From Coupling Relations to Mated Invariants for Checking Information Flow
(Extended Abstract)

David A. Naumann[*]

Stevens Institute of Technology, Hoboken NJ 07030 USA

Abstract. This paper investigates a technique for using automated program verifiers to check conformance with information flow policy, in particular for programs acting on shared, dynamically allocated mutable heap objects. The technique encompasses rich policies with forms of declassification and supports modular, invariant-based verification of object-oriented programs. The technique is based on the known idea of self-composition, whereby noninterference for a command is reduced to an ordinary partial correctness property of the command sequentially composed with a renamed copy of itself. The first contribution is to extend this technique to encompass heap objects, which is difficult because textual renaming is inapplicable. The second contribution is a systematic means to validate transformations on self-composed programs. Certain transformations are needed for effective use of existing automated program verifiers and they exploit conservative flow inference, e.g., from security type inference. Experiments with the technique using ESC/Java2 and Spec# verifiers are reported.

1 Introduction

Consider an imperative command S acting on variables with declaration Γ. For example, Γ could be $x{:}\mathsf{int}, y{:}\mathsf{int}, z{:}\mathsf{bool}$ and S could be $z := (y > 0); x := x + 1; y := x$. A standard notion of confidentiality policy is to label variables with levels from a partially ordered set, e.g., $\{low, high\}$ with $low \leq high$. This is interpreted to mean that information is only allowed to flow from one variable to another, say x to y, if the level of x is at most the level of y. Such a policy is of interest only under the *mandatory access control assumption* that a principal at level λ can directly read only variables with confidentiality label at or below λ. (Our results apply as well to the dual, integrity.)

This paper investigates a technique for using ordinary program verifiers to check conformance with policy, in particular for programs acting on shared, mutable heap objects and policies that specify flows with finer granularity than individual program variables. The intended application is to programs in Java and similar languages but the technique pertains to any program using pointer structure. The technique is known as *self-composition* [25,7,30,15]; it reduces security to an ordinary partial correctness property of the program composed with itself. The first contribution is a novel extension of this technique to encompass the heap. The second contribution is a systematic means

[*] Supported in part by NSF grants CCR-0208984 and CCF-0429894.

to validate certain transformations on self-composed programs that are needed to make effective use of automated program verifiers to check security; this draws on work by Benton [9], Terauchi and Aiken [30]. The third contribution is to report on promising experiments with the ESC/Java2 [18] and Spec# [6] tools and to pose challenges for improvement of these tools.

To explain the main ideas we begin by considering the scenario above, with imperative commands acting on variables of primitive type. For the specific lattice $\{low, high\}$ we write $x \in vis$ to express that x is low. The formal notion that visible outputs reveal no information about secret inputs (high variables) is called *noninterference* [27] and is expressed in terms of two runs of the program:[1] If two initial states s and s' agree on variables in vis and t, t' are the final states from running the program on s and s' respectively, then t and t' also agree on variables in vis.

In program verification, a dashed identifier like s' is often used to refer to the final state corresponding to s; we do not use dashes that way, but rather to indicate a coupling relation.

Because the noninterference property involves two runs, it cannot simply be monitored at runtime or expressed directly as a pre/post specification. For static analysis, a popular approach is by means of a type system in which types include security labels [31,21,27,24,4,8]. The rules prevent, e.g., assignment to a low variable of an expression containing a high variable. Static analysis is useful to detect bugs and trojans; it does not prevent attacks that violate the abstractions embodied by the language semantics on which the analysis and definition of noninterference are based.

Self-composition. Besides type checking, another approach that has been explored is based on Hoare logic [14,10,11]. The condition "s and s' agree on visible variables" can be expressed by an assertion $x = x' \wedge y = y' \wedge \ldots$ with an equation $x = x'$ for each $x \in vis$, where x' is fresh variable. Such an assertion can be interpreted in a pair of states s, s', where x' is the value of x in s', or better still in a single state that assigns values to both x and x'. Now the property that S is noninterferent can be expressed using a renamed copy S' acting on the dashed variables. Add to the language a combinator $|$ so that $S|S'$ means parallel, independent execution of S and S'. Then S is noninterferent just if $S|S'$ takes initial states satisfying $x = x' \wedge y = y' \wedge \ldots$ to final states satisfying the same. For example, S at the beginning of the paper is noninterferent for $vis = \{x\}$ and for $vis = \{x, y\}$ but not for $vis = \{x, z\}$.

Two features of this formulation are interesting in terms of policy. Most importantly, the pre-post specification can be extended to allow partial releases at finer granularity than variables. For example, the equation $encrypt(k, secret) = encrypt(k', secret')$ in

[1] This paper focuses on termination-insensitive noninterference. Covert channels such as timing and power consumption are also ignored. The rationale is that such flows are harder to exploit as well as to prevent —our aim is a practical means to reduce the risk of trojans and bugs in production software. For further simplification in this paper, programs are deterministic, only initial and final states are observable, and the heap is unbounded. Pointer arithmetic is disallowed, just as it is in Java and its cousins (ignoring `hashcode`), since otherwise it is very hard to constrain information flow. On the other hand, the results make no assumption about the memory allocator, which may depend on the entire state; this entails some complications concerning but is a price worth paying for applicability to real systems.

a precondition would allow release of the encryption but not the secret plaintext. The policy with postcondition $z = z'$ and precondition $(y > 0) = (y' > 0)$ is satisfied by the S at the beginning of the paper. Preconditions can also condition secrecy of a variable on event history or permissions, or allow one but not both of two secrets to be released (e.g., [10,29,1,5]).

The second feature relevant to policy is that the formulation does not directly handle label lattices bigger than $\{low, high\}$. But it is well known that the noninterference property for a general lattice L can be reduced to a conjunction of properties, using just $\{low, high\}$ with *low* representing all the levels below a fixed level in L and *high* representing the rest. Henceforth we consider policy in the form of a set *vis* of low variable and field names.

This paper focuses on checking programs for conformance with given policy. Because only terminating computations are considered, and because S and S' act on disjoint variables, computations of $S|S'$ are the same as computations of the sequence $S;S'$ (and also $S';S$). We have arrived at the *self-composition* technique [7,30]: noninterference of S is reduced to an ordinary partial correctness property of $S;S'$ with respect to specifications over dashed and undashed copies of program variables.

Partial correctness is undecidable whereas the type-based approach is fast. But efficiency is gained at the cost of conservative abstraction; typical type systems are flow insensitive and may be very conservative in other ways, e.g., lack of arithmetic simplification. With a complete proof system, and at the cost of interactive proving, any noninterferent program can be proved so using self-composition. What is really interesting is the potential to use existing automated verification tools to check security of programs that are beyond the reach of a conventional type-based analysis. There are two significant obstacles to achieving this potential; overcoming them is the contribution of the paper.

