occurring in* Σ.

An administrative map is a structure $N \in \mathfrak{C} \rightharpoonup_{fin} \mathcal{T}$*, associating a type to each colour in its domain. The object domain of N is defined by* $ODom(N) = \{\gamma \mid \exists C.\ N{\downarrow}\gamma = \mathsf{class}(C)\}$.

Example. Figure 8 shows an abstract state with the four colours □, ○, ●, *and* ■. *The state consists of an abstract operand stack with three elements, an abstract store with entries for variables x and y, and an abstract heap comprising a single element* □ *of class* C_1 *with three abstract fields. The figure also shows an administrative map indicating the typing information of the four colours.*

Abstract state $\Sigma = (S, \Gamma, H)$ is *closed* with respect to N if $Dom(\Sigma) \subseteq dom\ N$, $Dom(\Sigma) \cap ODom(N) \subseteq dom\ H$, and for all $H{\downarrow}\gamma = (C, F)$ there is some C' with $C \leq C'$ and $N{\downarrow}\gamma = \mathsf{class}(C')$. This means that all colours in Σ occur in N, H contains objects for all colours in Σ that identify heap objects, and all abstract objects in H are associated with an object colour in N such that the class associated with γ in H is a subclass of that associated with γ in N. In particular, the second condition expresses that there are no dangling abstract pointers.

Example. The abstract state in Figure 8 is not closed since the colour ■ *occurs in S but H does not contain a corresponding object. Figure 9 shows an appropriately extended abstract state* Σ*, which is indeed closed provided that* $C_1 \leq C$ *and* $D_1 \leq D$ *hold.*

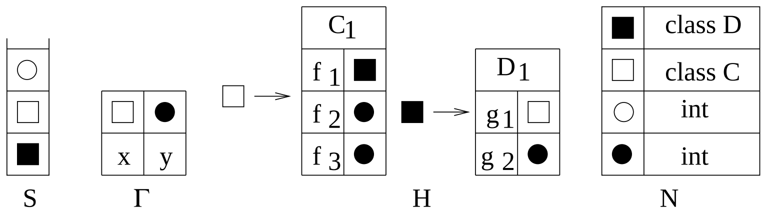

Figure 9: Abstract state $\Sigma = (S, \Gamma, H)$ closed w.r.t. administrative map N

Next, we define the interpretation $[\![\gamma]\!]^s_{\Sigma,I}$ of colour γ with respect to abstract state Σ, concrete state s, and a function I assigning addresses to colours. The interpretation consists of the set of values stored in s at Σ-positions containing γ.

$$[\![\gamma]\!]^{(O,\sigma,h)}_{(S,\Gamma,H),I} \equiv \{O!n \mid S!n = \gamma\} \cup \{\sigma{\downarrow}x \mid \Gamma{\downarrow}x = \gamma\} \cup X_{H,I,\gamma} \cup \left\{A{\downarrow}f \mid \begin{array}{l} \exists\, \varepsilon\; C\; C'\; F.\; H{\downarrow}\varepsilon = (C,F) \wedge \\ F{\downarrow}f = \gamma \wedge h{\downarrow}(I\varepsilon) = (C',A) \end{array}\right\}$$

where $X_{H,I,\gamma} = \begin{cases} \{I\gamma\} & \text{if } \gamma \in \mathit{dom}\ H \\ \emptyset & \text{otherwise.} \end{cases}$

Definition 2. *(Interpretation of abstract states) Concrete state $s = (O, \sigma, h)$ satisfies abstract state $\Sigma = (S, \Gamma, H)$ with respect to interpretation I and map N, notation $s \models^I_N \Sigma$, if*

- *Σ is closed with respect to N*
- *O and S are of identical length, dom $\Gamma \subseteq$ dom σ, and for all $H{\downarrow}\gamma = (C, F)$ there are $C' \leq C$ and A such that $h{\downarrow}(I\gamma) = (C', A)$ and dom $F \subseteq$ dom A*
- *for all $\gamma \in Dom(\Sigma)$, $[\![\gamma]\!]^s_{\Sigma,I}$ is a singleton set,*
- *$Dom(\Sigma) \cap ODom(N) \subseteq$ dom I, and*
- *I is injective on dom H.*

This interpretation admits the concrete heap h to contain objects not tracked by H, but the injectivity condition for I enforces that distinct abstract objects are interpreted as distinct concrete objects.

Example. For $I(■) = a_1$ and $I(□) = a_2$ and distinct addresses $a_1, \ldots, a_3$, the concrete state s shown in Figure 10 satisfies the abstract state Σ from Figure 9, provided that $C_2 \leq C_1$ and $D_2 \leq D_1$. In addition to the entities tracked by Σ, the concrete state contains an additional variable z, an additional object at location a_3, and an additional field G_3 in the object at a_1. The two integer colours ○ and ● happen to be interpreted by the same number.

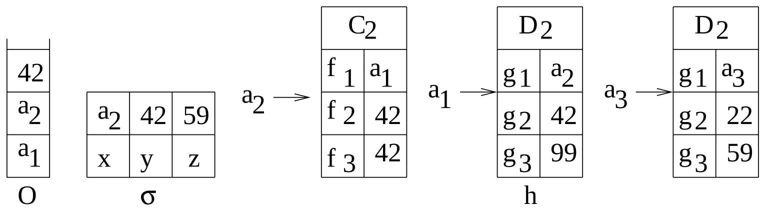

Figure 10: A concrete state $s = (O, \sigma, h)$ satisfying the abstract state Σ shown in Figure 9

We let $H * K$ denote the union of abstract heaps of distinct domains, and $N * M$ the union of administrative maps of distinct domains. These uses of the (overloaded) operator $*$ are treated commutatively and associatively. For $\Sigma = (S, \Gamma, H)$ we write $\Sigma * K$ for $(S, \Gamma, K * H)$.

3.2. Judgements and proof rules

The unary proof system is built from two judgement forms, namely a small-step judgement form, $U \vdash \ell : \Sigma, N \Rightarrow \Pi, M, L$, and a big-step judgement form, $U \vdash \ell : \Sigma, N \Downarrow H, \gamma, M$. Here, Σ and Π range over (not necessarily closed) abstract states, ℓ over labels, H over abstract heaps, L over sets of labels, and N and M over administrative maps. Furthermore, U ranges over (unary) proof contexts, which are sets of entries of the form $(\ell_i, (\Sigma_i, N_i, H_i, \gamma_i, M_i))$.

The separation into two judgement forms follows the structure of the operational semantics, i.e. the two forms are defined mutually recursively and apply to single instructions and method bodies in a similar way as the operational rules.

A first intuitive reading of a small-step judgement $U \vdash \ell : \Sigma, N \Rightarrow \Pi, M, L$ is that whenever $\mathcal{P} \vdash s, \ell \rightarrow t, \ell'$ and s satisfies Σ, then t satisfies Π and $\ell' \in L$. Similarly, a judgement $U \vdash \ell : \Sigma, N \Downarrow H, \gamma, M$ implies that for $\mathcal{P} \vdash s, \ell \Downarrow h, v$ with s satisfying Σ, $([v], [\,], h)$ satisfies $([\gamma], [\,], H)$. Additionally, the terminal administrative components M in both judgements contain the type information of any colours introduced by the subject phrases, and are always an extension of the initial maps N. The formal definition of the interpretation includes the Kripke-style extension mentioned in the introduction and will be given once the proof rules have been introduced, in Section 3.4.

3.2.1. Syntax-directed proof rules

Figure 11 presents a representative selection of the syntax-directed unary proof rules. The first rule, UniInstr treats basic instruction forms. The cases for load x, store x, getf $C.f$, and putf $C.f$ merely transfer abstract entities between the various state components. The next three instruction forms introduce fresh colours, which are inserted into the administrative map N with the appropriate type. Note that while the rules for instruction forms const i and new C look similar, their effect is slightly different, since object colours are interpreted in a separating fashion. Indeed, the freshness

ι	$S(\iota)$	$N'(\iota)$	$\Pi(\iota)$	$\Phi(\iota)$
load x	S	N	$(\gamma :: S, \Gamma, H)$	$\Gamma{\downarrow}x = \gamma$
store x	$\gamma :: S'$	N	$(S', \Gamma[x \mapsto \gamma], H)$	
getf $C.f$	$\gamma_1 :: S'$	N	$(\gamma_2 :: S', \Gamma, H)$	$\begin{cases} H{\downarrow}\gamma_1 = (D, F) \\ F{\downarrow}f = \gamma_2 \end{cases}$
putf $C.f$	$\gamma_1 :: \gamma_2 :: S'$	N	(S', Γ, K)	$\begin{cases} H{\downarrow}\gamma_2 = (D, F) \\ F' = F[f \mapsto \gamma_1] \\ K = H[\gamma_2 \mapsto (D, F')] \end{cases}$
const i	S	$N[\gamma \mapsto \mathsf{int}]$	$(\gamma :: S, \Gamma, H)$	$\gamma \notin \mathit{dom}\, N$
binop $\oplus$	$\gamma_1 :: \gamma_2 :: S'$	$N[\gamma \mapsto \mathsf{int}]$	$(\gamma :: S', \Gamma, H)$	$\begin{cases} \gamma \notin \mathit{dom}\, N \\ N{\downarrow}\gamma_i = \mathsf{int} \end{cases}$
new C	S	$N[\gamma \mapsto tp]$	$(\gamma :: S, \Gamma, K)$	$\begin{cases} \gamma \notin \mathit{dom}\, N,\ C \leq D \\ tp = \mathsf{class}(D) \\ K = H[\gamma \mapsto (C, [\,])] \end{cases}$
ifeq ℓ'	$\gamma :: S'$	N	(S', Γ, H)	
goto ℓ'	S	N	(S, Γ, H)	

$$\textsc{UniInstr}\ \frac{\mathcal{P}(\ell) = \iota \quad \Phi(\iota) \quad L(\ell, \iota) = \begin{cases} \{\ell'\} & \text{if } \iota = \mathsf{goto}\ \ell' \\ \{\ell + 1, \ell'\} & \text{if } \iota = \mathsf{ifeq}\ \ell' \\ \{\ell + 1\} & \text{otherwise} \end{cases}}{U \vdash \ell : (S(\iota), \Gamma, H), N \Rightarrow \Pi(\iota), N'(\iota), L(\ell, \iota)}$$

$$\textsc{UniInvV}\ \frac{\begin{array}{c} \mathcal{P}(\ell) = \mathsf{invVirt}\ C.m \qquad \mathit{params}(C, m) = [x_1, \ldots, x_n] \qquad H{\downarrow}\gamma_0 = (D, F) \\ \forall\ C' \leq D.\ U \vdash (C', m, 0) : ([], [x_i \mapsto \gamma_i, \mathsf{this} \mapsto \gamma_0], H), N \Downarrow K, \gamma, N' \end{array}}{U \vdash \ell : ([\gamma_1, \ldots, \gamma_n, \gamma_0]@S, \Gamma, H), N \Rightarrow (\gamma :: S, \Gamma, K), N', \{\ell + 1\}}$$

$$\textsc{UniVret}\ \frac{\mathcal{P}(\ell) = \mathsf{vreturn}}{U \vdash \ell : (\gamma :: S, \Gamma, H), N \Downarrow H, \gamma, N}$$

Figure 11: Unary proof system: selected syntax-directed rules

$$\textsc{UniRun}\ \frac{U \vdash \ell : \Sigma, N \Rightarrow \Pi, N', L \qquad \forall \ell' \in L.\ U \vdash \ell' : \Pi, N' \Downarrow H, \gamma, M}{U \vdash \ell : \Sigma, N \Downarrow H, \gamma, M}$$

$$\textsc{UniAx}\ \frac{(\ell, (\Sigma, N, H, \gamma, M)) \in U}{U \vdash \ell : \Sigma, N \Downarrow H, \gamma, M}$$

$$\textsc{UniBSFrame}\ \frac{U \vdash \ell : \Sigma, N_1 \Downarrow K, \gamma, M}{U \vdash \ell : \Sigma * H, N_1 * N \Downarrow K * H, \gamma, M * N}$$

$$\textsc{UniSSFrame}\ \frac{U \vdash \ell : \Sigma, N_1 \Rightarrow \Pi, M, L}{U \vdash \ell : \Sigma * H, N_1 * N \Rightarrow \Pi * H, M * N, L}$$

$$\textsc{UniRen}\ \frac{\begin{array}{c}(\Sigma, N) \xrightarrow{\varphi} (\Sigma_1, N_1) \qquad (\Pi, N') \xrightarrow{\xi} (\Pi_1, N'_1) \\ \forall\ \gamma\ \gamma' . \varphi(\gamma) = \gamma' \to \xi(\gamma) = \gamma' \qquad \mathit{injectiveOn}_{\mathit{dom}\ \xi \setminus \mathit{dom}\ \varphi}(\xi) \\ \mathit{ImgOf}(\xi, \mathit{dom}\ \xi \setminus \mathit{dom}\ \varphi) \cap \mathit{ImgOf}(\xi, \mathit{dom}\ \varphi) = \emptyset \\ U \vdash \ell : \Sigma, N \Rightarrow \Pi, N', L\end{array}}{U \vdash \ell : \Sigma_1, N_1 \Rightarrow \Pi_1, N'_1, L}$$

$$\textsc{GarbageHP}\ \frac{\begin{array}{c}U \vdash \ell : (S, \Gamma, H), N \Downarrow K, \gamma, N' \qquad \mathit{dom}\ H \subseteq \mathit{dom}\ K - \delta \qquad N \subseteq N' \\ \delta \notin \mathit{Dom}((S, \mathit{\Gamma}, H)) \cup \mathit{dom}\ N \cup \mathit{Dom}(([\gamma], [\,], K - \delta))\end{array}}{U \vdash \ell : (S, \Gamma, H), N \Downarrow K - \delta, \gamma, N'}$$

$$\textsc{GarbageN}\ \frac{\begin{array}{c}U \vdash \ell : \Sigma, N \Downarrow K, \gamma, N' \\ N \subseteq N' - \delta \qquad \delta \notin \mathit{Dom}(\mathit{\Sigma}) \cup \mathit{Dom}(([\gamma], [\,], K))\end{array}}{U \vdash \ell : \Sigma, N \Downarrow K, \gamma, N' - \delta}$$

Figure 12: Unary proof system: selected structural rules

of γ with respect to N in the rule for new C corresponds to the freshness of the associated runtime value (as a new object is allocated), while no such freshness condition is guaranteed to hold for the integer values. The rule for conditionals simply pops the topmost entry off the abstract operand stack. The rule for unconditional jumps is similar. In all rules, the component L collects the (static) control flow successors of the label in focus.

Rule UniInvV treats virtual method calls. The (abstract) receiver object is extracted from the abstract heap (determined by the number of formal parameters associated with the invoked method). Following the behavioural subtyping approach, we then require that any overriding method of $C.m$ in some subclass C' of the (abstract) receiver's class D satisfies the big-step specification given by the abstract reflection of argument and return value passing.

Neither the rules for field access nor UniInvV include the side condition $D \leq C$. As the operational judgement occurs negatively in the interpretation of proof rules (see Section 3.4), the side conditions of the operational rules guarantee that the expected subclass relationships hold whenever an execution step succeeds.

Finally, the rule UniVret describes the behaviour of a return instruction. We omit the rules for static methods and further basic instructions (pop, dup, etc.).

3.2.2. Structural proof rules

Figure 12 presents selected structural rules – the rule set is necessarily incomplete due to the restriction to static verification technology.

Rule UniRun injects small-step typing judgements into the big-step judgement, similar to the operational rule Run. The hypotheses require that Π, N' is valid at all one-step control flow successors ℓ' of ℓ, and that the phrases at these program points all satisfy $U \vdash \ell' : \Pi, N' \Downarrow H, \gamma, M$.

Rule UniAx extracts an assumption from the context, similar to an axiom in Hoare logic. As proof contexts are arbitrary sets, polyvariance is obtained by associating multiple specifications with a single label.

Frame rules are given for both judgement forms, and allow us to embed a judgement in a context with additional abstract objects and/or colours. The frame rule of separation logic [12] imposes a side condition on formulae that are framed onto the hypothetical judgement. This side condition requires that no free variable of such a formula is modified by the program phrase. The objects framed onto judgements in our case are not arbitrary formulae but only abstract heaps and administrative maps and do thus not contain free variables or other items that could be directly modified by the program. Thus, we do not need to impose additional side conditions. We will see applications of the frame rules later, in the verification of heap-allocated data structures in Section 5.