Obstacles. To see the first obstacle, note first that there is reason to be optimistic about automation: pre-post specification of policy involves only simple equalities, not full functional correctness. But Terauchi and Aiken [30] point out that to verify a simple correctness judgement $\{x = x'\}S;S'\{y = y'\}$ requires —in some way or another depending on the verification method— to find an intermediate condition that holds at the semicolon. Suppose S involves a loop computing a value for y. The intermediate condition needs to describe the final state of y with sufficient accuracy that after S' it can be determined that $y = y'$. In the worst case this is nothing less than computing strongest postconditions. The weakest precondition for S' would do as well but is no easier to compute without a loop invariant being given. (And similarly for method calls.) This obstacle will reappear when we consider the second obstacle.

The second obstacle is due to dynamically allocated mutable objects. Note first that there is little practical motivation, or means, to assign labels to references themselves since upward flows can create useful low-high aliases and reference (pointer) literals are not available in most languages. Rather, field names and variables are labeled, as in Jif [21] and [4,8]. But noninterference needs to be based on a notion of indistinguishability taking into account that some but not all references are low visible. References to objects allocated in “high computations” (i.e., influenced by high branching conditions) must not flow to low variables and fields. References that are low-visible may differ between two runs, since the memory allocator may be forced to choose different addresses due to differing high allocations (unless we make unrealistic assumptions about the allocator as in some works). Suppose a state s takes the form $s = (h, r)$ where r is an assignment to variables as before and h is a partial function from references to objects (that map field names to values). Indistinguishability of (h, r) and (h', r') can be formalized in terms of a partial bijective relation on references, interpreted as a renaming between the visible references in $\mathsf{dom}\, h$ and those in $\mathsf{dom}\, h'$ (as in [4,8,22]).

The second obstacle is that self-composition requires a "renamed copy" of the heap —but objects are anonymous. To join (h, r) and (h', r') into a single state, r and r' are combined as a disjoint union $r \uplus r'$ as before. But how can h and h' be combined into a single heap? And how can S be transformed to an S' such that computations of $S; S'$ correspond to pairs of computations of S? —all in a way that can be expressed using assertions in a specification language like JML [19]. Our solution adds ghost fields to every object; these are used in assertions that express the partition of the heap into dashed and undashed parts as well as a partial bijection between them. Theorem 1 says that our solution is sound and complete (relative to the verification system). The theorem applies to relational properties in general, not just noninterference. The full significance of this pertains to data abstraction and is beyond the scope of this paper, but the importance of relations between arbitrary pairs S and S' should become apparent in the following paragraphs.

Theorem 1 says that S is noninterferent just if the corresponding "partial correctness judgement" (Hoare triple) is valid for $S; S'$ where S' is the dashed copy. The point is to use an off-the-shelf verifier to prove it. But our relations involve the ghost fields that encode a partial bijection; as usual with auxiliary variables, a proof will only be possible if the program is judiciously augmented with assignments to these variables. The assignments are only needed at points where visible objects are allocated: for $x := \mathbf{new}\ C$ in the original program S (where C is the name of the object's class), we need in S' to add assignments to fields of the object referenced by x' to link the two —after both have been allocated: $x := \mathbf{new}\ C; x' := \mathbf{new}\ C; Mate(x, x')$ where $Mate$ abbreviates the ghost assignments. But consider the following toy example where allocation occurs in a loop. The policy is that `secret` does not influence the result, which is an array of objects each with `val` field set to `x`.

```
Node[] m(int x, int secret) {
   Node[] m_result;  m_result= new Node[10];  int i= 0;
   while (i<10) { m_result[i]= new Node();  m_result[i].val= x; i++; }
   return m_result;         }
```

If two copies of the method body are sequentially composed, all the undashed objects have been allocated before any of the dashed ones are, so they cannot be paired up as required, at least not without additional reasoning about the postcondition of the first loop —the first obstacle reappears!

To overcome the first obstacle, Terauchi and Aiken [30] exploit that the sequence $S; S'$ has special structure. They give a transformation whereby S' is interleaved and partially merged with S so that equalities between undashed variables and their dashed counterparts are more easily tracked by an automated verifier. In particular, for a loop **while** E **do** S **od** with guard condition E known to be low, the two copies can be merged as **while** E **do** $S; S'$ **od** rather than **while** E **do** S **od**; **while** E' **do** S' **od**. (Example method `m` is shown self-composed in Fig. 3 and transformed in Fig. 2.) There are other optimizations, e.g., commuting independent commands and replacing $x := E; x' := E'$ by $x := E; x' := x$ in case E is an integer expression known to be low (and a mating version for reference types). The transformations depend on prior information and of course they must be proved sound. The prior information is itself a noninterference property (e.g., $E = E'$ modulo renaming of references, for the loop transformation), but it can be weaker than the ultimate policy to be checked. The idea is that a type based analysis is used first; if it fails to validate conformance with the desired policy, it may still determine that some expressions are low and this can be exploited to facilitate the self-composition technique.

Similar considerations apply to modular reasoning about method calls, for which thorough investigation is left to future work.

This paper formulates the transformations using *relational Hoare logic* as advocated by Benton [9]. The observation is that the contextual information on which many transformations depend (e.g., compiler optimizations) can be expressed as relational properties, typically partial equivalence relations, that are checked by various static analyses (including information flow typing). To adapt and extend Benton's logic from simple imperative programs to objects requires that renamings be incorporated and additional rules are needed. Type inference would be performed on the originaal program S, but the program to be transformed is the self-composed version $S; S'$, so an intricate embedding is needed to justify use of the transformed program to check security of S. This is Theorem 2, which is formulated semantically. Development of the requisite proof rules is left to future work.

Overview. Sect. 2 sketches the language for which our results are formalized, focusing on the model of state. Sect. 3 formalizes relational correctness judgements ("Hoare quadruples") and defines some important relations like indistinguishability. Noninterference for command S in context Γ is expressed as the relational correctness judgement $\Gamma | \Gamma \models S \sim S : \mathscr{R} \longrightarrow\!\!\triangleright \mathscr{S}$ where $\mathscr{R}$ and $\mathscr{S}$ are the precondition and postcondition expressing the security policy. (Notation adapted from [28,9].)

Sect. 4 gives the main definitions, which use ghost fields and local conditions to encode a pair of states as a single state, and thereby encode relations as predicates. Sect. 5 gives the first theorem: a program satisfies a relational correctness judgement just if the self-composed version satisfies the partial correctness judgement obtained by combining pairs of initial and final states. That is, $\Gamma | \Gamma \models S \sim S : \mathscr{R} \longrightarrow\!\!\triangleright \mathscr{S}$ is equivalent to validity of a partial correctness judgement $\Delta \models \{\mathscr{R}^1\}\ S; S'\ \{\mathscr{S}^1\}$ where context Δ is the combined state space, also written $\Gamma \uplus \Gamma'$, that declares both dashed and undashed copies of the variables. Here $\mathscr{R}^1$ and $\mathscr{S}^1$ are predicates on this state space that encode relations $\mathscr{R}$ and $\mathscr{S}$, and S' is the dashed copy of S.

Sect. 6 illustrates how the encoding of state pairs from Sect. 4 can be expressed as a formula in a specification language like JML [19] and it reports on encouraging experiments. Sect. 7 describes rules by which $S; S'$ can be transformed to a merged form S^* under the assumption of some weaker security property $\Gamma | \Gamma \models S \sim S : \mathscr{T} \longrightarrow\!\!\triangleright \mathscr{T}$ that would be obtained by type inference. The transformation itself is expressed in a form like $\Delta | \Delta \models S; S' \sim S^* : \mathscr{T} \longrightarrow\!\!\triangleright \mathscr{T}$ that says the two are equivalent under the assumption. The second theorem confirms that $\Delta \models \{\mathscr{R}^1\}\ S^*\ \{\mathscr{S}^1\}$ implies $\Delta \models \{\mathscr{R}^1\}\ S; S'\ \{\mathscr{S}^1\}$, thereby reducing the original security problem, $\Gamma|\Gamma \models S \sim S : \mathscr{R} \longrightarrow\!\!\triangleright \mathscr{S}$, to verification of $\Delta \models \{\mathscr{R}^1\}\ S^*\ \{\mathscr{S}^1\}$.

Sect. 8 discusses related work and issues in modular specification of the "mating invariant" in JML and similar specification languages. An online version gives omitted definitions and proofs.

2 Programs, Semantics, and Partial Correctness Judgements

Our results pertain to languages like Java where objects are instances of named classes and the class of a reference can be tested (and cast). No arithmetic operations are applicable to references, except for equality test. One key lemma (Lemma 1), is proved by induction on syntax and thus requires a semantics for commands. The full version of the paper uses a language similar to that in [3]: a complete program is a class table, i.e., closed set of class declarations in which fields and methods can be mutually recursive. Commands include field update, assignment, control structures and local blocks dynamically bound method calls —essentially sequential Java. The main ideas and results only involve the semantic entities, in particular states and state transformers.

Programs are assumed to be type-correct. A command in context, written $\Gamma \vdash S$, denotes a state transformer of type $\Gamma \rightsquigarrow \Gamma$, i.e., a total function from initial states for Γ to $\bot$-lifted states for Γ. The improper state $\bot$ represents divergence and error. This denotational style of semantics is used primarily in order to support modular reasoning about method invocations. (The full version of the paper addresses modular, per-method verification as it is done in tools like ESC/Java2.)

A single syntactic category, "variable names", is used for field, parameter, and local variable names, with typical element x. The data types are given by $T ::= \mathsf{int} \mid \mathsf{bool} \mid C$ where C ranges over names of declared classes. A value of a class type C is either null or a reference to an allocated object of type C. Subtyping is the reflexive, transitive relation $\leq$ determined by the immediate superclass given in each class declaration.

States have no dangling pointers and every object's field and every local variable holds a value compatible with its type. To formalize these conditions and others, it is convenient to separate the type of an object from the state of its fields. A *ref context* is a finite partial function ρ that maps references to class names. The idea is that if $o \in \mathsf{dom}\,\rho$ then o is allocated and moreover o points to an object of type $\rho\, o$. We write $[\![T]\!]\rho$ for the set of values of type T in a state where ρ is the ref context. In case T is a primitive type, $[\![T]\!]\rho$ is a set of values, independent from ρ. But if T is a class C then $[\![C]\!]\rho$ is the set containing nil and all the allocated references $o \in \mathsf{dom}\,\rho$ with $\rho\, o \leq C$.

Given context Γ and ref context ρ, a *store for Γ in ρ* is an assignment r of values to variables, such if $x : T$ is in Γ then $r x$ is in $[\![T]\!]\rho$. Let $[\![\mathsf{Sto}\Gamma]\!]\rho$ be the set of stores for Γ in ρ. For instance, we write $\mathsf{fields}C$ for the variable context of declared and inherited fields of objects of exactly type C. Thus a store in $[\![\mathsf{Sto}(\mathsf{fields}C)]\!]\rho$ represents the state of a C-object. A *heap* for ρ is a function that maps each reference $o \in \mathsf{dom}\,\rho$ to an object of class $\rho\, o$, i.e., to an element of $[\![\mathsf{Sto}(\mathsf{fields}(\rho\, o))]\!]\rho$. Thus for $h \in \mathsf{Heap}\,\rho$ and $o \in \mathsf{dom}\,\rho$, the application $h o x$ (sometimes written $h o.x$ for clarity) denotes the value of field x of object o in h. A *pre-heap* is like a heap but with dangling pointers allowed. A *program state* for given context Γ is a triple (ρ, h, r) containing a ref context, a heap, and a store for Γ. The set of states for Γ is written $[\![\Gamma]\!]$ —note the absence of parameter ρ, since the ref context is part of the state. Finally, the meaning $[\![\Gamma \vdash S]\!]$ of a command S is a (total) function from $[\![\Gamma]\!]$ to $[\![\Gamma]\!] \cup \{\bot\}$.

Partial correctness judgements. It is usual for postconditions to be two-state predicates, i.e., to have some means to refer to the initial state and thereby relate initial and final values, e.g., "old" expressions in JML or auxiliary variables. We formalize auxiliaries in terms of indexed families of predicates. Suppose that $\mathscr{P}$ and $\mathscr{Q}$ are indexed families of predicates $\mathscr{P}_\tau \subseteq [\![\Gamma]\!]$, $\mathscr{Q}_\tau \subseteq [\![\Gamma]\!]$ for some Γ and with τ ranging over some set. Then define $\Gamma \models \{\mathscr{P}\}\ S\ \{\mathscr{Q}\}$ to mean $\forall \tau\ .\ \Gamma \models \{\mathscr{P}_\tau\}\ S\ \{\mathscr{Q}_\tau\}$. Here $\Gamma \models \{\mathscr{P}_\tau\}\ S\ \{\mathscr{Q}_\tau\}$ means $\forall t \in [\![\Gamma]\!]\ .\ \mathscr{P}_\tau t \wedge [\![\Gamma \vdash S]\!] t \neq \bot \Rightarrow \mathscr{Q}_\tau([\![\Gamma \vdash S]\!] t)$. In fact our only use of auxiliaries is to manipulate pointer renamings encoded in the state.