Rule UniRen allows us to recolour judgements according to (type-preserving) functions φ and ξ. The side conditions are defined and motivated as follows.

$(\Sigma, N) \xrightarrow{\varphi} (\Sigma_1, N_1)$ indicates that Σ_1 and N_1 are, respectively, the images of Σ and N under φ. We require that φ is injective on $ODom(N)$, so that distinctness of objects is preserved, and that φ respects N, i.e. that $\varphi(\gamma) = \varphi(\delta)$ implies $N{\downarrow}\gamma = N{\downarrow}\delta$. Similar conditions apply to $(\Pi, N') \xrightarrow{\xi} (\Pi_1, N'_1)$.

$\forall\, \gamma\, \gamma'.\varphi(\gamma) = \gamma' \to \xi(\gamma) = \gamma'$ guarantees that ξ extends φ, i.e. that the recolouring of the final state respects the recolouring of the initial state.

$injectiveOn_{dom\ \xi \setminus dom\ \varphi}(\xi)$ stipulates that colours not present in the initial state can only be recoloured injectively. For example, suppose Π and N' include two colours $\gamma \neq \delta$ not present in Σ and N, with $N'{\downarrow}\gamma = N'{\downarrow}\delta = \mathsf{int}$. The corresponding positions in a concrete state satisfying Π thus contain integer values which the derivation $U \vdash \ell : \Sigma, N \Rightarrow \Pi, N', L$ could not guarantee to be identical. The condition $\xi(\gamma) \neq \xi(\delta)$ ensures that this potential non-equality is also captured by Π_1.

$ImgOf(\xi, dom\ \xi \setminus dom\ \varphi) \cap ImgOf(\xi, dom\ \varphi) = \emptyset$ prevents colours present in (Π, N') but not in (Σ, N) to be identified by ξ with colours in (Σ, N). For example, a colour $\gamma \notin dom\ N \cup ODom(\Sigma)$ with $\gamma \in ODom(\Pi)$ represents an object freshly allocated by the phrase located at ℓ, and should thus not be equated with previously allocated objects.

As an example, consider the situation in Figure 13 (top). The initial abstract state contains one object and two integer colours. The final state contains two further integer

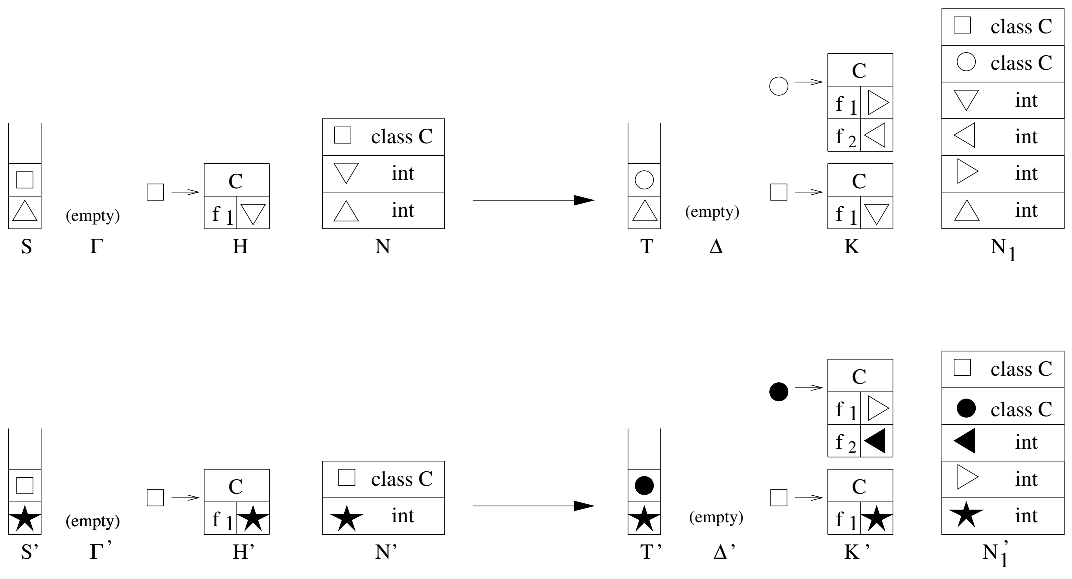

Figure 13: Rule UniRen: example

colours, and a further object. The following application of rule UniRen justifies the more restrictive judgement in Figure 13 (bottom). The renaming function for the initial state, φ, identifies $\triangle$ and $\triangledown$ as $\bigstar$, and acts as the identity map on $\square$. The renaming function for the terminal state, ξ, extends φ by additionally mapping $\circ$ to $\bullet$ and $\lhd$ to $\blacktriangleleft$. It is easily verified that all side conditions of the rule are satisfied. In particular, the colours that are present in $((T,\Delta,K),N_1)$ but not in $((S,\Gamma,H),N)$, namely $\circ$, $\lhd$ and $\rhd$, are mapped injectively and to colours distinct from the image of φ.

By admitting renamings that are not entirely injective, the renaming rule thus incorporates aspects of a subtyping rule in type systems, or of a rule of consequence in program logics, albeit on the level of judgements.

The two final rules, GarbageHP and GarbageN, remove unreachable objects and unused colours from the final abstract states of judgements, using the operation $. - .$ discussed in Section 2. The other side conditions ensure that only colours allocated by the subject phrase may be garbage-collected, in order not to violate the preservation of colours mandated by the interpretation of judgements (see below, Definition 5). As an example consider the judgement depicted in Figure 14 (top), which indicates that a phrase allocates a new object $\circ$ with a possibly fresh integer value in field g_2. As the object colour is unreachable from the rest of the terminal state, rule UniGarbageHP applies, and the object may be deleted from the abstract state (second row). Its colour, and the integer colour $\triangledown$, are subsequently removed in the final two rows. All deletions are possible as the deleted objects and colours do not occur in the initial state.

Definition 3. *We call a derivation for the big-step judgement form* progressive *if it*

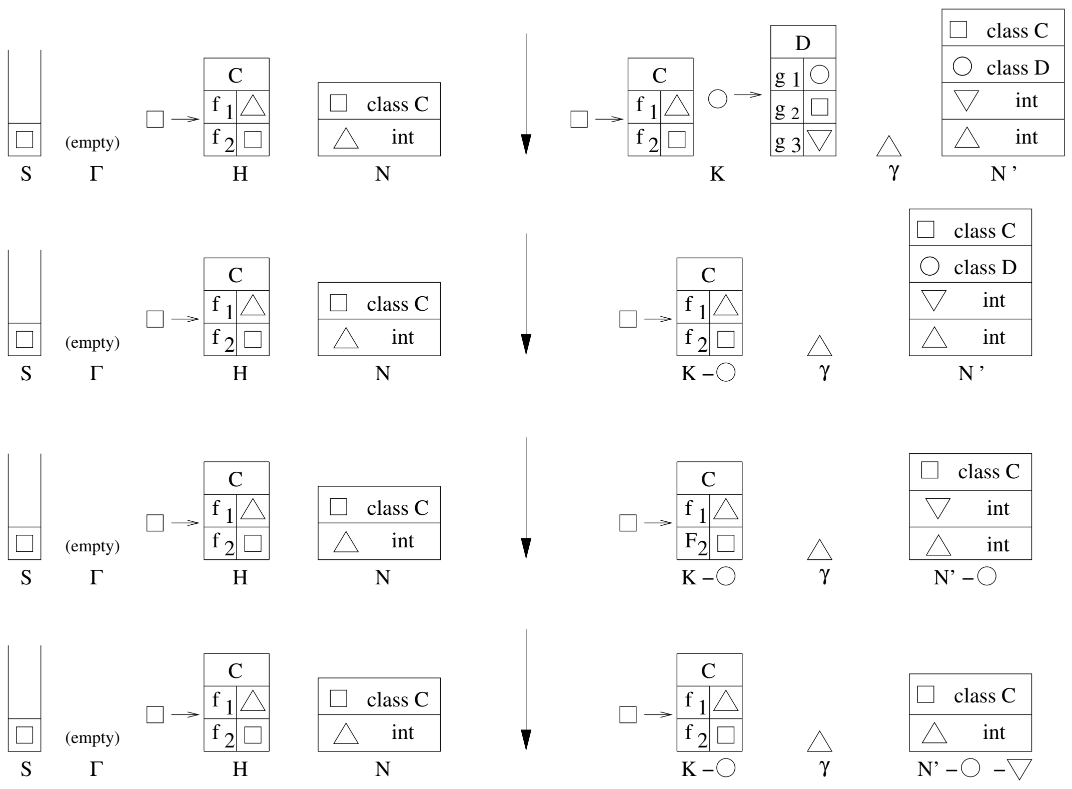

Figure 14: Rules UNIGARBAGEHP and UNIGARBAGEN : example

```
int C.m1(int y){
   [0] load y       [2] load this      [4] putf C.A
   [1] dup          [3] swap           [5] vreturn
}
```

Figure 15: Bytecode for method `C.m1` from Figure 1

contains at least one application of rule UniVret *or rule* UniRun. *Context U is* verified*, notation $\vdash U$, if for each $(\ell, (\Sigma, N, H, \gamma, M)) \in U$ there is a progressive derivation with final sequent $U \vdash \ell : \Sigma, N \Downarrow H, \gamma, M$.*

3.3. Example verification

Consider the bytecode in Figure 15, which represents the result of compiling method `C.m1` from Section 1.1, Figure 1. We verify that this code satisfies the expected specification relating the final content of field `A` to the argument `y`. To that effect we show that $\emptyset \vdash (\mathtt{C}, \mathtt{m1}, \mathbb{0}) : \Sigma, N \Downarrow H, \delta, N$ is derivable for

$$\begin{array}{rcl} \Sigma &=& ([\,], [\mathsf{this} \mapsto \gamma, \mathsf{y} \mapsto \delta], [\gamma \mapsto (\mathtt{C}, F)]), \\ H &=& [\gamma \mapsto (\mathtt{C}, F[\mathtt{A} \mapsto \delta])], \\ N &=& [\delta \mapsto \mathsf{int}] * [\gamma \mapsto \mathsf{class}(\mathtt{C})], \end{array}$$

and arbitrary[3] F. The proof proceeds essentially in a syntax-directed fashion: the list of rule applications is UniRun, UniLoad, UniRun, UniDup, UniRun, UniLoad, UniRun, UniSwap, UniRun, UniPutf, UniVret. All side conditions are easily discharged (we silently use the assumption $\mathsf{this} \neq \mathsf{y}$).

3.4. Interpretation and Soundness

We first give the formal definition of the non-Kripke-extended interpretation of judgements mentioned at the beginning of Section 3.2.

Definition 4. *Label ℓ* conforms to specification *Σ, N, Π, M, notation $\models \Sigma, N \xrightarrow{\ell} \Pi, M$, if for all s, I, h, and v with $s \models_N^I \Sigma$ and $\mathcal{P} \vdash s, \ell \Downarrow h, v$, there is some J with $([v], [\,], h) \models_M^J \Pi$ such that*

a) *for all $\gamma \in ODom(N)$, $J\gamma = I\gamma$ holds, and*
b) *for all $\gamma \in dom\ N$, $[\![\gamma]\!]_{\Pi,J}^{([v],[\,],h)} \subseteq [\![\gamma]\!]_{\Sigma,I}^{s}$.*

This definition requires that any terminating execution commencing in some state s that satisfies Σ (with respect to N and I) leads to a terminal state that satisfies Π (with respect to M and J), such that the interpretation of each colour γ defined in the initial administrative map N is preserved. In particular,

[3]This is a *local* specification in the sense of [12] as no additional heap objects are specified.

a) the interpretation J is an extension of I on object colours (hence the tracked objects remain in place),
b) the interpretation of γ in the terminal state $([v], [\,], h)$ (with respect to Π and J) is contained in the interpretation of γ in the initial state s (with respect to Σ and I). If $\gamma \in Dom(\Sigma)$ holds, then the set $[\![\gamma]\!]^{s}_{\Sigma,I}$ is in fact a singleton set, by the definition of $s \models^{I}_{N} \Sigma$. Consequently, the set $[\![\gamma]\!]^{([v],[\,],h)}_{\Pi,J}$ is in this case either the same singleton set (hence the represented values are identical) or is empty (if $\gamma \notin Dom(\Pi)$). If $\gamma \notin Dom(\Sigma)$ holds, then $[\![\gamma]\!]^{s}_{\Sigma,I} = \emptyset$ follows, hence $[\![\gamma]\!]^{([v],[\,],h)}_{\Pi,J} = \emptyset$. The condition $[\![\gamma]\!]^{([v],[\,],h)}_{\Pi,J} \subseteq [\![\gamma]\!]^{s}_{\Sigma,I}$ thus relates the interpretation of γ in the final state of a judgement to the interpretations of γ in the initial state. This enables the tracking of copies across method calls.

Note that these conditions only concern colours present in the initial map N. Also note that $\models \Sigma, N \xrightarrow{\ell} \Pi, M$ trivially holds if Σ fails to be closed w.r.t. N, as $s \models^{I}_{N} \Sigma$ would be violated, cf. Definition 2. Furthermore, if Σ *is* closed w.r.t. N (and ℓ terminates) then Π is closed w.r.t. M as well, because of $([v], [\,], h) \models^{J}_{M} \Pi$.

The full interpretation of a big-step judgement now requires conformance to all specifications that arise as *separated extensions* of the abstract states occurring in the judgement, and includes syntactic constraints that enforce the preservation of abstract entities.

Definition 5. *Label* ℓ is sound for (Σ, N) and (H, γ, M)*, notation*

$$\models \ell : \Sigma, N \Downarrow H, \gamma, M$$

if for $\Sigma = (S, \Gamma, K)$*, the following three properties hold*

1. *dom* $K \subseteq$ *dom* H
2. M *contains* N*, and*
3. *for all* H' *and* N'*,* $\models \Sigma * H', N * N' \xrightarrow{\ell} ([\gamma], [\,], H) * H', M * N'$.

The first two conditions are syntactic and guarantee that abstract colours and objects are (in a type-respecting fashion) preserved. The third condition represents the semantic guarantee and asserts conformance to all specifications that arise by framing some H' onto the pre-and post-heaps, and some N' onto the administrative maps. The implicit side condition of operator $*$ ensures *dom* $H' \cap$ *dom* $H = \emptyset$ and *dom* $N' \cap$ *dom* $M = \emptyset$, i.e. only distinct heaps and administrative maps are framed onto the entities mentioned explicitly in judgements. In agreement with the syntactic conditions on judgements, Σ and $([\gamma], [\,], H)$ are not required to be closed w.r.t. N and M, respectively. However, in combination with Definitions 4 and 2, the third clause makes a non-trivial claim only for those H' and N' that make $\Sigma * H'$ and $([\gamma], [\,], H) * H'$ closed w.r.t. $N * N'$ and $M * N'$, respectively.

The interpretation of small-step unary judgements is similar and hence omitted. The following theorem establishes the soundness of the unary proof system.

Theorem 1. *Let* $\vdash U$ *and* $U \vdash \ell : \Sigma, N \Downarrow H, \gamma, M$*. Then* $\models \ell : \Sigma, N \Downarrow H, \gamma, M$*.*

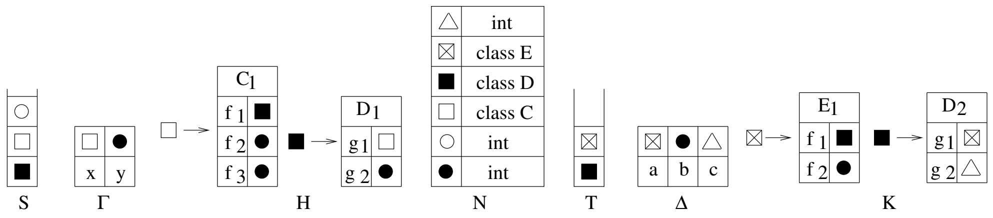

Figure 16: An example RSD

The proof of this theorem employs a relativised notion of soundness that bounds the derivation height of operational judgements. In particular, an auxiliary lemma establishes relativised soundness by induction on the proof rules, in the style of formalised soundness proofs of Hoare logics as presented by Kleymann and Nipkow [26, 27]. However, due to the mutually recursive dependence, the auxiliary lemma is performed by a joint induction on the two judgement forms.