3 Coupling Relations and Relational Correctness Judgements

In this section we specify noninterference in terms of relational correctness judgements, where couplings involve bijective renaming of visible objects. Throughout the paper, we let τ and σ range over finite bijective relations on the (infinite) set of references. For such a relation we write $\tau : \rho \leftrightarrow \rho'$, and say τ is a *typed bijection from* ρ *to* ρ', if and only if $\mathsf{dom}\,\tau \subseteq \mathsf{dom}\,\rho$, $\mathsf{rng}\,\tau \subseteq \mathsf{dom}\,\rho'$, and $\rho\, o = \rho'\, o'$ for all $(o,o') \in \tau$. Finally, τ is *total*, and called a *renaming*, if $\mathsf{dom}\,\tau = \mathsf{dom}\,\rho$ and $\mathsf{rng}\,\tau = \mathsf{dom}\,\rho'$. For states we write $\tau : s \leftrightarrow s'$ to abbreviate $\tau : \rho \leftrightarrow \rho'$ where $s = (\rho,h,r)$ and $s' = (\rho',h',r')$. Also $(o,o') \in \tau$ is often written as a curried application $\tau\, o\, o'$, that is, we confuse sets with characteristic functions.

For any Γ and Γ', an *indexed relation from* Γ *to* Γ' is a family, $\mathscr{R}$, indexed on typed bijections, such that $\mathscr{R}_\tau \subseteq [\![\Gamma]\!] \times [\![\Gamma']\!]$ for all τ and moreover if $\mathscr{R}_\tau(\rho,h,r)(\rho',h',r')$ then $\tau : \rho \leftrightarrow \rho'$. Note that we do not write $\mathscr{R}_\tau \subseteq [\![\Gamma]\!]\rho \times [\![\Gamma']\!]\rho'$ —we have not defined $[\![\Gamma]\!]\rho$. The first example is a kind of identity relation that takes into account that programs are insensitive to renaming, owing to the absence of address arithmetic. Indexed relations (on states) are usually defined in terms of a hierarchy of relations on simpler semantic objects —stores and values— and we reflect this in the notation.

$$
\begin{array}{lll}
\mathsf{Id}_\tau \Gamma\,(\rho,h,r)\,(\rho',h',r') & \iff & (\tau : \rho \leftrightarrow \rho',\ \text{is total}) \ \wedge\ \mathsf{Id}_\tau\,(\mathsf{Sto}\,\Gamma)\,r\,r' \\
 & & \wedge\ \forall (o,o') \in \tau\ .\ \mathsf{Id}_\tau\,(\mathsf{Sto}(\mathsf{fields}(\rho\, o)))\,(h\, o)\,(h'\, o') \\
\mathsf{Id}_\tau\,(\mathsf{Sto}\,\Gamma)\,r\,r' & \iff & \forall (x{:}T) \in \Gamma\ .\ \mathsf{Id}_\tau T\,(r x)\,(r' x) \\
\mathsf{Id}_\tau T\, v\, v' & \iff & v = v' \quad \text{for primitive type } T \\
\mathsf{Id}_\tau C\, v\, v' & \iff & (v = \mathsf{nil} = v') \vee \tau\, v\, v' \quad \text{for class } C
\end{array}
$$

Strictly speaking, it is the function mapping τ to $\mathsf{Id}_\tau \Gamma$ that is an indexed relation (in this case, from Γ to Γ); but we indulge in harmless rearrangement of parameters for clarity.

To understand the third conjunct in the definition of $\mathsf{Id}_\tau \Gamma$, recall that $\mathsf{fields}\,C$ is the typing context for the fields declared and inherited in class C, and $\rho\, o$ is the class of reference o. So this conjunct uses the instantiation $\Gamma := \mathsf{fields}(\rho\, o)$ of the relation $\mathsf{Id}_\tau\,(\mathsf{Sto}\,\Gamma)$ for stores. Note that $\mathsf{Id}_\tau T \subseteq [\![T]\!]\rho \times [\![T]\!]\rho'$ and $\mathsf{Id}_\tau\,(\mathsf{Sto}\,\Gamma) \subseteq [\![\mathsf{Sto}\,\Gamma]\!]\rho \times [\![\mathsf{Sto}\,\Gamma]\!]\rho'$.

Although $\mathsf{Id}_\tau \Gamma$ requires τ to be total on the relevant ref contexts, the definitions of $\mathsf{Id}_\tau(\mathsf{Sto}\,\Gamma)$ and $\mathsf{Id}_\tau T$ make no such restriction on τ. This is exploited in the definition of relation Ind and others below which require τ to be from ρ to ρ' but not total.

The next relation is the simple form of indistinguishability with respect to a set *vis* of low fields and variables, used, e.g., in [4].

$$\begin{aligned}
\mathsf{Ind}^{vis}_\tau \Gamma\,(\rho,h,r)\,(\rho',h',r') &\iff (\tau{:}\rho \leftrightarrow \rho') \wedge \mathsf{Ind}^{vis}_\tau(\mathsf{Sto}\,\Gamma)\,r\,r' \\
&\qquad \wedge\ \forall (o,o') \in \tau\,.\ \mathsf{Ind}^{vis}_\tau(\mathsf{Sto}(\mathsf{fields}(\rho\, o)))\,(h\,o)\,(h'\,o') \\
\mathsf{Ind}^{vis}_\tau(\mathsf{Sto}\,\Gamma)\,r\,r' &\iff \forall (x{:}T) \in \Gamma\,.\ x \in vis \Rightarrow \mathsf{Id}_\tau T\,(r\,x)\,(r'\,x)
\end{aligned}$$

Note that $\mathsf{Ind}^{vis}_\tau \Gamma$ differs from $\mathsf{Id}_\tau \Gamma$ in that Ind^{vis}_τ does not require τ to be total. References in $\mathsf{dom}\,\tau$ or in $\mathsf{rng}\,\tau$ are forced to include all those that are visible to the low observer; this is because the definition of $\mathsf{Ind}^{vis}_\tau(\mathsf{Sto}\,\Gamma)$ requires the Id_τ relation to hold for all visible variables (and fields), which in turn requires that references in these variables are related by τ.

Whereas Id and Ind are from Γ to itself, we need similar relations from Γ to the dashed copy Γ'. (Recall that fields do not get renamed, only locals.)

$$\begin{aligned}
\mathsf{Idd}_\tau \Gamma\,(\rho,h,r)\,(\rho',h',r') &\iff (\tau{:}\rho \leftrightarrow \rho' \text{ is total}) \wedge \mathsf{Idd}_\tau(\mathsf{Sto}\,\Gamma)\,r\,r' \\
&\qquad \wedge\ \forall (o,o') \in \tau\,.\ \mathsf{Id}_\tau(\mathsf{Sto}(\mathsf{fields}(\rho\, o)))\,(h\,o)\,(h'\,o') \\
\mathsf{Idd}_\tau(\mathsf{Sto}\,\Gamma)\,r\,r' &\iff \forall (x{:}T) \in \Gamma\,.\ \mathsf{Id}_\tau T\,(r\,x)\,(r'\,x')
\end{aligned}$$

Note that relation Id suffices for the heap objects and for values; the only real difference is that for stores we need to compare $r\,x$ with $r'\,x'$, i.e., to impose $x = x'$. Similarly:

$$\begin{aligned}
\mathsf{Indd}^{vis}_\tau \Gamma\,(\rho,h,r)\,(\rho',h',r') &\iff (\tau{:}\rho \leftrightarrow \rho') \wedge \mathsf{Indd}^{vis}_\tau(\mathsf{Sto}\,\Gamma)\,r\,r' \\
&\qquad \wedge\ \forall (o,o') \in \tau\,.\ \mathsf{Ind}^{vis}_\tau(\mathsf{Sto}(\mathsf{fields}(\rho\, o)))\,(h\,o)\,(h'\,o') \\
\mathsf{Indd}^{vis}_\tau(\mathsf{Sto}\,\Gamma)\,r\,r' &\iff \forall (x{:}T) \in \Gamma\,.\ x \in vis \Rightarrow \mathsf{Id}_\tau T\,(r\,x)\,(r'\,x')
\end{aligned}$$

For heap objects, the fields are not renamed, so $\mathsf{Ind}^{vis}_\tau(\mathsf{Sto}(\mathsf{fields}(\rho\, o)))$ is used here.

Suppose S and S' are commands over Γ and Γ' respectively, and $\mathscr{R}$, $\mathscr{S}$ are indexed relations from Γ to Γ'. Here Γ' can be any variable context, not necessarily the dashed copy of Γ; even $\Gamma' = \Gamma$ is allowed. We define the *relational correctness judgement* $\Gamma|\Gamma' \models S \sim S' : \mathscr{R} \longrightarrow\!\!\triangleright\, \mathscr{S}$ to mean $\forall \tau\,.\ \Gamma|\Gamma' \models [\![\Gamma \vdash S]\!] \sim [\![\Gamma' \vdash S']\!] : \mathscr{R}_\tau \longrightarrow\!\!\triangleright\, \mathscr{S}_\tau$. This in turn is defined in the following.

Definition 1 (relational correctness judgement). Suppose f is in $\Gamma \rightsquigarrow \Gamma$, f' is in $\Gamma' \rightsquigarrow \Gamma'$, both $\mathscr{R}$ and $\mathscr{S}$ are indexed relations from Γ to Γ', and τ is a typed bijection. Define $\Gamma|\Gamma' \models f \sim f' : \mathscr{R}_\tau \longrightarrow\!\!\triangleright\, \mathscr{S}_\tau$ iff $\forall s,s'\,.\ \mathscr{R}_\tau\, s\, s' \wedge f\,s \neq \bot \wedge f'\,s' \neq \bot \Rightarrow \mathscr{S}_\tau(f\,s)(f'\,s')$.

For the languages of [3,4], one can prove that programs are insensitive to renaming in the following sense: For all $\Gamma \vdash S$ we have $\Gamma|\Gamma \models S \sim S : \mathsf{Id}\,\Gamma \longrightarrow\!\!\triangleright\, \mathsf{Id}\,\Gamma$.

One might expect to express noninterference for S and policy *vis* as the judgement $\Gamma|\Gamma \models S \sim S : \mathsf{Ind}^{vis} \longrightarrow\!\!\triangleright\, \mathsf{Ind}^{vis}$ but this does not take into account that newly allocated objects can exist in the final state. In fact a sensible formulation of policy is $\Gamma|\Gamma \models S \sim S : (\exists \tau\,.\ \mathsf{Ind}^{vis}_\tau) \longrightarrow\!\!\triangleright\, (\exists \tau\,.\ \mathsf{Ind}^{vis}_\tau)$ which is attractive in that it eliminates the need for top level quantification over τ. But for modular checking of method calls, in particular

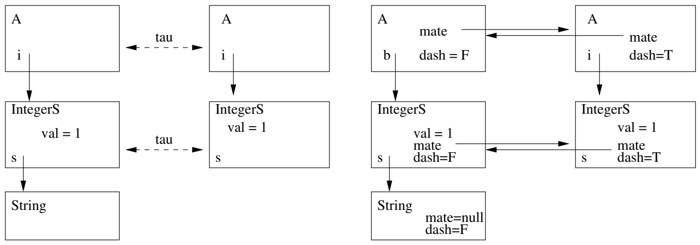

Fig. 1. On the left are two related heaps, encoded as one five-object heap on the right

to reason about the caller's store after a method call, it is important for the bijection supporting the final state to be an extension of the initial one. The property shown to be enforced by a type system in [4] is indeed this slightly stronger but more intricate property: $\Gamma|\Gamma \models S \sim S : \mathsf{Ind}^{vis}_{\tau} \rhd\!\!\!- \exists \sigma \supseteq \tau \,.\, \mathsf{Ind}^{vis}_{\sigma}$ for all τ. (Here we write a logical quantifier but the meaning is a union $\cup_{\sigma \supseteq \tau} \mathsf{Ind}^{vis}_{\sigma}$.) That is, our main use of relational correctness judgements will instantiate $\mathscr{S}_\tau$ in Def. 1 by $\mathscr{S}_\tau := \exists \sigma \,.\, \sigma \supseteq \tau \wedge \mathscr{R}_\sigma$. In the self-composed version to be used by a verifier, the bijection is encoded in auxiliary state and the condition $\sigma \supseteq \tau$ comes for free because the ghost fields need never be updated after initialization.

It is convenient to express noninterference in terms of the renamed program. Let Γ' be the dashed copy of Γ and S' be the dashed copy of S. Then clearly we have $\Gamma|\Gamma \models S \sim S : \mathsf{Ind}^{vis} \rhd\!\!\!- \mathsf{Ind}^{vis}$ if and only if $\Gamma|\Gamma' \models S \sim S' : \mathsf{Indd}^{vis} \rhd\!\!\!- \mathsf{Indd}^{vis}$.

4 Ghost Mating: Encoding Relations as State Predicates

This section defines the encoding of two states as one. Class Object is assumed to declare ghost fields dash : bool and mate : Object, so that they are present in all objects. None of the considered relations or programs should depend on these fields except through explicit use in the encoding.