The requirement that derivations justifying entries of verified contexts contain at least one progressive rule forces these justifications to actually inspect their subject code block. In particular, a justification that establishes a context entry for a label that represents a loop header by immediately applying the axiom rule is not permitted. This technique is an adaptation of soundness proofs for proof rules for recursive methods [27] to low-level languages and was first presented in [28].

For the details of the proof, the interested reader is referred to [11].

4. Relational proof system

We now turn to the relational system, the core contribution of our article.

4.1. Relational state descriptions

Definition 6. *A relational state description (RSD) is a structure $\phi = (\Sigma, N, \Pi)$ where Σ and Π are abstract states and N is an administrative map. ϕ is closed if Σ and Π are closed with respect to N.*

Example. Figure 16 extends the abstract state Σ from Figure 9 to the RSD $\phi = (\Sigma, N, \Pi)$ by defining abstract state $\Pi = (T, \Delta, K)$ and extending the administrative map N appropriately. Colours ■ and ● occur in both abstract states. The two operand stacks are of different height, and the abstract stores refer to different variables. RSD ϕ is closed if $D_2 \leq D$ and $E_1 \leq E$ hold, in addition to the earlier constraints $C_1 \leq C$ and $D_1 \leq D$.

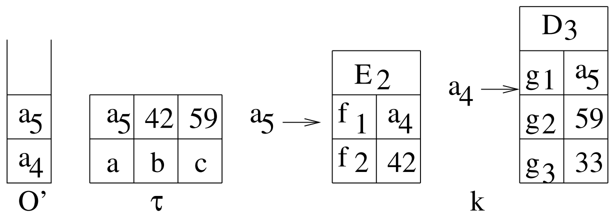

Figure 17: A concrete state $t = (O', \tau, k)$ satisfying the abstract state Π from Figure 16

For $\phi = (\Sigma, M, \Pi)$ we write $\phi * (H, N, K)$ for $(\Sigma * H, M * N, \Pi * K)$.

RSD are interpreted over state pairs, relative to functions I and J that interpret the object colours in ϕ as concrete addresses.

Definition 7. *(RSD interpretation) A pair s, t of states satisfies RSD $\phi = (\Sigma, N, \Pi)$ with respect to interpreting functions I and J, notation $(s, t) \models_{I,J} \phi$, if*

- *$s \models^I_N \Sigma$ and $t \models^J_N \Pi$, and*
- *for all $\gamma \in Dom(\Sigma) \cap Dom(\Pi)$, $N{\downarrow}\gamma = \mathsf{int}$ implies $[\![\gamma]\!]^s_{\Sigma,I} = [\![\gamma]\!]^t_{\Pi,J}$.*

The first condition in this definition simply requires the two states to satisfy their abstract counterparts. The second condition requires for any integer colour γ occurring in both Σ and Π, that the runtime values interpreting γ in s and t (each such value being uniquely determined by the first condition) should be equal. Definition 7 thus defines the relation paraphrased as $s =_\phi t$ in the introduction.

Example. Continuing our example, state $t = (O', \tau, k)$ in Figure 17 satisfies abstract state Π and N from Figure 16 with respect to an interpreting function J with $J(\blacksquare) = a_4$ and $J(\boxtimes) = a_5$ for $a_4 \neq a_5$, where $D_3 \leq D_2$ and $E_2 \leq E_1$. Again, the concrete state contains a field that is not tracked by the abstract state.

Together with state s from Figure 10 and ϕ as in Figure 16, we have $(s, t) \models_{I,J} \phi$: the shared integer colour ● is interpreted by the same number in both states.

The reader who is familiar with the work of Banerjee-Naumann [23] and Barthe et al. [6] may have expected Definition 7 to include some explicit partial bijection β on addresses, capturing when objects in the heaps of s and t are indistinguishable. For example, one might have expected a further condition like

for all $\gamma \in Dom(\Sigma) \cap Dom(\Pi)$, and all $C, N{\downarrow}\gamma = \mathsf{class}(C)$ implies $h(a) \sim_\beta h'(a')$

where $a = [\![\gamma]\!]^s_{\Sigma,I}$, $a' = [\![\gamma]\!]^t_{\Pi,J}$, and h and h' are the heaps of s and t, respectively. In the cited papers, object indistinguishability $o \sim_\beta o'$ requires the objects o and o' to be of a subclass of C and to contain indistinguishable values in all visible fields, where value indistinguishability is given by containment in β on address values and by equality on other values. This discipline is enforced in the rules of [23] and [6] by only allowing low values to be written into such fields.

Given an RSD $\phi = (\Sigma, N, \Pi)$ and its interpretation (I, J), it is easy to see that

$$\beta = \bigcup_{\gamma \in dom\ H \cap dom\ K} (I\gamma, J\gamma)$$

where $\Sigma = (S, \Gamma, H)$ and $\Pi = (T, \Delta, K)$ is indeed a partial bijection. This relation plays a similar role as the partial bijections of Banerjee-Naumann and Barthe et al. in that it captures when objects may be considered correlated. However, the fact that β is determined by ϕ and (I, J) means that there is no need for us to introduce it formally. In our example, the induced partial bijection is the singleton $\{(a_1, a_4)\}$, as these locations are the images of the shared colour ■ under I and J, respectively.

As a further difference, the interpretation of RSD's does not require objects residing at related addresses to be low equivalent, i.e. to contain indistinguishable values in public fields. As we do not require fields to be classified upfront according to their security level, there is no need to define an explicit notion of object indistinguishability. Instead, RSD's track the content of fields in a more fine-grained fashion, and our proof rules do not constrain field assignments according to the visibility of the assigned value.

If, however, a classification of fields happens to be applicable, we can stipulate that RSD $\phi = (\Sigma, N, \Sigma')$ satisfies object indistinguishability by defining

$$\begin{aligned} H \sim_N^{\mathcal{G}} H' = \forall\ \gamma\ C\ D\ F\ D'\ F'.\ & (N{\downarrow}\gamma = \mathsf{class}(C) \& H{\downarrow}\gamma = (D, F) \& H'{\downarrow}\gamma = (D', F')) \\ & \Longrightarrow \forall\ f \in \mathcal{G}.\ F{\downarrow}f = F'{\downarrow}f \end{aligned}$$

and then requiring that for $\Sigma = (S, \Gamma, H)$ and $\Sigma' = (S', \Gamma', H')$, $H \sim_N^{\mathcal{G}} H'$ holds, where $\mathcal{G}$ contains the field names (statically) considered public.

Thus, RSD's offer a mechanism to talk about public and private variables or fields without having to introduce security levels formally: for example, an integer variable that is associated with an identical colour in both abstract states amounts to the variable being low at the pair of program points where the RSD is valid. In contrast, if this variable is associated with different colours in the two abstract stores then neither equality nor non-equality of the corresponding runtime values are guaranteed. The latter situation thus amounts to the variable being private at such a pair of program points. Of course, these two cases are only special cases as a variable may behave differently at different pairs of program points, and the two abstract states of an RSD are not required to be formulated with respect to the same sets of variables. In this sense, RSD's are also more flexible than assertions of the form $x\ltimes$ that appear in Amtoft et al.'s logic [14]. On the other hand, the generalisation of these assertions to *agreements* by Banerjee and Naumann [29] is not presently supported by RSD's, as abstract locations in our case represent single locations rather than arbitrary regions.

4.2. Judgements and proof rules

The relational system employs a single judgement form, $G \vdash \ell \sim \ell' : \phi \to \psi$. Here, ϕ and ψ are RSD's, ℓ and ℓ' are program labels, and G is a relational proof context with entries of the form $((\ell_i, \ell'_i), (\phi_i, \psi_i))$. These judgements are interpreted in a big-step fashion: a pair of terminal states is expected to satisfy ψ whenever the pair of initial

$$\textsc{UniL}\ \dfrac{\begin{array}{c} U \vdash \ell : \Sigma, N \Rightarrow \Sigma_1, N_1, L \qquad \vdash U \\ \forall\, \ell_1 \in L.\, G \vdash \ell_1 \sim \ell' : (\Sigma_1, N_1, \Pi) \to \psi \end{array}}{G \vdash \ell \sim \ell' : (\Sigma, N, \Pi) \to \psi}$$

$$\textsc{ConstConst}\ \dfrac{\begin{array}{c} \mathcal{P}(\ell) = \mathsf{const}\ i \quad \mathcal{P}(\ell') = \mathsf{const}\ i \quad \gamma \notin dom\ N \\ \phi = ((\gamma :: S, \Gamma, H), N[\gamma \mapsto \mathsf{int}], (\gamma :: T, \Delta, K)) \\ G \vdash \ell + 1 \sim \ell' + 1 : \phi \to \psi \end{array}}{G \vdash \ell \sim \ell' : ((S, \Gamma, H), N, (T, \Delta, K)) \to \psi}$$

$$\textsc{BinBin}\ \dfrac{\begin{array}{c} \mathcal{P}(\ell) = \mathsf{binop} \oplus \qquad \mathcal{P}(\ell') = \mathsf{binop} \oplus \\ \gamma \notin dom\ N \qquad N{\downarrow}\gamma_i = \mathsf{int} \qquad M = N[\gamma \mapsto \mathsf{int}] \\ \phi = ((\gamma :: S, \Gamma, H), M, (\gamma :: T, \Delta, K)) \\ G \vdash \ell + 1 \sim \ell' + 1 : \phi \to \psi \\ \xi = ((\gamma_1 :: \gamma_2 :: S, \Gamma, H), N, (\gamma_1 :: \gamma_2 :: T, \Delta, K)) \end{array}}{G \vdash \ell \sim \ell' : \xi \to \psi}$$

$$\textsc{NewNew}\ \dfrac{\begin{array}{c} \mathcal{P}(\ell) = \mathsf{new}\ C_1 \qquad \mathcal{P}(\ell') = \mathsf{new}\ C_2 \qquad \gamma \notin dom\ N \\ C_1 \le C \qquad C_2 \le C \qquad N' = N[\gamma \mapsto \mathsf{class}(C)] \\ H' = H[\gamma \mapsto (C_1, [\,])] \qquad K' = K[\gamma \mapsto (C_2, [\,])] \\ \phi = ((\gamma :: S, \Gamma, H'), N', (\gamma :: T, \Delta, K')) \\ G \vdash \ell + 1 \sim \ell' + 1 : \phi \to \psi \end{array}}{G \vdash \ell \sim \ell' : ((S, \Gamma, H), N, (T, \Delta, K)) \to \psi}$$

$$\textsc{IfIf}\ \dfrac{\begin{array}{c} \mathcal{P}(\ell) = \mathsf{ifeq}\ \ell_1 \quad \mathcal{P}(\ell') = \mathsf{ifeq}\ \ell'_1 \quad N{\downarrow}\gamma = \mathsf{int} \\ G \vdash \ell + 1 \sim \ell' + 1 : ((S, \Gamma, H), N, (T, \Delta, K)) \to \psi \\ G \vdash \ell_1 \sim \ell'_1 : ((S, \Gamma, H), N, (T, \Delta, K)) \to \psi \end{array}}{G \vdash \ell \sim \ell' : ((\gamma :: S, \Gamma, H), N, (\gamma :: T, \Delta, K)) \to \psi}$$

$$\textsc{RetRet}\ \dfrac{\begin{array}{c} \mathcal{P}(\ell) = \mathsf{vreturn} \qquad \mathcal{P}(\ell') = \mathsf{vreturn} \\ \psi = (([\gamma], [\,], H), N, ([\gamma'], [\,], K)) \end{array}}{G \vdash \ell \sim \ell' : ((\gamma :: S, \Gamma, H), N, (\gamma' :: T, \Delta, K)) \to \psi}$$

Figure 18: Relational proof system: syntax-directed rules I

states satisfied ϕ. In particular, the interpretation captures termination-insensitive non-interference, i.e. the judgement is vacuously valid if either of the two executions fails to terminate. Again, the formal interpretation of judgements is postponed until the rules and some examples have been given.

Figures 18 to 20 present an excerpt of the relational proof system. The first rule of Figure 18 injects the small-step unary proof system into the left component of the relational proof system. Thus, operations that are not correlated between the two executions are treated independently. Note that this includes one-sided object allocations, method invocations, and conditionals. The second hypothesis requires us to prove a specification for code pairs comprising a control flow successor of ℓ in the left component, and the unmodified label ℓ' in the right component. The specification for the label pair (ℓ_1, ℓ') is given by updating the initial RSD according to the effect of the instruction at ℓ. A similar rule for injecting unary derivations into the right component is omitted.

The rules ConstConst and NewNew capture correlated executions of instruction forms $\mathsf{const}\ i$ and $\mathsf{new}\ C$. Again, these are modelled by choosing fresh colours, which

$$\textsc{InvVInvV}\ \frac{\begin{array}{c}
\mathcal{P}(\ell) = \mathsf{invVirt}\ C_1.m_1 \qquad \mathcal{P}(\ell') = \mathsf{invVirt}\ C_2.m_2 \\
\mathit{params}(C_1, m_1) = [x_1, \ldots, x_n] \qquad \mathit{params}(C_2, m_2) = [x'_1, \ldots, x'_m] \\
G \vdash \ell + 1 \sim \ell' + 1 : ((\gamma :: S, \Gamma, H'), N', (\gamma' :: T, \Delta, K')) \to \psi' \\
H{\downarrow}\gamma_0 = (D_1, F_1) \qquad D_1 \le C_1 \qquad K{\downarrow}\gamma'_0 = (D_2, F_2) \qquad D_2 \le C_2 \\
\psi = (([\gamma_1, \ldots, \gamma_n, \gamma_0]@S, \Gamma, H), N, ([\gamma'_1, \ldots, \gamma'_m, \gamma'_0]@T, \Delta, K)) \\
\phi = (([], [x_i \mapsto \gamma_i, \mathsf{this} \mapsto \gamma_0], H), N, ([], [x'_i \mapsto \gamma'_i, \mathsf{this} \mapsto \gamma'_0], K)) \\
\phi' = (([\gamma], [\,], H'), N', ([\gamma'], [\,], K')) \\
\forall D'_1 \le D_1.\ \forall D'_2 \le D_2.\ G \vdash (D'_1, m_1, 0) \sim (D'_2, m_2, 0) : \phi \to \phi'
\end{array}}{G \vdash \ell \sim \ell' : \psi \to \psi'}$$

$$\textsc{InvSInvS}\ \frac{\begin{array}{c}
\mathcal{P}(\ell) = \mathsf{invStat}\ C_1.m_1 \qquad \mathcal{P}(\ell') = \mathsf{invStat}\ C_2.m_2 \\
\mathit{params}(C_1, m_1) = [x_1, \ldots, x_n] \qquad \mathit{params}(C_2, m_2) = [x'_1, \ldots, x'_m] \\
G \vdash \ell + 1 \sim \ell' + 1 : ((\gamma :: S, \Gamma, H'), N', (\gamma' :: T, \Delta, K')) \to \psi \\
\phi = (([], [x_i \mapsto \gamma_i], H), N, ([], [x'_i \mapsto \gamma'_i], K)) \\
\phi' = (([\gamma], [\,], H'), N', ([\gamma'], [\,], K')) \\
G \vdash (C_1, m_1, 0) \sim (C, m_2, 0) : \phi \to \phi'
\end{array}}{G \vdash \ell \sim \ell' : (([\gamma_1, \ldots, \gamma_n]@S, \Gamma, H), N, ([\gamma'_1, \ldots, \gamma'_m, \gamma'_0]@T, \Delta, K)) \to \psi}$$

$$\textsc{InvSInvV}\ \frac{\begin{array}{c}
\mathcal{P}(\ell) = \mathsf{invStat}\ C_1.m_1 \qquad \mathcal{P}(\ell') = \mathsf{invVirt}\ C_2.m_2 \\
\mathit{params}(C_1, m_1) = [x_1, \ldots, x_n] \qquad \mathit{params}(C_2, m_2) = [x'_1, \ldots, x'_m] \\
G \vdash \ell + 1 \sim \ell' + 1 : ((\gamma :: S, \Gamma, H'), N', (\gamma' :: T, \Delta, K')) \to \psi \\
K{\downarrow}\gamma'_0 = (C, F) \qquad C \le C_2 \\
\phi = (([], [x_i \mapsto \gamma_i], H), N, ([], [x'_i \mapsto \gamma'_i, \mathsf{this} \mapsto \gamma'_0], K)) \\
\phi' = (([\gamma], [\,], H'), N', ([\gamma'], [\,], K')) \\
\forall C' \le C.\ G \vdash (C_1, m_1, 0) \sim (C', m_2, 0) : \phi \to \phi'
\end{array}}{G \vdash \ell \sim \ell' : (([\gamma_1, \ldots, \gamma_n]@S, \Gamma, H), N, ([\gamma'_1, \ldots, \gamma'_m, \gamma'_0]@T, \Delta, K)) \to \psi}$$

Figure 19: Relational proof system: methods invocation rules

are now inserted in both abstract states concurrently. In addition, ConstConst enforces the equality relation expected by the interpretation of RSD's. A similar side condition is not imposed in rule NewNew, since addresses are only related by the partial bijection that is implicitly determined by the interpretation of colours. If no correlation between the objects is desired, two applications of rule UniInstr (case New) may be used, one on each side. The class name C may be freely chosen subject to $C_1 \le C$ and $C_2 \le C$. Rule BinBin treats correlated operations on integers. Future work may generalise this rule so that algebraic properties of operators $\oplus$ may be exploited [20].