Suppose ρ and ρ' are disjoint ref contexts, written $\rho \# \rho'$ (meaning $\mathsf{dom}\,\rho \# \mathsf{dom}\,\rho'$). Suppose we have typed bijection $\tau : \rho \leftrightarrow \rho'$, not necessarily total, and heaps $h \in \mathsf{Heap}\,\rho$, $h' \in \mathsf{Heap}\,\rho'$. We aim to encode a pair h, h' as a single heap k for $\rho \uplus \rho'$. The idea is that, in k, an object $o \in \mathsf{dom}\,\rho$ will have $k\,o.\mathsf{dash} = \mathsf{false}$ whereas an $o \in \mathsf{dom}\,\rho'$ will have $k\,o.\mathsf{dash} = \mathsf{true}$. Moreover, if $\tau\, o\, o'$ then we will have $k\,o.\mathsf{mate} = o'$ and $k\,o'.\mathsf{mate} = o$. This arrangement is formalized by conditions on k which can be expressed in formulas as $o.\mathsf{mate} \neq \mathsf{nil} \Rightarrow o.\mathsf{dash} = \neg o.\mathsf{mate}.\mathsf{dash} \wedge o.\mathsf{mate}.\mathsf{mate} = o$ and $o.x \neq \mathsf{nil} \Rightarrow o.\mathsf{dash} = o.x.\mathsf{dash}$ for every class type field $(x{:}C) \in \mathsf{fields}(\rho\, o)$, with $x \not\equiv \mathsf{mate}$. The right side of Figure 1 depicts a well mated heap. Given disjoint contexts Γ and Γ', a well mated state for $\Gamma \uplus \Gamma'$ is one where references in variables of Γ are to undashed objects (and Γ' to dashed). This notion is only used in the case that Γ' is the dashed copy of Γ.

Definition 2 (well mated state). Given disjoint contexts $\Gamma \# \Gamma'$, state $(\rho, h, r) \in [\![\Gamma \uplus \Gamma')]\!]$ is *well mated for* Γ *and* Γ' iff (a) h is well mated; (b) $r x = \mathsf{nil} \vee h(r x).\mathsf{dash} = \mathsf{false}$, for every x in $\mathsf{dom}\,\Gamma$ with Γx a class type; and (c) $r x' = \mathsf{nil} \vee h(r x').\mathsf{dash} = \mathsf{true}$, for every x' in $\mathsf{dom}\,\Gamma'$ with $\Gamma' x'$ a class type.

Note that there is no restriction on primitive values. We have $x \not\equiv \mathsf{mate}$ here because we assume field names are never reused as variable names.

Suppose that Γ is disjoint from Γ'. Given a typed bijection τ from ρ to ρ', we aim to combine states, say $(\rho,h,r) \in [\![\Gamma]\!]$ and $(\rho',h',r') \in [\![\Gamma']\!]$, into a single state that is well mated and reflects τ. We cannot assume $\rho\#\rho'$ —from initial states with disjoint heaps, running a pair of programs could lead to non-disjoint heaps (since we make no assumptions about the allocator). Instead, our construction includes a suitable renaming to make the heaps disjoint.

As a first step, function match is defined as follows. The idea is that for any $h \in \mathsf{Heap}\,\rho$, any $\tau : \rho \leftrightarrow \rho'$, and any boolean b, $\mathsf{match}(h,\tau,b)$ is a pre-heap where $o.\mathsf{dash} = b$ for every o and moreover if o is in the domain of τ then $o.\mathsf{mate}$ is a —dangling!— reference in accord with τ.

Now we define the combined state, $\mathsf{join}_\tau(\rho,h,r)(\rho',h',r')$, by the following steps. First, choose $\hat{\rho}$ and $\hat{\tau}$ such that $\hat{\rho}\#\rho$ and $\hat{\tau}$ is a renaming from ρ' to $\hat{\rho}$. Let $\hat{h}$ be the renaming of h' by $\hat{\tau}$ and *mutatis mutandis* for $\hat{r}$ and r'. Let $h_0 = \mathsf{match}(h,(\tau;\hat{\tau}),\mathsf{false})$ and also $\hat{h}_0 = \mathsf{match}(\hat{h},(\hat{\tau}^{-1};\tau^{-1}),\mathsf{true})$, writing ";" for relational composition. Finally, define $\mathsf{join}_\tau(\rho,h,r)(\rho',h',r') = ((\rho \uplus \hat{\rho}), h_0 \uplus \hat{h}_0, r \uplus \hat{r})$

Note that (ρ,h_0,r) is not quite an element of $[\![\Gamma]\!]$, because h_0 is only a pre-heap due to the dangling mate fields. For the same reason, $(\hat{\rho},\hat{h},\hat{r})$ is almost but not quite an element of $[\![\Gamma']\!]$. What matters is that if τ is a typed bijection from ρ to ρ' and $\Gamma\#\Gamma'$ then $\mathsf{join}_\tau(\rho,h,r)(\rho',h',r')$ is in $[\![\Gamma \uplus \Gamma']\!]$ and is well mated.

To partition a well mated heap into two, we first define $\mathsf{dsh}\ h = \{o \in \mathsf{dom}\,h \mid h\,o.\mathsf{dash} = \mathsf{true}\}$ and $\mathsf{undsh}\ h = \{o \in \mathsf{dom}\,h \mid h\,o.\mathsf{dash} = \mathsf{false}\}$. Roughly speaking, $h\restriction(\mathsf{dsh}\,h)$, i.e., h with its domain restricted to *include* only dashed objects, is in $\mathsf{Heap}(\rho\restriction(\mathsf{dsh}\,h))$. But in fact $h\restriction(\mathsf{dsh}\,h)$ may have dangling references in mate fields, so we define a function dematch that sets all mate fields to nil.

For splitting to invert joining we cannot just discard mates. If k in $\mathsf{Heap}\,\rho$ is well mated then we obtain typed bijection $\tau : (\rho\restriction(\mathsf{undsh}\,h)) \leftrightarrow (\rho\restriction(\mathsf{dsh}\,h))$ by

$$\tau\, o\, o' \iff k\,o.\mathsf{dash} = \mathsf{false} \wedge k\,o.\mathsf{mate} = o' \qquad (1)$$

Splitting and joining are mutually inverse, modulo renaming. The intricate details are omitted in this extended abstract. One consequence is that every well mated state in $[\![\Gamma \uplus \Gamma']\!]$ is equal, up to renaming, to one in the range of join. Even more, a well mated state that encodes via (1) a particular bijection τ is in the range of join_τ. Thus, for a relation that is insensitive to renaming, we can give a pointwise definition of a corresponding predicate.