Rule IfIf is the rule for correlated conditionals, which applies if the top colours of the abstract operand stacks are identical. Under this condition, the two branches will evaluate identically at runtime. Hence, only two hypotheses are required, one for each possible outcome. Non-correlated branches may be verified using two applications of the unary rule for conditionals (one application on each side), resulting in subgoals corresponding to all four combinations of possible outcomes of the branch conditions. Note that the program labels ℓ and ℓ', and also the segments S and T of the abstract operand stacks, may well be different, resulting in additional flexibility compared to the traditional typing rule for low conditionals.

Rule RetRet models the effect of return instructions. The abstract colours that are returned need not be identical.

Figure 19 collects rules for correlated method invocations. The rule for correlated

$$\textsc{Ax}\ \frac{((\ell,\ell'),(\psi,\phi)) \in G}{G \vdash \ell \sim \ell' : \psi \to \phi} \qquad \textsc{Frame}\ \frac{G \vdash \ell \sim \ell' : \phi \to \psi}{G \vdash \ell \sim \ell' : \phi * (H,N,K) \to \psi * (H,N,K)}$$

$$\textsc{Ren}\ \frac{\begin{array}{c} G \vdash \ell \sim \ell' : \psi \to \phi \qquad \psi \xrightarrow{\varphi} \psi' \qquad \phi \xrightarrow{\xi} \phi' \\ \forall\, \gamma\, \gamma'.\varphi(\gamma) = \gamma' \to \xi(\gamma) = \gamma' \qquad \mathit{injectiveOn}_{dom\ \xi \setminus dom\ \varphi}(\xi) \\ \mathit{ImgOf}(\xi, dom\ \xi \setminus dom\ \varphi) \cap \mathit{ImgOf}(\xi, dom\ \varphi) = \emptyset \end{array}}{G \vdash \ell \sim \ell' : \psi' \to \phi'}$$

$$\textsc{UniUni}\ \frac{\begin{array}{c} dom\ N' \cap dom\ N'' = \emptyset \qquad \vdash U \qquad \vdash V \\ U \vdash \ell : \Sigma, N \Downarrow H, \gamma, (N * N') \qquad V \vdash \ell' : \Pi, N \Downarrow K, \delta, (N * N'') \end{array}}{G \vdash \ell \sim \ell' : (\Sigma, N, \Pi) \to (([\gamma], [\,], H), N * N' * N'', ([\delta], [\,], K))}$$

Figure 20: Relational proof system: structural rules (excerpt)

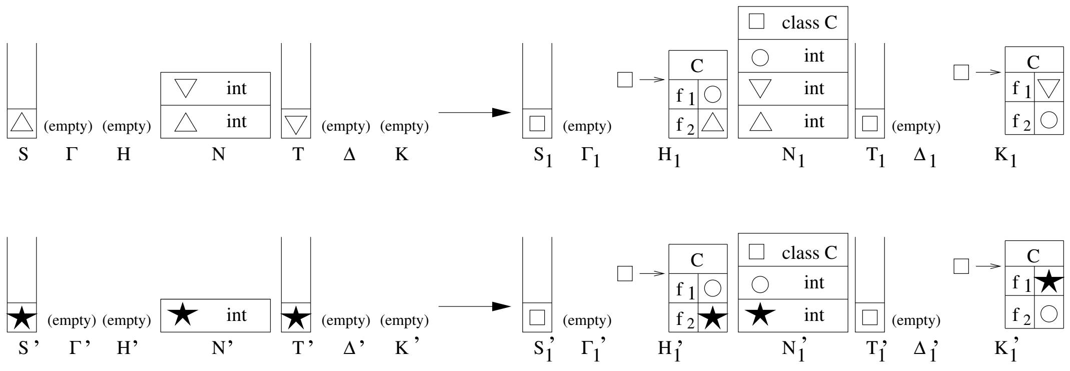

Figure 21: Rule Ren: example

calls of (not necessarily identical) virtual methods, InvVInvV, is similar to the unary rule UniInvV, but the hypothesis on (overriding definitions of) the invoked methods (i.e. the final hypothesis) exploits correlations between the method bodies. The rules for correlated static methods, and a correlation pair comprising a virtual and a static method, are defined in a similar way.

Selected structural rules are shown in Figure 20 and include an axiom rule, a frame rule, and a rule for renaming that is similar to its unary counterpart. The side condition $\psi \xrightarrow{\varphi} \psi'$ of the latter rule is defined by $(\Sigma, N) \xrightarrow{\varphi} (\Sigma', N')$ and $(\Pi, N) \xrightarrow{\varphi} (\Pi', N')$ where $\psi = (\Sigma, N, \Pi)$ and $\psi' = (\Sigma', N', \Pi')$, and similarly for the side condition $\phi \xrightarrow{\xi} \phi'$. The other side conditions are as in Section 3.2.2. An example for a relational renaming is shown in Figure 21. The two (integer) colours ▽ and △ are merged to ★. Continuing our comparison to traditional formalisms for non-interference we observe that this rule has a similar effect as the subtyping rule in the flow-sensitive system of Hunt and Sands [25]. The cited rule sanctions to lower the security types associated with variables in the initial variable environment. In line with the rough correspondence of non-correlated to *high*-security values and of correlated values to *low*-security values, a similar effect is exhibited by rule Ren. Values that are not required to be correlated

```
int C.m2(int l; int h){
   [0] load l        [6] load l         [12] vreturn        [18] dup
   [1] store x       [7] load x         [13] load x         [19] load x
   [2] load h        [8] binop plus     [14] const 3        [20] putf C.A
   [3] ifeq 13       [9] new C          [15] binop plus     [21] getf C.A
   [4] const 3       [10] swap          [16] store x        [22] vreturn
   [5] store x       [11] invVirt C.m1  [17] new C
}
```

Figure 22: Bytecode for method C.m2 from Figure 1

between the two executions in the hypothesis may be entangled by correlating them in the conclusion. Note that this operation is only possible for colours occurring already in the initial RSD. Indeed, merging colours freshly introduced in the final RSD would be unsound, as the conclusion would assert the equality of certain values that the hypothetical judgement could not show to be equal. As in rule UniRen, renaming affects the entire judgement, imposes distinctness and injectivity conditions on the recolouring of the final RSD, and does not permit to merge objects.

The final rule, UniUni, covers cases where the two subject phrases execute independently until the end of the current method frames, by injecting two big-step unary judgements. Colours that are freshly introduced by the two phrases must not overlap, and are merged with the colours already in use.

Definition 8. *A relational derivation is called* progressive *if it contains at least one application of a syntax-directed rule, and (relational) context G is* verified*, notation* $\vdash G$*, if each entry of G is justified by a progressive derivation.*

4.3. Example verifications

Continuing with the the example from Figure 1, we verify non-interference of method C.m2, the bytecode of which is shown in Figure 22. We derive the judgement $\emptyset \vdash (\mathtt{C}, \mathtt{m2}, \mathbb{0}) \sim (\mathtt{C}, \mathtt{m2}, \mathbb{0}) : \phi \to \psi$ where

$$\begin{aligned}
\phi &= (([\,], [\mathtt{l} \mapsto \gamma, \mathtt{h} \mapsto \delta], [\,]), N, ([\,], [\mathtt{l} \mapsto \gamma, \mathtt{h} \mapsto \varepsilon], [\,]))\\
N &= [\gamma \mapsto \mathsf{int}] * [\delta \mapsto \mathsf{int}] * [\varepsilon \mapsto \mathsf{int}]\\
\psi &= (([\beta], [\,], [\mu \mapsto (\mathtt{C}, [\mathtt{A} \mapsto \beta])]), M, ([\beta], [\,], [\nu \mapsto (\mathtt{C}, [\mathtt{A} \mapsto \beta])]))\\
M &= N * [\alpha \mapsto \mathsf{int}] * [\beta \mapsto \mathsf{int}] * [\mu \mapsto \mathsf{class}(\mathtt{C})] * [\nu \mapsto \mathsf{class}(\mathtt{C}))].
\end{aligned}$$

The proof applies unary rules on both sides (injected using rule UniL or its counterpart for the phrase on the right-hand side, UniR) until the conditional is met (label 3). The top elements of the abstract operand stacks at this point are δ and ε, hence rule IfIf is not applicable. We therefore apply the rule for unary conditionals twice (once on each side), leading to proof branches for the label pairs $(4, 4)$, $(4, 13)$, $(13, 4)$, and $(13, 13)$. All four branches proceed in a similar way. We apply rule ConstConst for the fresh colour α when arriving at an instruction pair $(\mathsf{const}\ 3, \mathsf{const}\ 3)$ (i.e. at label pairs $(4, 4)$, $(4, 14)$, $(14, 4)$, and $(14, 14)$), and rule BinBin for the fresh colour β when

```
int C.m4 (int l){
  [0] load l            [3] store x        [6] putf C.A
  [1] const 3           [4] new C          [7] load x
  [2] binop plus        [5] load x         [8] vreturn
}
```

Figure 23: Bytecode for method C.m4 from Section 1.1

arriving at an instruction pair (binop *plus*, binop *plus*) (i.e. at label pairs $(8,8)$, $(8,15)$, $(15,8)$, and $(15,15)$). The object allocations result in applications of the unary rule UniInstr (case New), using the fresh colour μ when applied to the left phrase and the fresh colour ν when applied to the right phrase. The invocations of method m1 are discharged using the unary rule UniInvV. The hypothesis of this proof rule that concerns the body of m1 is discharged by an invocation of the unary soundness result for m1 (verified in Section 3.3 above), guarded by an application of rule UniBSFrame in order to remove additional colours. All other instructions are verified using the unary syntax-directed rules, and the leaves of all proof branches are terminated by applications of rule RetRet.

The program also satisfies the alternative specification

$$\emptyset \vdash (\mathtt{C},\mathtt{m2},\mathbb{0}) \sim (\mathtt{C},\mathtt{m2},\mathbb{0}) : \phi \to \psi$$

where ϕ is as above and

$$\begin{array}{rcl}\psi & = & (([\beta],[\,],[\mu \mapsto (\mathtt{C},[\mathtt{A} \mapsto \beta])]), M, ([\beta],[\,],[\mu \mapsto (\mathtt{C},[\mathtt{A} \mapsto \beta])]))\\ M & = & N * [\alpha \mapsto \mathsf{int}] * [\beta \mapsto \mathsf{int}] * [\mu \mapsto \mathsf{class}(\mathtt{C})].\end{array}$$

Here, the colour μ occurs in both terminal abstract heaps, eliminating colour ν. The proof correlates the allocation events using rule NewNew instead of applying the unary rule UniInstr (case New), but otherwise proceeds as before.

Next, we prove that m2 is equivalent to the simplified method m4, the bytecode of which is shown in Figure 23. Again, two proofs are possible, differing on the correlation of allocations events. The judgement for the correlated case is

$$\emptyset \vdash (\mathtt{C},\mathtt{m2},\mathbb{0}) \sim (\mathtt{C},\mathtt{m4},\mathbb{0}) : \phi \to (\Sigma, M, \Sigma)$$

where

$$\begin{array}{rcl}\phi & = & (([\,],[\mathtt{l} \mapsto \gamma, \mathtt{h} \mapsto \delta],[\,]), N, ([\,],[\mathtt{l} \mapsto \gamma],[\,]))\\ N & = & [\gamma \mapsto \mathsf{int}] * [\delta \mapsto \mathsf{int}]\\ \Sigma & = & ([\beta],[\,],[\mu \mapsto (\mathtt{C},[\mathtt{A} \mapsto \beta])])\\ M & = & N * [\alpha \mapsto \mathsf{int}] * [\beta \mapsto \mathsf{int}] * [\mu \mapsto \mathsf{class}(\mathtt{C})]\end{array}$$

As the body of m4 consists of a single basic block, the proof tree consists only of two branches, resulting from an application of the unary rule for conditionals to instruction 3 of the left phrase. The pairs $(4,1)$, $(14,1)$, $(8,2)$, $(15,2)$, $(9,4)$, and $(17,4)$ are correlated using ConstConst, BinBin and NewNew.

Java:
```
class D {
  int m (int l, int h){
    x:=l;
    if  h
    then {   z:=new C;
             z.F:=l;
             return z.F }
    else  return x
}
}
```

Bytecode:
```
class D {
  int m (int l, int h){
     [0] load l       [5] vreturn      [10] putf C.F
     [1] store x      [6] new C        [11] load z
     [2] load h       [7] store z      [12] getf C.F
     [3] ifeq 6       [8] load z       [13] vreturn
     [4] load x       [9] load l
 }
}
```

Figure 24: Tracking the flow of values through objects

The uncorrelated case is proven by deriving

$$\emptyset \vdash (\mathtt{C}, \mathtt{m2}, \mathbb{0}) \sim (\mathtt{C}, \mathtt{m4}, \mathbb{0}) : \phi \longrightarrow (\Sigma, M, \Pi)$$

where ϕ and Σ are as above and

$$\begin{array}{rcl} \Pi & = & ([\beta], [\,], [\nu \mapsto (\mathtt{C}, [\mathtt{A} \mapsto \beta])]) \\ M & = & N * [\alpha \mapsto \mathsf{int}] * [\beta \mapsto \mathsf{int}] * [\mu \mapsto \mathsf{class}(\mathtt{C})] * [\nu \mapsto \mathsf{class}(\mathtt{C})] \end{array}$$

The proof uses the unary rules for the allocations but (in order to guarantee the equality of the return values) nevertheless correlates the instruction pairs for the instruction forms $\mathsf{const}\ 3$ and $\mathsf{binop}\ \mathit{plus}$.

Finally, we consider a program with branches that differ in their allocation behaviour. The verification of method D.m in Figure 24 garbage-collects the colour allocated in instruction 6 using the rules GarbageHP and GarbageN in the terminal abstract state of the positive branch. The overall judgement $\emptyset \vdash (\mathtt{D}, \mathtt{m}, \mathbb{0}) \sim (\mathtt{D}, \mathtt{m}, \mathbb{0}) : \phi \longrightarrow (\Sigma, N, \Sigma)$ where

$$\begin{array}{rcl} \phi & = & (([\,], [\mathtt{l} \mapsto \alpha, \mathtt{h} \mapsto \beta], [\,]), N, ([\,], [\mathtt{l} \mapsto \alpha, \mathtt{h} \mapsto \gamma], [\,])) \\ \Sigma & = & ([\alpha], [\,], [\,]) \\ N & = & [\alpha \mapsto \mathsf{int}] * [\beta \mapsto \mathsf{int}] * [\gamma \mapsto \mathsf{int}] \end{array}$$

only contains colours that specify values present in the initial and final states. The judgement represents the greatest common abstraction of the two branches in the sense that only objects are included in the final state on whose existence the program continuation may rely. The proof consists of four branches, and each branch is verified using the rule UniUni, i.e. by treating the two phrases independently. In all branches, the phrases with initial label 4 are verified syntax-directly, using the sequence UniRun, UniInstr (case Load), UniVret. The verification of the phrases for label 6 commences with the rule sequence GarbageN, GarbageHP, UniInstr (case New) and then proceed syntax-directly. The colour that is introduced in the third proof step (i.e. the application of rule UniInstr (case New)) is precisely the one to which the garbage-collection rules are applied in the first two proof steps.

The details of all proofs are available in [11].

4.4. Interpretation and soundness

Proceeding in a similar way as in Section 3.4 we first define when a pair of labels conforms to a relational specification

Definition 9. *Label pair* (ℓ, ℓ') conforms to specification ϕ, ψ, *notation*

$$\models \phi \xrightarrow{(\ell,\ell')} \psi,$$

if for all s, h, v, t, k, w, I, and J with

- $(s, t) \models_{I,J} \phi$,
- $\mathcal{P} \vdash s, \ell \Downarrow h, v$, *and* $\mathcal{P} \vdash t, \ell' \Downarrow k, w$,

there are I_1 *and* J_1 *such that* $(([v], [\,], h), ([w], [\,], k)) \models_{I_1,J_1} \psi$, *and*

a) *for all* $\gamma \in ODom(N)$, $I_1\gamma = I\gamma$ *and* $J_1\gamma = J\gamma$, *and*
b) *for all* $\gamma \in dom\ N$, $[\![\gamma]\!]^{([v],[\,],h)}_{\Omega,I_1} \subseteq [\![\gamma]\!]^{s}_{\Sigma,I}$ *and* $[\![\gamma]\!]^{([w],[\,],k)}_{\Xi,J_1} \subseteq [\![\gamma]\!]^{t}_{\Pi,J}$,

where $\phi = (\Sigma, N, \Pi)$ *and* $\psi = (\Omega, M, \Xi)$.

Definition 9 corresponds to the property *Safe* from the introduction.

For $\psi = ((S', \Gamma', H'), N', (T', \Delta', K'))$ and $\phi = ((S, \Gamma, H), N, (T, \Delta, K))$, we say that ψ *preserves* ϕ, notation $\phi \leq \psi$, if $dom\ H \subseteq dom\ H'$, $dom\ K \subseteq dom\ K'$, and N' contains N.

As a special case, consider $\phi \leq \psi$ where $\phi = ((S, \Gamma, H), N, (T, \Delta, K))$ and $\psi = ((\gamma :: S', \Gamma', H'), N', (\delta :: T', \Delta', K'))$ satisfy the object indistinguishability conditions $H \sim^{\mathcal{G}}_{N} K$ and $H' \sim^{\mathcal{G}}_{N'} K'$ from Section 4.1, for some fixed set $\mathcal{G}$ of public fields. Then, $\models \phi \xrightarrow{(\ell,\ell')} \psi$ guarantees that the two terminal heaps agree on the content of fields $\mathcal{G}$ whenever the initial heaps do. Provided that $\gamma = \delta$ holds, this condition amounts to the non-interference part of [6]'s notion of method safety (ignoring arrays and exceptions).

A second condition of [6]'s concept of method safety, *heap effect safety*, governs which fields may be updated. The condition concerns both executions individually, and is again formulated using the static visibility annotation of fields. In our setting, this condition may again be expressed as object indistinguishability, namely by requiring that $H \sim^{\mathcal{G}}_{N} H'$ and $K \sim^{\mathcal{G}}_{N} K'$ hold. Note that these ("horizontal") conditions relate the initial and final heaps of both abstract executions individually (and are both formulated with respect to the initial administrative map N), in contrast to the "vertical" conditions $H \sim^{\mathcal{G}}_{N} K$ and $H' \sim^{\mathcal{G}}_{N'} K'$.

The following definition thus captures non-interference in the sense of [6] for our language, with respect to statically fixed sets $\mathcal{F}_{low}$ and $\mathcal{X}_{low}$ of public fields and variables.

Definition 10. *The phrase with initial label* ℓ *is* non-interferent *w.r.t.* $\models \phi \xrightarrow{(\ell,\ell')} \psi$ *where* $\phi = ((S, \Gamma, H), N, (T, \Delta, K))$ *and* $\psi = ((\gamma :: S', \Gamma', H'), N', (\delta :: T', \Delta', K'))$ *if the following conditions are satisfied*

- *(low equivalence of initial stores): for all $x \in \mathcal{X}_{low}$, there is some γ with $\Gamma\downarrow x = \gamma = \Delta\downarrow x$.*
- *(preservation of object indistinguishability): $H \sim_N^{\mathcal{F}_{low}} K$ and $H' \sim_{N'}^{\mathcal{F}_{low}} K'$*
- *(heap effect safety): $H \sim_N^{\mathcal{F}_{low}} H'$ and $K \sim_N^{\mathcal{F}_{low}} K'$,*
- *(low-equality of return values): $\gamma = \delta$*

The proof system is more liberal than the one in [6] in that it does not enforce any of the properties involving $\mathcal{F}_{low}$ or $\mathcal{X}_{low}$ at intermediate program points.

The criterion with respect to which we prove the soundness of the proof system corresponds to the predicate *Interprete* from the introduction. It requires $\models \phi \xrightarrow{(\ell,\ell')} \psi$ to hold for all separated extensions of (ϕ, ψ), similar to the development for the unary system. Specialising this property to non-interference thus yields a stronger property than [6].

Definition 11. *Label pair* (ℓ, ℓ') is sound for ϕ and ψ*, notation $\models \ell \sim \ell' : \phi \Rightarrow \psi$, if $\phi \leq \psi$ is satisfied and*

$$\models \phi * (H, N, K) \xrightarrow{(\ell,\ell')} \psi * (H, N, K)$$

holds for all (H, N, K) that make the separated RSD-extension $\psi * (H, N, K)$ (and by $\phi \leq \psi$, also $\phi * (H, N, K)$) well-defined.

Again, the interpretation requires colours and allocated abstract addresses to be preserved. This preservation may be compared to the condition imposed in [23, 6] mandating that the partial bijection relating the terminal heaps should contain the partial bijection relating the initial heaps. However, our condition also preserves colours and objects that occur in only one of the two initial states. We remark that even in the absence of objects (and hence of the partial bijections), an invariant linking initial to final states is usually included in the interpretation of previous type systems for interference. Indeed, without this stronger invariant (which in the case of type systems in the style of Volpano et al. concerns the interpretation of judgements with *pc*-type *high*), the security property one is really interested in (non-interference, whose formulation indeed does *not* relate initial values to final ones) could not be established. Our condition requiring the preservation of colours (and their interpretation) may thus be seen as a generalisation of the strengthened interpretation of previous type judgements.

The soundness result of the relational proof system is shown in a similar way as Theorem 1, and in fact employs the result of Theorem 1 in the cases where the relational proof rules have unary hypotheses.

Theorem 2. *Let $\vdash G$ and $G \vdash \ell \sim \ell' : \phi \to \psi$. Then $\models \ell \sim \ell' : \phi \Rightarrow \psi$.*

Again, the proof of this theorem follows the technique developed by Kleymann and Nipkow [26, 27]. The absence of a small-step judgement makes the auxiliary induction

```
NIL  Copy(){  [0] new NIL      [1] vreturn}
CONS  Copy(){
   [0] load this       [5] store t          [10] store z          [15] load t
   [1] getf CONS.HD    [6] load t           [11] load z           [16] putf CONS.TL
   [2] store h         [7] invVirt LIST.Copy [12] load h          [17] load z
   [3] load this       [8] store t          [13] putf CONS.HD     [18] vreturn
   [4] getf CONS.TL    [9] new CONS         [14] load z
}
```

Figure 25: Bytecode representation of the code from Figure 2

refer only to a single judgement form, but the relativised notion of validity takes the derivation height indices of both phrases into account by bounding their sum.

In the soundness proofs of both derivation systems, the cases referring to the frame rules are straight-forward due to the inclusion of the frame extension in the interpretations. Of course, this extension complicates the proofs of some of the other rules, in particular the renaming rules and the rules for garbage-collecting unused colours. Here, the colours chosen by the renaming functions for extensions of the RSD's in the hypotheses are a priori not guaranteed to be distinct from the colours used in the extension of the concluding judgements. In order to deal with such clashes, the formalised proofs involve intermediate renamings.

5. Heap-allocated data structures

We now turn to the verification of data structures that are laid out in the heap, and to recursive methods operating on these structures. Using the code for copying lists given in Section 1.1 (Figure 2) as a running example, we introduce meta-level operators that define or specify components of RSD's in a similar way as datatype representation predicates constrain concrete states in Reynolds' exposition of separation logic [12]. These specifications are then used to verify unary and relational properties of the copying routine. Figure 25 shows possible bytecode resulting from translating the code from Figure 2.

5.1. Abstract representation predicates

Datatype representation predicates as employed in separation logic [12] are formulae of the object logic that capture the representation of high-level data structures as a collection of heap cells. Typically, the formulae specify collections of separated cells (making critical use of separated conjunction) that are related to each other by heap references and by logical invariants over values held inside the cells. The definition of the predicates follows the structure of the high-level data types, for example using structural induction.

In the absence of logical features that would enable the formulation of complex invariants involving numeric relationships between values, representation predicates suitable for our purpose only concern colours and their (non-)separation given by the interpretation of RSD's. Due to the structure of judgements in our calculi, the predicates we introduce do not yield arbitrary logical formulae but construct or characterise components of RSD's.

The first predicate **Lst** γ Λ constructs an abstract heap containing the spine of a list rooted at γ. The argument Λ contains a sequence of pairs (δ, ε), where δ specifies an abstract pointer to the HD-element of a cell and ε represents the abstract pointer to the successor cell. The predicate is defined by induction on Λ.

$$\begin{aligned} \mathbf{Lst}\,\gamma\,[\,] &\equiv [\gamma \mapsto (\mathsf{NIL}, [])] \\ \mathbf{Lst}\,\gamma\,(\delta,\varepsilon) :: \Lambda &\equiv \mathbf{Lst}\,\varepsilon\,\Lambda * [\gamma \mapsto (\mathsf{CONS}, [\mathsf{HD} \mapsto \delta, \mathsf{TL} \mapsto \varepsilon])] \end{aligned}$$

The implicit distinctness condition of operator $*$ enforces the colours representing nodes of the list's spine (i.e. γ and all the ε's) to be distinct. The pointers in the δ-position are unconstrained. In particular, we leave the values held inside these objects unspecified, as the verifications we consider below do not concern the functional faithfulness of representations of high-level data but only structural properties of their layout. In accordance with this, the definition of **Lst** γ Λ proceeds not by induction on high-level lists but by induction over lists of (pairs of) colours.

Abstract states applicable at the beginning of Copy's body are specified by

$$\mathbf{State}\,\gamma\,\Lambda \quad \equiv \quad ([], [\mathsf{this} \mapsto \gamma], \mathbf{Lst}\,\gamma\,\Lambda).$$

This predicate specifies a state with an empty operand stack, a heap containing the list specified by Λ and γ, and an abstract store that associates variable this with the initial cell of the list.

In a similar way as abstract heaps we specify administrative maps. For the verifications in Sections 5.2 and 5.3, the following construction suffices.

$$\begin{aligned} \mathbf{N}\,\gamma\,[] &\equiv [\gamma \mapsto \mathsf{class(NIL)}] \\ \mathbf{N}\,\gamma\,(\delta,\varepsilon) :: \Lambda &\equiv \mathbf{N}\,\varepsilon\,\Lambda * [\gamma \mapsto \mathsf{class(CONS)}] \end{aligned}$$

Each element in the list's spine results in one entry in the administrative map. The terminal cell is associated with class NIL, the other cells with class CONS. In Section 5.4 we will introduce a refinement of this construction that also specifies entries for the δ-positions. Regarding the verifications in Sections 5.2 and 5.3, even the type of the δ's may depend on the context program. The existence of dangling abstract pointers in judgements is permitted, as abstract states in judgements are not required to be closed (see the first paragraph of Section 3.2).

5.2. Unary verification

Aiming to give an input-output specification of Copy, we define specification quintuples **Spec** $\gamma\,\Lambda\,\delta\,\Upsilon = (\Sigma, N, H, \delta, M)$ where

$$\begin{aligned} \Sigma &= \mathbf{State}\,\gamma\,\Lambda & \qquad H &= \mathbf{Lst}\,\gamma\,\Lambda * \mathbf{Lst}\,\delta\,\Upsilon \\ N &= \mathbf{N}\,\gamma\,\Lambda & \qquad M &= \mathbf{N}\,\gamma\,\Lambda * \mathbf{N}\,\delta\,\Upsilon. \end{aligned}$$

(Here, Υ is again a list of colour pairs.) Specifications are collected in specification sets $\mathbf{Sp}\,n$ for $n \geq 0$ by

$$\mathbf{Sp}\,n = \{\mathbf{Spec}\ \gamma\ \Lambda\ \delta\ \Upsilon \mid |\Lambda| = n \wedge \mathit{map\,fst}\ \Lambda = \mathit{map\,fst}\ \Upsilon\}.$$

$\mathbf{Sp}\,n$ contains the specification quintuples for input and result lists with n CONS-nodes, such that the abstract HD-pointers in the two lists agree at each position. This property is enforced by the condition $\mathit{map\,fst}\ \Lambda = \mathit{map\,fst}\ \Upsilon$, which also guarantees $|\Lambda| = |\Upsilon|$. By the definition of $\mathbf{Spec}\ \gamma\ \Lambda\ \delta\ \Upsilon$ – in particular the use of $*$ in the definition of the components H and M – the spines are laid out distinctly, i.e. the colours in $\mathit{map\,snd}\ \Lambda$ and $\mathit{map\,snd}\ \Upsilon$ are guaranteed not to overlap.

We define the unary specification context $U_{\mathsf{Copy}} = U^0_{\mathsf{Copy}} \cup U^+_{\mathsf{Copy}}$ where

$$\begin{array}{rcl} U^0_{\mathsf{Copy}} & = & \{(\mathsf{NIL}, \mathsf{Copy}, 0), S) \mid S \in \mathbf{Sp}\,0\} \\ U^+_{\mathsf{Copy}} & = & \bigcup_{n>0} \{((\mathsf{CONS}, \mathsf{Copy}, 0), S) \mid S \in \mathbf{Sp}\,n\}. \end{array}$$

The context associates method NIL.Copy with the specifications for empty lists and the method CONS.Copy with the specifications for non-empty lists.

In order to show that these specifications are satisfied, we verify $\vdash U_{\mathsf{Copy}}$. The proof consists of two cases, one for an entry from U^0_{Copy} and one for an entry from U^+_{Copy}. The former entry concerns NIL.Copy and requires us to prove

$$U_{\mathsf{Copy}} \vdash (\mathsf{NIL}, \mathsf{Copy}, 0) : \Sigma_0, N_0 \Downarrow H_0, \delta, M_0,$$

where $\Sigma_0, \ldots, M_0$ arise from the definition of $\mathbf{Sp}\,0$ as

$$\begin{array}{rcl} \Sigma_0 & = & \mathbf{State}\ \gamma\ [\,] = ([], [\mathsf{this} \mapsto \gamma], \mathbf{Lst}\ \gamma\ [\,]) = ([], [\mathsf{this} \mapsto \gamma], [\gamma \mapsto (\mathsf{NIL}, [])]) \\ N_0 & = & \mathbf{N}\ \gamma\ [\,] = [\gamma \mapsto \mathsf{class(NIL)}] \\ H_0 & = & \mathbf{Lst}\ \gamma\ [\,] * \mathbf{Lst}\ \delta\ [\,] = [\gamma \mapsto (\mathsf{NIL}, [])] * [\delta \mapsto (\mathsf{NIL}, [])] \\ M_0 & = & \mathbf{N}\ \gamma\ [\,] * \mathbf{N}\ \delta\ [\,] = [\gamma \mapsto \mathsf{class(NIL)}] * [\delta \mapsto \mathsf{class(NIL)}]. \end{array}$$

The verification proceeds by applying the syntax-directed rule sequence UniRun, UniNew, UniVret as indicated in Figure 26, where we omit the class/method names NIL and Copy from all labels. The intermediate abstract state Π_0 is

$$\Pi_0 \quad = \quad ([\delta], [\mathsf{this} \mapsto \gamma], H_0).$$

The side condition $\delta \notin \mathit{dom}\ N_0$ follows from the inequality $\gamma \neq \delta$ that is implicit in the use of $*$ in the definition of H_0 and M_0.