Definition 3 (coupling relation to mated predicate). Given an indexed relation $\mathscr{R}$ from Γ to Γ' with $\Gamma\#\Gamma'$, define a predicate $\mathscr{R}^1_\tau \subseteq [\![\Gamma \uplus \Gamma']\!]$ by

$$\mathscr{R}^1_\tau t \iff \exists s,s' \,.\, t = \mathsf{join}_\tau s s' \wedge \mathscr{R}_\tau s s'$$

That is, t is in $\mathscr{R}^1_\tau$ iff t is well mated for τ and splits as some s,s' with $\mathscr{R}_\tau s s'$.

As an example, if a state with heap h satisfies $(Ind^{vis}_\tau)^1$ then for any o with $h\,o.\mathsf{mate} \neq \mathsf{nil}$ the visible primitive fields of $h\,o$ are equal to those of $h\,o.\mathsf{mate}$ and the visible class fields of $h\,o$ are mated. There is no constraints for field names not in *vis*.

5 Hoare Quadruples Reduced to Triples

This section shows that a relational correctness judgement for some coupling relation is equivalent to a partial correctness judgement for the corresponding predicate and the self-composed program. To begin, it is convenient to define $f \rhd f'$ which applies state transformer f and then f'. Suppose $\Gamma\#\Gamma'$, f is in $\Gamma \rightsquigarrow \Gamma$, and f' is in $\Gamma' \rightsquigarrow \Gamma'$. Define $f \rhd f'$ to be an element of $\Gamma \uplus \Gamma' \rightsquigarrow \Gamma \uplus \Gamma'$ as follows, where we partition the store as r, r' in accord with Γ, Γ'.

$$\begin{aligned}(f \rhd f')(\rho, h, r \uplus r') = \; & \mathsf{let}\ (\rho_0, h_0, r_0) = f(\rho, h, r)\ \mathsf{in} \\ & \mathsf{let}\ (\rho_1, h_1, r_1) = f'(\rho_0, h_0, r')\ \mathsf{in}\ (\rho_1, h_1, r_0 \uplus r_1)\end{aligned}$$

Our meta-notation "$\mathsf{let}\ -\ \mathsf{in}$" is $\bot$-strict, so $f \rhd f'$ returns $\bot$ if either f or f' does.

This notion is useful in case f acts only on the undashed part of the heap and f' on the dashed part, but the definition is more general. Note also that the domain $\Gamma \uplus \Gamma' \rightsquigarrow \Gamma \uplus \Gamma'$ includes state transformers that in no way respect the $\mathsf{dash}/\mathsf{mate}$ structure. In particular, well matedness of s does not imply the same for $(f \rhd f')s$. But we have the following.

Lemma 1. If $\Gamma \vdash S$, $\Gamma' \vdash S'$, and $\Gamma\#\Gamma'$ then $([\![\Gamma \vdash S]\!] \rhd [\![\Gamma' \vdash S']\!])s = [\![\Gamma \uplus \Gamma' \vdash S;S']\!]s$ for any well mated s. (Recall that we assume dash and mate do not occur in S or S'.)

A state transformer f is *independent from* $\mathit{mate}, \mathit{dash}$ provided it does not update these fields on initially existing objects or newly allocated objects and moreover $s \lfloor dm = t \lfloor dm \Rightarrow (f\,s) \lfloor dm = (f\,t) \lfloor dm$, where we abbreviate dm for $\mathsf{dash}, \mathsf{mate}$ and $\lfloor$ removes elements from a function's domain.

Lemma 2. Suppose $\Gamma\#\Gamma'$, f is in $\Gamma \rightsquigarrow \Gamma$, and f' is in $\Gamma' \rightsquigarrow \Gamma'$. (a) For any τ, u, s, s', if $u \lfloor dm = ((f \rhd f')(\mathsf{join}_\tau s s')) \lfloor dm$ and $u = \mathsf{join}_\sigma t t'$ for some σ, t, t' then $t = f\,s$ and $t' = f'\,s'$. (b) If f, f' are independent from $\mathsf{mate}, \mathsf{dash}$ then $((f \rhd f')(\mathsf{join}_\tau s s')) \lfloor dm$ is equivalent to $\mathsf{join}_\sigma\,(f\,s)(f'\,s')$ for some σ, up to renaming (i.e., related by Id).

To state the precise correspondence, in terms of states that include the dash and mate fields, we need to mask them as follows. Define $\hat{\mathscr{R}}^1_\tau$ by

$$\hat{\mathscr{R}}^1_\tau t \iff \exists u \,.\, u \lfloor dm = t \lfloor dm \wedge \mathscr{R}^1_\tau u$$

Theorem 1. Suppose $\Gamma\#\Gamma'$ and consider commands in context $\Gamma \vdash S$ and $\Gamma' \vdash S'$. Suppose $\mathscr{R}$ and $\mathscr{S}$ are indexed relations from Γ to Γ' that are insensitive to renaming. Then for any τ we have

$$\Gamma | \Gamma' \models S \sim S' : \mathscr{R}_\tau \longrightarrow\!\!\rhd \exists \sigma \supseteq \tau \,.\, \mathscr{S}_\sigma \quad \text{iff} \quad \Gamma \uplus \Gamma' \models \{\hat{\mathscr{R}}^1_\tau\}\ S;S'\ \{\exists \sigma \,.\, \sigma \supseteq \tau \wedge \hat{\mathscr{S}}^1_\sigma\}$$

In particular, noninterference for a command S and policy vis is, by definition, the property that $\Gamma | \Gamma \models \mathscr{R}_\tau \sim S : S \longrightarrow\!\!\rhd \exists \sigma \,.\, \sigma \supseteq \tau \wedge \hat{\mathscr{R}}_\sigma$ (for all τ) where $\mathscr{R}$ is $\mathsf{Ind}^{vis}_\tau \Gamma$. This is the same as the renamed version $\Gamma | \Gamma' \models \mathscr{R}_\tau \sim S : S' \longrightarrow\!\!\rhd \exists \sigma \,.\, \sigma \supseteq \tau \wedge \hat{\mathscr{R}}_\sigma$ where $\mathscr{R}$ is $\mathsf{Indd}^{vis}_\tau \Gamma$. The Theorem reduces this to the triple $\Gamma \uplus \Gamma' \models \{\mathscr{R}^1_\tau\}\ S;S'\ \{\exists \sigma \,.\, \sigma \supseteq \tau \wedge \hat{\mathscr{R}}^1_\sigma\}$.

6 Experiments: Expressing Well Mating and Relations as Assertions

Sections 4 and 5 formulate the technique in semantic terms. A key feature of our encoding is that it requires no special instrumentation of program semantics but rather is expressed by first order conditions over auxiliary state. One can think of a number of variations on encoding; we describe one convenient pattern.