The verification of an entry from U^+_{Copy} requires us to justify

$$U_{\mathsf{Copy}} \vdash (\mathsf{CONS}, \mathsf{Copy}, 0) : \Sigma_n, N_n \Downarrow H_n, \delta, M_n$$

where the components $\Sigma_n, \ldots, M_n$ arise from the definition of $\mathbf{Sp}\,n$, for some $n > 0$. We thus have some Λ_n and Υ_n of length n, with identical first projection and

$$\begin{array}{rclrcl} \Sigma_n & = & \mathbf{State}\ \gamma\ \Lambda_n = ([\,], [\mathsf{this} \mapsto \gamma], K_n) & H_n & = & K_n * \mathbf{Lst}\ \delta\ \Upsilon_n \\ N_n & = & \mathbf{N}\ \gamma\ \Lambda_n & M_n & = & N_n * \mathbf{N}\ \delta\ \Upsilon_n, \end{array}$$

$$\textsc{UniRun}\;\dfrac{\textsc{UniNew}\;\dfrac{\delta \notin dom\; N_0}{U_{\mathsf{Copy}} \vdash 0 : \Sigma_0, N_0 \Rightarrow \Pi_0, M_0, \{1\}} \qquad \textsc{UniRet}\;\dfrac{}{U_{\mathsf{Copy}} \vdash 1 : \Pi_0, M_0 \Downarrow H_0, \delta, M_0}}{U_{\mathsf{Copy}} \vdash 0 : \Sigma_0, N_0 \Downarrow H_0, \delta, M_0}$$

Figure 26: Verification of method body NIL.Copy.

$$\textsc{UniInvV}\;\dfrac{\ldots \qquad \textsc{UniBSFrame}\;\dfrac{\textsc{UniAx}\;\dfrac{((\mathsf{NIL},\mathsf{Copy},0),(\Sigma,\mathbf{N}\;\varepsilon\;\Lambda_0,H,\varepsilon',N)) \in U^0_{\mathsf{Copy}} \subseteq U_{\mathsf{Copy}}}{U_{\mathsf{Copy}} \vdash (\mathsf{NIL},\mathsf{Copy},0) : \Sigma, \mathbf{N}\;\varepsilon\;\Lambda_0 \Downarrow H,\varepsilon',N}}{U_{\mathsf{Copy}} \vdash (\mathsf{NIL},\mathsf{Copy},0) : ([\,],[\mathsf{this} \mapsto \varepsilon],K_1),N_1 \Downarrow K,\varepsilon',M}}{U_{\mathsf{Copy}} \vdash (\mathsf{CONS},\mathsf{Copy},7) : ([\varepsilon],\Gamma,K_1),N_1 \Rightarrow ([\varepsilon'],\Gamma,K),M,\{(\mathsf{CONS},\mathsf{Copy},8)\}}$$

$$\begin{array}{rcl@{\qquad}rcl}
\Gamma & = & [\mathsf{this} \mapsto \gamma, \mathsf{h} \mapsto \omega, \mathsf{t} \mapsto \varepsilon] & \Sigma & = & ([\,],[\mathsf{this} \mapsto \varepsilon], \mathbf{Lst}\;\varepsilon\;\Lambda_0) \\
H & = & \mathbf{Lst}\;\varepsilon\;\Lambda_0 * \mathbf{Lst}\;\varepsilon'\;\Upsilon_0 & N & = & \mathbf{N}\;\varepsilon\;\Lambda_0 * \mathbf{N}\;\varepsilon'\;\Upsilon_0 \\
K & = & K_1 * \mathbf{Lst}\;\varepsilon'\;\Upsilon_0 & M & = & N_1 * \mathbf{N}\;\varepsilon'\;\Upsilon_0
\end{array}$$

Figure 27: Verification of method body CONS.Copy for $k = 0$

where K_n abbreviates $\mathbf{Lst}\;\gamma\;\Lambda_n$. Writing $n = k + 1$, there are thus Λ_k and Υ_k of length $k \geq 0$, and $\omega, \ldots, \varepsilon'$, such that $\Lambda_n = (\omega, \varepsilon) :: \Lambda_k$, $\Upsilon_n = (\omega', \varepsilon') :: \Upsilon_k$ and

$$\begin{array}{rcl}
K_n & = & \mathbf{Lst}\;\varepsilon\;\Lambda_k * [\gamma \mapsto (\mathsf{CONS}, [\mathsf{HD} \mapsto \omega, \mathsf{TL} \mapsto \varepsilon])] \\
N_n & = & \mathbf{N}\;\varepsilon\;\Lambda_k * [\gamma \mapsto \mathsf{class}(\mathsf{CONS})] \\
H_n & = & K_n * \mathbf{Lst}\;\varepsilon'\;\Upsilon_k * [\delta \mapsto (\mathsf{CONS}, [\mathsf{HD} \mapsto \omega', \mathsf{TL} \mapsto \varepsilon'])] \\
M_n & = & N_n * \mathbf{N}\;\varepsilon'\;\Upsilon_k * [\delta \mapsto \mathsf{class}(\mathsf{CONS})],
\end{array}$$

where $\omega = \omega'$ follows from the identity of the first projections of Λ_n and Υ_n, and $\gamma \neq \delta$ and $\varepsilon \neq \varepsilon'$ follow from the use of $*$ in the definition of H_n and M_n.

Again, the proof proceeds by traversing the program in forward direction by interleaving UniRun with the appropriate syntax-directed rules. However, there is a case distinction due to the recursive call to LIST.Copy at instruction 7. In the case of $k = 0$ (i.e. the original list was of length one), the receiver object of the invocation is of class NIL. Consequently, the final hypothesis of rule UniInvV (i.e. the judgement concerning the invoked method) in this case concerns the body of NIL.Copy. We apply rule UniBSFrame in order to hide the invoking object $[\gamma \mapsto (\mathsf{CONS}, [\mathsf{HD} \mapsto \omega, \mathsf{TL} \mapsto \varepsilon])]$ and the corresponding entry $[\gamma \mapsto \mathsf{class}(\mathsf{CONS})]$ of the administrative map, before extracting the assumption on NIL.Copy from the proof context using rule UniAx. The small-step hypothesis of the application of UniRun at instruction 7 is thus discharged by the proof tree in Figure 27. The big-step hypothesis of said application of UniRun reads

$$U_{\mathsf{Copy}} \vdash (\mathsf{CONS}, \mathsf{Copy}, 8) : ([\varepsilon'], \Gamma, K), M \Downarrow H_1, \delta, M_1$$

and is again discharged in a syntax-directed fashion. The allocation instruction at label 9 introduces the fresh colour δ to M and also an abstract object to K, yielding M_1 and an abstract heap that is subsequently transformed to H_1 by instructions 10 to 16, in particular the field-modifications at labels 13 and 16.

```
NIL  Copy1(){[0]new NIL    [1]vreturn}
CONS  Copy1(){
   [0]load this         [5]invVirt LIST.Copy1    [10]putf CONS.HD
   [1]getf CONS.HD      [6]store t               [11]dup
   [2]store h           [7]new CONS              [12]load t
   [3]load this         [8]dup                   [13]putf CONS.TL
   [4]getf CONS.TL      [9]load h                [14]vreturn
}
```

Figure 28: List copy example (alternative version)

In the case of $k > 0$, the receiver object is of class CONS, hence the judgement extracted from $U_{\textsf{Copy}}$ by the axioms rule stems from $U^{+}_{\textsf{Copy}}$. Apart from this, the proof proceeds similar to the first case.

The need to perform this case distinction results from the design decision to enforce behavioural subtyping locally in the invocation rules. An alternative would be to introduce a global method specification table and to impose a behavioural subtyping condition on the entries of this table, as was done in our formalisation of a program logic for bytecode [30, 31]. The issue how to enforce behavioural subtyping is orthogonal to the subject of the present paper.

5.3. Relational verifications

The verification of a relational property follows a similar pattern as that of a unary property. Using the abstract representation predicates from Section 5.1, we first define specification tuples, and combine these to specifications that are parametrised by list lengths. We then introduce a (relational) proof context that associates such specifications to pairs of program labels. Finally, we prove that the resulting context is verified.

The first relational verification compares the original bytecode from Figure 25 to the code shown in Figure 28. In the latter code, the variables holding the tail pointer of the input list and the reference to the newly created CONS object have been eliminated, with the instructions operating on these variables either having been eliminated as well or having been replaced by stack operations.

Specification tuples for relational properties are given by RSD pairs. For the first verification we define tuples

$$\mathbf{RelSpec}_1\ \gamma\ \Lambda\ \delta\ \Upsilon = ((\Sigma, N, \Sigma), (\Pi, M, \Pi))$$

where $\Pi = ([\delta], [\,], H)$ and

$$\begin{array}{rclcrcl} \Sigma & = & \mathbf{State}\ \gamma\ \Lambda & \quad & N & = & \mathbf{N}\ \gamma\ \Lambda \\ H & = & \mathbf{Lst}\ \gamma\ \Lambda * \mathbf{Lst}\ \delta\ \Upsilon & & M & = & \mathbf{N}\ \gamma\ \Lambda * \mathbf{N}\ \delta\ \Upsilon. \end{array}$$

Both, the initial RSD (Σ, N, Σ) and the final RSD (Π, M, Π) are symmetric, i.e. contain identical left and right abstract states.

Specification tuples are combined to specifications $\mathbf{RelSp}_1\ n$ by

$$\mathbf{RelSp}_1\ n = \{\mathbf{RelSpec}_1\ \gamma\ \Lambda\ \delta\ \Upsilon \mid |\Lambda| = n \wedge \mathit{map\,fst}\ \Lambda = \mathit{map\,fst}\ \Upsilon\}.$$

Finally, we define the specification context $G_1 = G_1^0 \cup G_1^+$ where

$$\begin{array}{rcl} G_1^0 & = & \{(((\mathsf{NIL}, \mathsf{Copy}, 0), (\mathsf{NIL}, \mathsf{Copy1}, 0)), S) \mid S \in \mathbf{RelSp}_1\, 0\} \\ G_1^+ & = & \bigcup_{n>0} \{(((\mathsf{CONS}, \mathsf{Copy}, 0), (\mathsf{CONS}, \mathsf{Copy1}, 0)), S) \mid S \in \mathbf{RelSp}_1\, n\}. \end{array}$$

The proof of $\vdash G_1$ proceeds in a similar fashion as the proof of the unary specification, and is comprised of a part for $n = 0$ and a part for $n > 0$. The former part consists of the rule sequence NewNew, RetRet. It thus correlates the allocation events in the two phrases by employing a single fresh colour jointly in both components. The latter part applies unary proof rules independently in both phrases (injected by UniL and UniR) until the recursive method invocations are reached at the label pair $(7, 5)$. As in the unary case, we then perform a case distinction on $n = 1$ in order to separate invocations on receiver objects of class NIL from invocations on objects of class CONS. Both cases proceed by applying the rule for correlated method invocations, InvVInvV, whose final hypothesis is justified by appealing to the axiom rule, guarded by an application of the rule Frame. Having completed the step involving the method invocations, we proceed by applying the unary rule UniInstr (case Store) on both sides, and thus arrive at the allocation instructions in both phrases (label pair $(9, 7)$). Again, we correlate these instructions using NewNew. We then apply unary syntax-directed rules until reaching the end of both method bodies, and finally apply RetRet.

The fact that the proof correlates the allocations in both proof branches is related to the symmetry of the RSD (Π, M, Π) in the definition of $\mathbf{RelSpec}_1\ \gamma\ \Lambda\ \delta\ \Upsilon$, which forces both abstract executions to construct the copy using the colours in *map snd* Υ. Our second relational verification concerns an alternative specification for the same program pairs. However, the spines of the copies are now laid out using distinct colours. To this end, we define the specification entries

$$\mathbf{RelSpec}_2\ \gamma\ \Lambda\ \upsilon\ \Upsilon\ \upsilon'\ \Upsilon' = ((\Sigma, N, \Sigma), (\Pi, M, \Xi))$$

where

$$\begin{array}{rclcrcl} \Pi & = & ([\upsilon], [\,], H) & & \Xi & = & ([\upsilon'], [\,], K) \\ \Sigma & = & \mathbf{State}\ \gamma\ \Lambda & & N & = & \mathbf{N}\ \gamma\ \Lambda \\ H & = & \mathbf{Lst}\ \gamma\ \Lambda * \mathbf{Lst}\ \upsilon\ \Upsilon & & K & = & \mathbf{Lst}\ \gamma\ \Lambda * \mathbf{Lst}\ \upsilon'\ \Upsilon' \\ M & = & \mathbf{N}\ \gamma\ \Lambda * \mathbf{N}\ \upsilon\ \Upsilon * \mathbf{N}\ \upsilon'\ \Upsilon'. & & & & \end{array}$$

and combine these to specification sets

$$\mathbf{RelSp}_2\, n = \left\{ \mathbf{RelSpec}_2\ \gamma\ \Lambda\ \upsilon\ \Upsilon\ \upsilon'\ \Upsilon' \mid \left(\begin{array}{c} |\Lambda| = n\ \wedge \\ map\, fst\ \Lambda = map\, fst\ \Upsilon\ \wedge \\ map\, fst\ \Lambda = map\, fst\ \Upsilon' \end{array} \right) \right\}.$$

The nodes of the constructed list spines are distinct, due to the use of $*$ in the definition of M. In contrast, the references to the content nodes agree between the two copies (and coincide with the references used in the input list), thanks to the second and third conditions in the definition of $\mathbf{RelSp}_2\, n$. These conditions also ensure $|\Upsilon| = |\Upsilon'| = n$. The initial RSD's of specifications remain symmetric.

```
NIL  Copy2(){[0]new NIL    [1]vreturn}
CONS  Copy2(){
   [0]new CONS       [3]load this        [6]load this               [9]putf CONS.TL
   [1]dup            [4]getf CONS.HD     [7]getf CONS.TL            [10]vreturn
   [2]dup            [5]putf CONS.HD     [8]invVirt LIST.Copy2
}
```

Figure 29: List copy example (second alternative)

For context $G_2 = G_2^0 \cup G_2^+$ defined by

$$\begin{aligned} G_2^0 &= \{(((\mathsf{NIL}, \mathsf{Copy}, 0), (\mathsf{NIL}, \mathsf{Copy1}, 0)), S) \mid S \in \mathbf{RelSp}_2\, 0\} \\ G_2^+ &= \bigcup_{n>0} \{(((\mathsf{CONS}, \mathsf{Copy}, 0), (\mathsf{CONS.Copy1}, 0)), S) \mid S \in \mathbf{RelSp}_2\, n\} \end{aligned}$$

the proof of $\vdash G_2$ proceeds similar to the previous proof (in particular, the same case distinctions are made) but does not correlate the allocation instructions. Instead, each pair of allocations is verified by two applications of the one-sided rule UniInstr (case New). Regarding the recursive method invocations, two proofs are possible. The first proof correlates the invocations using rule InvVInvV. The final hypothesis of this rule is discharged by reference to the axiom rule for relational contexts, which is guarded by an application of Frame with respect to the invoking objects on both sides. Alternatively, we may apply the unary rule for virtual methods, UniInvV, once on each side, in which case we use the result proven in the previous section to discharge the final hypothesis of UniInvV in the left program component (using rules UniBSFrame and UniAx), and a similar result for method Copy1 to discharge the same hypothesis in the right program component.

Specifications using decoupled lists as in G_2 are also useful for verifying the equivalence of Copy to the variation shown in Figure 29. Here, the object allocation is moved to the front of the method, preceding the recursive method invocation. The appropriate context $G_3 = G_3^0 \cup G_3^+$ where

$$\begin{aligned} G_3^0 &= \{(((\mathsf{NIL}, \mathsf{Copy}, 0), (\mathsf{NIL}, \mathsf{Copy2}, 0)), S) \mid S \in \mathbf{RelSp}_2\, 0\} \\ G_3^+ &= \bigcup_{n>0} \{(((\mathsf{CONS}, \mathsf{Copy}, 0), (\mathsf{CONS}, \mathsf{Copy2}, 0)), S) \mid S \in \mathbf{RelSp}_2\, n\}. \end{aligned}$$

arises from G_2 by replacing Copy1 by Copy2. Using the rules given in Section 4, one may indeed show $\vdash G_3$. As in the previous proof we use the uncorrelated allocation rule UniInstr (case New), and may validate the invocations either using UniInvV twice (once on each side) or correlate them using InvVInvV. In contrast, an attempt to prove the equivalence between Copy and Copy2 for a context in the style of G_1 fails. As the order of the events *allocation* and *method invocation* do not agree between the two programs, neither rule NewNew nor rule InvVInvV may be invoked. In this sense, our calculus is less flexible than Necula's symbolic execution [20], as we require correlated events in the two executions to occur in the same relative order.

The proof attempts for the relation between Copy and Copy2 might suggest that relational contexts with correlated object colours are not very useful, as they appear less expressible and less flexible than contexts with uncorrelated object colours. Our final verification falsifies this suspicion.

5.4. Non-interference for alternating lists

The final verification concerns the non-interference of Copy, for lists which contain content elements of different visibility. The specification requires that the result list obeys the same visibility policy as the input list. We limit our attention to a policy where the visibility alternates between *high* and *low*. This regularity may easily be expressed using refined definitions of the abstract representation predicates. We expect that alternative patterns could be verified in a similar manner using appropriately modified abstract representation predicates.

Specifications comprise non-symmetric initial and final RSD's, such that each initial RSD models a list with content elements of alternating visibility and each final RSD models two such lists whose alternation patterns agree and coincide with that of the input list. The specification of the abstract heaps employs the previously defined predicate $\mathbf{Lst}\ \gamma\ \Lambda$. In order to model the (non-)distinctness of colours representing list cells and content elements, however, we refine the construction $\mathbf{N}\ \gamma\ \Lambda$ of administrative maps to the predicates $\mathbf{N_L}$ and $\mathbf{N_H}$. Both predicates are relations over nine-tuples

$$\mathcal{T} \times \mathfrak{C} \times \mathcal{L} \times \mathcal{L} \times (\mathfrak{C} \rightharpoonup_{fin} \mathcal{T}) \times \mathfrak{C} \times \mathcal{L} \times \mathcal{L} \times (\mathfrak{C} \rightharpoonup_{fin} \mathcal{T})$$

where $\mathcal{L}$ abbreviates $(\mathfrak{C} \times \mathfrak{C})$ *list*, and specify jointly the initial and final administrative maps of relational judgements. Intuitively, $(tp, \gamma, \Lambda, \Lambda', N, \delta, \Upsilon, \Upsilon', M) \in \mathbf{N_L}$ and $(tp, \gamma, \Lambda, \Lambda', N, \delta, \Upsilon, \Upsilon', M) \in \mathbf{N_H}$ both mean that N is an initial administrative map containing the colours used by the lists specified by γ, Λ, and Λ', whereas M is the corresponding terminal administrative map and contains additionally the colours from the lists specified by δ, Υ, and Υ'. In contrast to the maps constructed by $\mathbf{N}\ \gamma\ \Lambda$, maps N and M here contain not only entries for the lists' spines but also for the colours representing the content elements. The inclusion of the latter colours (which are associated with the type tp) enables us to model the visibility of content elements using (non)disjointness conditions.

Both predicates construct the initial and final administrative maps of a judgement w.r.t. a particular list length. The predicate $\mathbf{N_L}$ constructs the maps for *public* lists, by which we mean a list whose first content element is public, and whose tail is a private list. $\mathbf{N_H}$ constructs the maps for *private* list, by which we mean a list whose first content element is private, and whose tail is a public list. In both cases, the spines contain correlated colours and are of equal length. The (non)distinctness conditions expressed in the definition of the predicates enforce (non)-separation properties of the lists constructed by the predicate $\mathbf{Lst}\ \gamma\ \Lambda$. Figure 30 depicts a heaps of an RSD $\phi = (\Sigma, N, \Sigma')$ that models a public list of length five. Solid arrows depict references in Σ, dotted arrows references in Σ'. The predicates are defined in a mutually recursive fashion by the four rules shown in Figure 31 and are motivated as follows. Both, private and public lists may be empty (rules NH-Nil and NL-Nil), in which case the initial map N only contains a single element for the NIL cell, and the final map M contains an additional element for the freshly allocated copy. Similar to the definition

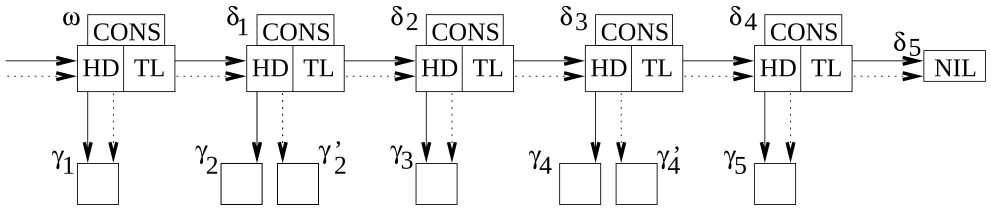

Figure 30: A public list of length five

$$\textsc{NH-Nil}\ \frac{N = [\omega \mapsto \mathsf{class(NIL)}] \qquad M = N * [\alpha \mapsto \mathsf{class(NIL)}]}{(tp, \omega, [], [], N, \alpha, [], [], M) \in \mathbf{N_H}}$$

$$\textsc{NH-Cons}\ \frac{\begin{array}{c}(tp, \delta, \Lambda, \Lambda', N, \epsilon, \Omega, \Omega', M) \in \mathbf{N_L} \\ N' = N * [\omega \mapsto \mathsf{class(CONS)}] * [\gamma \mapsto tp] * [\gamma' \mapsto tp] \\ M' = M * [\omega \mapsto \mathsf{class(CONS)}] * [\gamma \mapsto tp] \\ \quad * [\gamma' \mapsto tp] * [\alpha \mapsto \mathsf{class(CONS)}]\end{array}}{(tp, \omega, (\gamma, \delta) :: \Lambda, (\gamma', \delta) :: \Lambda', N', \alpha, (\gamma, \epsilon) :: \Omega, (\gamma', \epsilon) :: \Omega', M') \in \mathbf{N_H}}$$

$$\textsc{NL-Nil}\ \frac{N = [\omega \mapsto \mathsf{class(NIL)}] \qquad M = N * [\alpha \mapsto \mathsf{class(NIL)}]}{(tp, \omega, [], [], N, \alpha, [], [], M) \in \mathbf{N_L}}$$

$$\textsc{NL-Cons}\ \frac{\begin{array}{c}(tp, \delta, \Lambda, \Lambda', N, \epsilon, \Omega, \Omega', M) \in \mathbf{N_H} \\ N' = N * [\omega \mapsto \mathsf{class(CONS)}] * [\gamma \mapsto tp] \\ M' = M * [\omega \mapsto \mathsf{class(CONS)}] * [\gamma \mapsto tp] * [\alpha \mapsto \mathsf{class(CONS)}]\end{array}}{(tp, \omega, (\gamma, \delta) :: \Lambda, (\gamma, \delta) :: \Lambda', N', \alpha, (\gamma, \epsilon) :: \Omega, (\gamma, \epsilon) :: \Omega', M') \in \mathbf{N_L}}$$

Figure 31: Definition of predicates $\mathbf{N_L}$ and $\mathbf{N_H}$

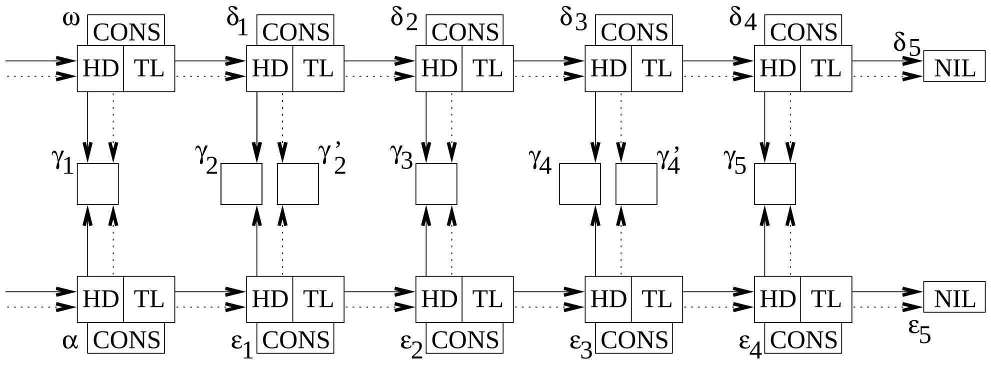

Figure 32: Copying the list from Figure 30

of earlier predicates, the use of $*$ enforces the distinctness of the colours α and ω. An administrative map N' for a non-empty *private* list (rule NH-Cons) contains the colours N of the *public* list with the joint head element δ and the tails Λ and Λ', where δ occurs in the head elements of both list descriptions of the rule's conclusion. The fact that δ is shared ensures that the spines of the two lists are correlated. The colour representing the head pointer of the newly constructed list, ω, is separatingly conjoined to N, ensuring that the lists are acyclic. The abstract pointers to the content elements, γ and γ', are also added to the administrative map, with their type specified by *tp*. Again, by the implicit distinctness condition of $*$, γ and γ' are distinct colours (i.e. the pointers to the content elements are not correlated), and are disjoint from ω and the colours in N. The latter condition means in particular that all content elements are coloured using distinct colours. The final map M' is constructed similarly by (separatingly) extending M by entries for ω, γ, and γ', and additionally by an entry for α, the colour representing the newly allocated object.

Administrative maps of non-empty *public* lists are constructed similarly (rule NL-Cons), modelling a *private* list specified by δ and the tails Λ and Λ', plus entries for the head cells ω and α, and the abstract content pointer γ. In contrast to rule NH-Cons, the content pointer is now required to be correlated: γ occurs in the first components of both abstract list descriptions. This models the fact that the head elements of public lists are indistinguishable.

We define non-interference specifications **NISpec** $tp\ \gamma\ \Lambda\ \Lambda'\ \upsilon\ \Upsilon\ \Upsilon'$ as the set of RSD pairs $((\Sigma, N, \Sigma'), (\Pi, M, \Pi'))$ such that

$$\begin{array}{ll} \Sigma = \textbf{State}\ \gamma\ \Lambda & \Sigma' = \textbf{State}\ \gamma\ \Lambda' \\ H = \textbf{Lst}\ \gamma\ \Lambda * \textbf{Lst}\ \upsilon\ \Upsilon & H' = \textbf{Lst}\ \gamma\ \Lambda' * \textbf{Lst}\ \upsilon\ \Upsilon' \end{array}$$

and $(tp, \gamma, \Lambda, \Lambda', N, \upsilon, \Upsilon, \Upsilon', M) \in \mathbf{N_H}$ or $(tp, \gamma, \Lambda, \Lambda', N, \upsilon, \Upsilon, \Upsilon', M) \in \mathbf{N_L}$, where $\Pi = ([\upsilon], [\,], H)$ and $\Pi' = ([\upsilon], [\,], H')$. Figure 32 shows the result of copying the list from Figure 30, i.e. an RSD that contains two copies of the list. The spine has been duplicated using colours $\alpha, \varepsilon_1, \ldots, \varepsilon_5$, but the links to the content elements are shared.

We define specification sets

$$\mathbf{NISp}_{tp}\ n = \bigcup_{|\Lambda|=n} \mathbf{NISpec}\ tp\ \gamma\ \Lambda\ \Lambda'\ \upsilon\ \Upsilon\ \Upsilon'$$

and the context $G_{tp} = G^0_{tp} \cup G^+_{tp}$ where

$$\begin{array}{rcl} G^0_{tp} & = & \{(((\mathsf{NIL}, \mathsf{Copy}, 0), (\mathsf{NIL}, \mathsf{Copy}, 0)), S) \mid S \in \mathbf{NISp}_{tp}\ 0\} \\ G^+_{tp} & = & \bigcup_{n>0} \{(((\mathsf{CONS}, \mathsf{Copy}, 0), (\mathsf{CONS}, \mathsf{Copy}, 0)), S) \mid S \in \mathbf{NISp}_{tp}\ n\}. \end{array}$$

The definition of the predicates $\mathbf{N_L}$ and $\mathbf{N_H}$ guarantees the required (non)-separation of colours in Λ, Λ', Υ and Υ'. Hence, no explicit conditions to this effect occur in the definition of $\mathbf{NISp}_{tp}\ n$.

The proof of $\vdash G_{tp}$ proceeds in a similar style as the relational verifications in the previous section. Both, the proof for $n = 0$ and the proof for $n > 0$ proceed largely syntax-directed and correlate the allocation instructions using rule NewNew. The recursive invocations of Copy in the proof for $n > 0$ are treated using the rule for correlated invocations, InvVInvV, guarded by an application of the frame rule, with a case distinction regarding $n = 1$.

The present section has demonstrated that RSD's together with inductively defined predicates yield an expressive specification formalism for heap-allocated data structures. Despite this flexibility, the verification proofs were only of moderate complexity, thanks to a combination of the frame rules and polyvariance.

6. Discussion

We discuss further related work, outline future work, and finally conclude.

6.1. Related work

In addition to the work already mentioned numerous analyses have been developed for non-interference and related notions. Sabelfeld and Myers' survey article [1] provides a comprehensive overview, concentrating mostly on high-level languages.

The use of the program dependence graph (PDG), respectively of def-use-chains, for the analysis of information flow has been advocated by Hammer et al. [19] and Bian et al. [32]. In particular, the combination of the PDG with a path analysis presented in [19] is reported to yield a more precise analysis than existing type systems. On the other hand, it is unclear how to formalise a semantic interpretation of the PDG in an efficient way, given the non-local nature of the PDG structure. Nevertheless, an extension of flow sensitivity towards path sensitivity would be an interesting avenue for future work. An analysis of non-interference of bytecode programs using a representation of program dependencies in terms of abstract transfer functions has been presented by [33]. Abadi et al.'s Core Calculus of Dependencies (CCD,[34]) formally unifies various notions of program dependencies, including Volpano et al.'s calculus and the SLam calculus [35], in an extension of Moggi's computational λ-calculus. As our approach does not explicitly track dependencies we do not see a direct way to relate it to CCD.

Self-composition [4] provides an alternative to type-based verification, avoiding relational calculi. Similar approaches have been advocated by Darvas et al. [3] and Joshi and Leino [2]. Terauchi and Aiken [5] demonstrate that integrating self-composition with program transformations leads to a verification approach that is more amenable to automated verification than the original system of [4]. A PCC implementation of self-composition would require trusted implementations of a verification condition generator and a verification condition checker, and evidence that the transformation converting a specific program into its self-composed form is applied correctly. In [36], the idea behind self-composition is recast in terms of a non-relational program logic, eliminating the need for the syntactic code transformation. It is then shown how the type system of Volpano et al., as well as that of Hunt and Sands [25] can be encoded as derived lemmas over restricted assertion formats. Hähnle et al. [37] present an encoding of the type system of Hunt and Sands in dynamic logics.

The use of relational techniques for studying information flow was pioneered by Sabelfeld in [38]. Naumann [39] proposed a combination of Benton's relational Hoare logic with information flow analysis by self-composition. In order to model self-composition for objects an encoding in ghost fields is proposed. Ghost fields are also employed in Warnier's specification of termination-sensitive non-interference in JML [40], and in work by Schubert and Chrzaszcz on the verification of a range of security properties using ESC/Java [41].

Pottier and Simonet [42] embed a higher-order functional language into a language of program pairs. Non-interference is obtained as a special case of a subject reduction theorem for a type system over this extended language.

Zanardini [43] outlined a variation of abstract non-interference for bytecode which shows some similarities to our approach, at the level of basic blocks. Little details are given about the treatment of objects and (non-)aliasing, and abstractions appear not to contain any formal conditions linking initial to final states.

An integrated treatment of security and transformations may also be of use for policies that fail to be semantics-respecting. Indeed transformations that are routinely (and silently) applied in compilers may destroy (or falsely establish) the satisfaction of such policies. In connection with this, and building upon earlier work by Agat [44] and Sabelfeld-Sands [45], Köpf and Mantel [46] present a framework for transforming programs that are sequentially secure, but have timing leaks, into observationally equivalent (hence again sequentially secure) programs without timing leaks. Treating a high-level imperative language extended by concurrency primitives, they apply a unification-based approach in which branches of high conditionals are padded by additional instructions. This framework may be seen as a technique also to eliminate sequentially insecure programs, as these cannot be transformed at all. The approach covers some cases of high branches, like **if** h_1 **then** $h_2{:=}e_1; l{:=}e_2$ **else** $l{:=}e_2$ where e_2 is a low expression. However, motivated by the concurrent setting, a stronger notion of security is employed which does not permit the reordering of low assignments to different variables. In contrast, our end-to-end interpretation admits such reorderings, and some corresponding transformations are indeed derivable from the rules given. Similar comments apply to the bisimulation-based approaches of [47, 45]. Again, these approaches concern the traces of events (where events include variable and field assignments) and retain the static distinction between public and private variables. On the other hand, the dichotomy of correlated and uncorrelated events indeed resembles weak bisimulation.

Formal proofs of code transformations on a per-program-basis are the subject of credible compilation [48] and translation validation [49, 20, 22, 50]. Rhodium [51] targets the formal justification of general optimisation algorithms. Using a domain-specific language, Rhodium separates the specification of transformation-enabling program analyses from the formulation and application of the transformations themselves. A type-based approach to the soundness of optimising bytecode transformations has been presented in [52]. The same authors recently presented an elegant proof of correctness of partial redundancy elimination [53].

6.2. Future work

We intend to extend our approach to a language fragment with exceptions and arrays. We expect that the transfer of control flow depending on the success of handling an exception should not be too difficult to deal with. Indeed, exceptions resemble conditionals in the setting of unstructured code. The inclusion of exceptions will then enable a more faithful treatment of object-related instructions that includes null-pointer cases. For this, the interpretation of abstract states will need to be adapted as object colours may not necessarily represent non-null locations.

As the relations between integer values abstracted to the same colour remain invariant throughout a judgement, large parts of our proof system are independent of the notion of indistinguishability. This motivates a generalisation to arbitrary relations over values, which might allow us to embed Banerjee et al.'s formulation of declassification policies [29] or Giacobazzi and Mastroeni's abstract non-interference [54]. Extending such a generalisation to object colours would enable one to express that the two references interpreting a colour in two states are either jointly null or jointly non-null.

An aspect of Amtoft et al.'s work that we have not discussed yet concerns the abstract locations that represent sets of concrete locations and are the basis of reasoning about separation in [14]. In our current work, abstract locations only represent single locations. Future work might generalise this discipline to groups of possibly aliasing locations, for example by using regions [55].

Following our intuitive introduction in Section 1, we point out that RSD's with empty abstract operand stacks are applicable to Java. A formulation of an RSD-based proof system at this language level would be a natural formalism for studying non-local program transformations. Subsequently, one might aim to develop a proof system for certified translation given by relational proof rules where the two program phrases stem from different languages.

Short of moving to a program logic altogether, more expressive variations of our technology might be obtained by equipping administrative maps either by a system of symbolic arithmetic expressions over colours or by a general relation over its domain.

Indeed, the formulation of relationships between (the values interpreting) colours appears necessary for the verification of complex program transformations such as partial redundancy elimination.

The present article considered a partial-correctness interpretation and termination-insensitive non-interference. A significant subset of the rules however appear also suitable for a termination-sensitive notion, or at least adaptable. The feature separating these two notions is co-termination, i.e. the property that two executions show equal termination behaviour. In order to satisfy this condition, all rules should ensure progress of their subject phrases. In the case of putfield and virtual method invocation, this requires the inclusion of the side condition $D \leq C$: without this, our operational semantics gets stuck. Furthermore, we expect that rule UniL would have to be equipped with a termination guarantee for the phrase at ℓ, in order to promote co-termination from (ℓ_1, ℓ') to (ℓ, ℓ'). In accordance with rule UniL, we envision a notion of co-termination that permits the two executions to take a differing number of basic instruction steps, and be formulated in big-step fashion. Finally, an initial exploration of these ideas suggests that co-termination is also beneficial for obtaining a horizontal composition rule

$$\frac{\emptyset \vdash \ell \sim \ell' : (\Sigma, N, \Sigma') \to (\Pi, M, \Pi') \quad \emptyset \vdash \ell' \sim \ell'' : (\Sigma', N, \Sigma'') \to (\Pi', M, \Pi'')}{\emptyset \vdash \ell \sim \ell'' : (\Sigma, N, \Sigma'') \to (\Pi, M, \Pi'')}$$

which composes transformations steps sequentially.

Soundness of the proof system being the focus of this article, we have not discussed proof inference or decidability. These are topics for future research, in particular in combination with translation validation and the algorithms included in some of the above-mentioned publications on that topic. Müller-Olm et al. [56] propose improved techniques for identifying polynomial identities between program variables, and Gulwani and Necula present an randomised analysis for affine equalities [57]. Both lines of work concern the values of program variables in a single program execution, whereas we require an analysis regarding the equivalence of values across two executions. One way to exploit their ideas may be to apply them to self-composed programs. Alternatively, an investigation of the relationship with static analyses techniques for copy propagation may be worth pursuing. Regarding the univariant case, dynamic programming techniques may be of use for deciding the syntax-directed fragment (i.e. the system without the renaming and frame rules) if one tabulates the specifications for all elements of the cross-product of program labels.

6.3. Conclusion

We presented technology for the certification of correlations of unstructured bytecode, based on relational state descriptions and unary and relational proof systems. Instead of tracking formal dependencies, our approach tracks the flow of correlated values through program executions. We avoid the calculation of control dependence regions and lift previous restrictions on the occurrence of assignments, allocations, field access and method invocations in branches. Our system incorporates instances of copy propagation, supports heap-local reasoning by frame rules, and admits the specification and verification of complex non-interference properties for heap-allocated data structures.

Acknowledgements

This work was supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and the DFG project InfoZert, grant number Be 3712/2-1. We are grateful to the members of both projects for discussions on type systems and program logics for information flow. We would also like to thank the reviewers for their insightful comments and suggestions, and the editors for organising the workshop and overseeing the publication process.

References

[1] A. Sabelfeld, A. C. Myers, Language-based information-flow security, IEEE Journal on Selected Areas in Communications 21 (1) (2003) 5 – 19, special issue on Formal Methods for Security.

[2] R. Joshi, K. R. M. Leino, A semantic approach to secure information flow, Science of Computer Programming 37 (2000) 113 – 138.

[3] Á. Darvas, R. Hähnle, D. Sands, A theorem proving approach to analysis of secure information flow, in: D. Hutter, M. Ullmann (Eds.), Proc. 2nd International Conference on Security in Pervasive Computing (SPC'05), Vol. 3450 of LNCS, Springer, 2005, pp. 193–209.

[4] G. Barthe, P. R. D'Argenio, T. Rezk, Secure information flow by self-composition, in: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04), IEEE Computer Society Press, 2004, pp. 100–114.

[5] T. Terauchi, A. Aiken, Secure information flow as a safety problem, in: C. Hankin, I. Siveroni (Eds.), Proceedings of the 12th International Symposium on Static Analysis (SAS '05), Vol. 3672 of LNCS, Springer, 2005, pp. 352–367.

[6] G. Barthe, D. Pichardie, T. Rezk, A Certified Lightweight Non-Interference Java Bytecode Verifier, in: R. D. Nicola (Ed.), Proceedings of 16th European Symposium on Programming (ESOP'07), Vol. 4421 of LNCS, Springer, 2007, pp. 125–140.

[7] R. Medel, A. B. Compagnoni, E. Bonelli, A typed assembly language for non-interference, in: M. Coppo, E. Lodi, G. M. Pinna (Eds.), Proceedings of the 9th Italian Conference on Theoretical Computer Science (ICTCS 2005), Vol. 3701 of LNCS, Springer, 2005, pp. 360–374.

[8] N. Kobayashi, K. Shirane, Type-based information analysis for low-level languages, in: Proceedings of the Third Asian Workshop on Programming Languages and Systems, (APLAS'02), 2002, pp. 302–316.

[9] G. C. Necula, Proof-carrying code, in: P. Lee, F. Henglein, N. D. Jones (Eds.), Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997), ACM Press, 1997, pp. 106–119.

[10] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002.

[11] L. Beringer, Relational bytecode correlations - Isabelle/HOL sources, available from the author's homepage, 2009.

[12] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002), IEEE Computer Society, 2002, pp. 55–74.

[13] N. Benton, Simple relational correctness proofs for static analyses and program transformations, in: Jones and Leroy [58], pp. 14–25.

[14] T. Amtoft, S. Bandhakavi, A. Banerjee, A logic for information flow in object-oriented programs, in: Morrisett and Peyton Jones [59], pp. 91–102.

[15] H. Yang, Relational separation logic, Theoretical Computer Science 375 (1-3) (2007) 308–334.

[16] L. Beringer, M. Hofmann, A. Momigliano, O. Shkaravska, Automatic certification of heap consumption, in: F. Baader, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Proceedings, Vol. 3452 of LNCS, Springer, 2004, pp. 347–362.

[17] A. W. Appel, Foundational proof-carrying code, in: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), IEEE Computer Society, 2001, pp. 247–258.

[18] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, Journal of Computer Security 4 (3) (1996) 167–187.

[19] C. Hammer, J. Krinke, G. Snelting, Information flow control for Java based on path conditions in dependence graphs, in: Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006), 2006, pp. 87–96.

[20] G. C. Necula, Translation validation for an optimizing compiler, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), Vol. 35(5) of SIGPLAN Notices, ACM, 2000, pp. 83–94.

[21] X. Leroy, Formal certification of a compiler back-end or: programming a compiler with a proof assistant, in: Morrisett and Peyton Jones [59], pp. 42–54.

[22] J.-B. Tristan, X. Leroy, Formal verification of translation validators: a case study on instruction scheduling optimizations, in: G. C. Necula, P. Wadler (Eds.), Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2008), ACM Press, 2008, pp. 17–27.

[23] A. Banerjee, D. A. Naumann, Stack-based access control and secure information flow, J. Funct. Program. 15 (2) (2005) 131–177.

[24] L. Birkedal, H. Yang, Relational parametricity and separation logic, in: H. Seidl (Ed.), Proceedings of the 10 International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2007), Vol. 4423 of LNCS, Springer, 2007, pp. 93–107.

[25] S. Hunt, D. Sands, On flow-sensitive security types, in: Morrisett and Peyton Jones [59], pp. 79–90.

[26] T. Kleymann, Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs, Ph.D. thesis, LFCS, University of Edinburgh, technical Report ECS-LFCS-98-392 (Sep. 1998).

[27] T. Nipkow, Hoare logics for recursive procedures and unbounded nondeterminism, in: J. C. Bradfield (Ed.), Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Proceedings, Vol. 2471 of LNCS, Springer, 2002, pp. 103–119.

[28] L. Beringer, M. Hofmann, A bytecode logic for JML and types, in: N. Kobayashi (Ed.), Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS'06), Vol. 4279 of LNCS, Springer, 2006, pp. 389 – 405.

[29] A. Banerjee, D. A. Naumann, S. Rosenberg, Towards a logical account of declassification, in: Proceedings of the 2007 ACM Workshop on Programming Languages and Analysis for Security (PLAS'07), ACM Press, 2007, pp. 61–65.

[30] Mobius-Consortium, Deliverable 3.1: Bytecode level specification language and program logic, available from `http://mobius.inria.fr` (2006).

[31] L. Beringer, M. Hofmann, M. Pavlova, Certification using the Mobius Base Logic, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W. P. de Roever (Eds.), Formal Methods for Components and Objects, 6th International Symposium, (FMCO 2007), Revised Lectures, Vol. 5382 of LNCS, Springer, 2008, pp. 25–51.

[32] G. Bian, K. Nakayama, Y. Kobayashi, M. Maekawa, Java bytecode dependence analysis for secure information flow, International Journal of Network Security 4 (1) (2007) 59–68.

[33] S. Genaim, F. Spoto, Information Flow Analysis for Java Bytecode, in: R. Cousot (Ed.), Proceedings of the Sixth International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2005), Vol. 3385 of LNCS, Springer, 2005, pp. 346–362.

[34] M. Abadi, A. Banerjee, N. Heintze, J. G. Riecke, A core calculus of dependency, in: A. W. Appel, A. Aiken (Eds.), Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999), ACM Press, 1999, pp. 147–160.

[35] N. Heintze, J. G. Riecke, The SLam calculus: programming with secrecy and integrity, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1998), ACM Press, 1998, pp. 365–377.

[36] L. Beringer, M. Hofmann, Secure information flow and program logics, in: Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07), IEEE, 2007, pp. 233–245.

[37] R. Hähnle, J. Pan, P. Rümmer, D. Walter, Integration of a security type system into a program logic, in: U. Montanari, D. Sannella, R. Bruni (Eds.), Trustworthy Global Computing, Second Symposium, TGC 2006, Revised Selected Papers, Vol. 4661 of LNCS, Springer, 2007, pp. 116–131.

[38] A. Sabelfeld, Semantic models for the security of sequential and concurrent programs, Ph.D. thesis, Chalmers University of Technology and University of Gothenburg (May 2001).

[39] D. Naumann, From coupling relations to mated invariants for checking information flow (extended abstract), in: D. Gollmann, J. Meier, A. Sabelfeld (Eds.), Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS 2006), Vol. 4189 of LNCS, Springer, 2006, pp. 279–296.

[40] M. Warnier, Language Based Security for Java and JML, Ph.D. thesis, Radboud University, Nijmegen, The Netherlands (2006).

[41] A. Schubert, J. Chrzaszcz, ESC/Java2 as a tool to ensure security in the source code of Java applications, in: Proceedings of Conference on Software Engineering Techniques - SET'2006, Vol. 227 of IFIP International Federation for Information Processing, Springer, 2006, pp. 337–348.

[42] F. Pottier, V. Simonet, Information flow inference for ML, in: J. Launchbury, J. C. Mitchell (Eds.), Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002), ACM Press, 2002, pp. 319–330.

[43] D. Zanardini, Abstract Non-Interference in a fragment of Java bytecode, in: Proceedings of the ACM Symposium on Applied Computing (SAC), ACM Press, 2006, pp. 1822–1826.

[44] J. Agat, Transforming out timing leaks, in: M. Wegman, T. Reps (Eds.), Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), ACM Press, 2000, pp. 40–53.

[45] A. Sabelfeld, D. Sands, Probabilistic noninterference for multi-threaded programs, in: Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00), IEEE, 2000, pp. 200–214.

[46] B. Köpf, H. Mantel, Eliminating implicit information leaks by transformational typing and unification, in: T. Dimitrakos, F. Martinelli, P. Y. A. Ryan, S. A. Schneider (Eds.), Formal Aspects in Security and Trust, Third International Workshop (FAST 2005), Revised Selected Papers, Vol. 3866 of LNCS, Springer, 2006, pp. 47–62.

[47] A. Bossi, C. Piazza, S. Rossi, Unwinding conditions for security in imperative languages, in: S. Etalle (Ed.), Revised Selected Papers of the 14th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2004), Vol. 3573 of LNCS, Springer, 2004, pp. 85–100.

[48] M. Rinard, D. Marinov, Credible compilation with pointers, in: Proceedings of the FLoC Workshop on Run-Time Result Verification, 1999.

[49] A. Pnueli, M. Siegel, E. Singerman, Translation validation, in: B. Steffen (Ed.), Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98), Vol. 1384 of LNCS, Springer, 1998, pp. 151–166.

[50] A. Zaks, A. Pnueli, Covac: Compiler validation by program analysis of the cross-product, in: J. Cuéllar, T. S. E. Maibaum, K. Sere (Eds.), Proceedings of the 15th International Symposium on Formal Methods (FM2008), Vol. 5014 of LNCS, Springer, 2008, pp. 35–51.

[51] S. Lerner, T. D. Millstein, E. Rice, C. Chambers, Automated soundness proofs for dataflow analyses and transformations via local rules, in: J. Palsberg, M. Abadi (Eds.), Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), ACM Press, 2005, pp. 364–377.

[52] A. Saabas, T. Uustalu, Type systems for optimizing stack-based code, in: M. Huisman, F. Spoto (Eds.), Proceedings of the 2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2007), Vol. 190(1) of ENTCS, Elsevier Science, 2007, pp. 103–119.

[53] A. Saabas, T. Uustalu, Proof optimization for partial redundancy elimination, in: R. Glück, O. de Moor (Eds.), Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2008), ACM, 2008, pp. 91–101.

[54] R. Giacobazzi, I. Mastroeni, Abstract non-interference: parameterizing non-interference by abstract interpretation, in: Jones and Leroy [58], pp. 186–197.

[55] J. M. Lucassen, D. K. Gifford, Polymorphic effect systems, in: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 1988), ACM Press, 1988, pp. 47–57.

[56] M. Müller-Olm, M. Petter, H. Seidl, Interprocedurally analyzing polynomial identities, in: Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS), Vol. 3884 of LNCS, Springer, 2006, pp. 50–67.

[57] S. Gulwani, G. C. Necula, Discovering affine equalities using random interpretation, in: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), ACM Press, 2003, pp. 74–84.

[58] N. D. Jones, X. Leroy (Eds.), Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004), ACM Press, 2004.

[59] J. G. Morrisett, S. L. Peyton Jones (Eds.), Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), ACM Press, 2006.