To express the self composed program using JML, a fresh method with two copies of the parameters is used. For non-static methods, the target object (`this`) needs to be made an explicit parameter so there can be two copies. Two copies of the result are needed; our encoding uses fields for this purpose. As a simple example, consider this method where the policy is that `secret` does not influence the result.

```
static int p(int x, int y, int secret) {
   x= secret; if (secret % 2 == 1) y=x * secret; else y= secret * secret;
   return y - secret * x;               }
```

For the self composed version, two fields and a new method `Pair_p` are added to the class, as follows (using $ for dash which is not legal in Java identifiers).

```
int p_result, p_result$; // new fields to hold the pair of results of p

/*@ requires x==x$ && y==y$;
  @ modifies p_result, p_result$;
  @ ensures p_result == p_result$;
  @*/
void Pair_p(int x, int y, int secret, int x$, int y$, int secret$) {
   x= secret; if (secret % 2 == 1) y=x * secret; else y= secret * secret;
   p_result= y - secret * x;
   x$= secret$; if(secret$ % 2==1) y$=x$*secret$;else y$=secret$*secret$;
   p_result$= y$ - secret$ * x$;                                   }
```

This is verified by ESC/Java2 (version 2.0a9) in 0.057sec; insecure versions are quickly rejected. Similar results for this and the other experiments were found using the Spec# tool. Note that ESC/Java2 is deliberately unsound in some ways, though not in ways that appear relevant to the present experiments.

Another experiment is to adapt the preceding method `p` by using a wrapper object for the result. For experiment we add the ghost fields explicitly where needed.

```
class Node {
   public int val;
  /*@ ghost public Node mate; */
  /*@ ghost public boolean dash; */   }
```

The variation using such a wrapper object verifies without difficulty, since the requisite ghost updates can be added following the second allocation.

As discussed in Sect. 1 and justified in Sect. 7 to follow, loops are most easily checked by applying an interleaving transformation for allocations that occur under low guards. For method `m` in Sect. 1 the self-composed version appears in Figure 2, where the transformation from **while** E **do** S **od**;**while** E' **do** S' **od** to **while** E **do** $S;S'$ **od**

```
Node[] m_result, m_result$; // new fields to hold the pair of results of m

  /*@ requires x==x$;  // policy
    // ordinary preconditions
    @ ensures m_result != null && m_result$ != null;
    @ ensures m_result.length==10 && m_result$.length==10;
    @ ensures (\forall int j; 0<=j&&j<10 ==> m_result[j]!=null);
    @ ensures (\forall int j; 0<=j&&j<10 ==> m_result$[j]!=null);
    // mating and policy
    @ ensures (\forall int j; 0<=j&&j<10 ==> m_result[j].mate==m_result$[j]);
    @ ensures (\forall int j; 0<=j&&j<10 ==> m_result$[j].mate==m_result[j]);
    @ ensures (\forall int j; 0<=j&&j<10 ==> !m_result[j].dash);
    @ ensures (\forall int j; 0<=j&&j<10 ==> m_result$[j].dash);
    @ ensures (\forall int j; 0<=j&&j<10 ==> m_result[j].val==m_result$[j].val);
    @ assignable m_result, m_result$, m_result[*], m_result$[*];
  */
  void Pair_m(int x, int secret, int x$, int secret$) {
      m_result= new Node[10];  m_result$= new Node[10];
      // **mating assignments for the arrays would go here**
      int i= 0;
      //@ maintaining ...
      while (i<10) {
         m_result[i]= new Node();  m_result$[i]= new Node();
         //@ set m_result[i].dash= false;         set m_result$[i].dash= true;
         //@ set m_result[i].mate= m_result$[i];  set m_result$[i].mate= m_result[i];
         m_result[i].val= x;  m_result$[i].val= m_result[i].val;
         i++;
      }                                                 }
```

Fig. 2. Self-composed and transformed method `m` from Sect. 1, with JML annotation

has been applied. The self-composed version needs to be annotated with assignments to the ghost fields (written as JML comments with keyword `set`), at the point where the dashed copy has been allocated and is low-visible. Here we do not mate the arrays themselves, since JML doesn't allow ghost fields to be added to arrays, but we do mate their contents. This example verifies in 0.425sec (using the -loopSafe option for soundness) with the elipses replaced by an obvious invariant derived from the postcondition by a standard heuristic (replace constant 10 by variable i).

To illustrate that the transformation is not necessary in general, Fig. 3 shows the running example self-composed but not transformed. The mating assignments are all in the second loop body and this version verifies in 0.529sec. It works because the objects created by the first loop are easily referenced since they are in an array. But whereas the loop invariants needed for Fig. 2 are obtained from the postcondition by simply replacing constant 10 by index variable i, the version in Fig. 3 requires additional invariants (the only ones shown) expressing the absence of aliasing since the allocations are separated in the code. If instead of an array one considers a linked list or other linked structure, it is more difficult to state such invariants.

7 Transforming the Self-composed Command

Terauchi and Aiken propose an interleaving transformation like the one used in the preceding experiment and described in Sect. 1. They show it sound, but in the setting of a simpler language without objects. It depends on conservative analysis that could be obtained by type inference. Their formulation does not suggest an obvious way to extend the results to richer language features or policies. This section sketches how to

```
// Specification same as in previous version...
void Pair_m(int x, int secret, int x$, int secret$) {
    m_result= new Node[10];   m_result$= new Node[10];
    int i= 0;
    //@ maintaining ...(\forall int j,k; 0<=j&&j<k&&k<i ==> m_result[j]!=m_result[k]);
    while (i<10) {
       m_result[i]= new Node();
       m_result[i].val= x;
       i++;
    }
    i= 0;
    //@ maintaining ...(\forall int j,k; 0<=j&&j<k&&k<10 ==> m_result[j]!=m_result[k]);
    //@ maintaining (\forall int j,k; 0<=j&&j<10&&0<=k&&k<i ==> m_result[j]!=m_result$[k]);
    while (i<10) {
       m_result$[i]= new Node();
       m_result$[i].val= x$;
       //@ set m_result[i].dash= false;         set m_result$[i].dash= true;
       //@ set m_result[i].mate= m_result$[i];  set m_result$[i].mate= m_result[i];
       i++;
    }                                           }
```

Fig. 3. Self-composed method m, not transformed

use relational correctness judgements to formulate the interface to the analysis as well as the transformations themselves. Indexing is elided for clarity.

Suppose the goal is to check the simple noninterference property $\Gamma|\Gamma \models S \sim S : \mathsf{Ind}^{vis} \longrightarrow\!\!\triangleright \mathsf{Ind}^{vis}$. After renaming the second copy, Theorem 1 tells us an equivalent partial correctness judgement $\Delta \models \{(\mathsf{Indd}^{vis})^1\}\ S;S'\ \{(\mathsf{Indd}^{vis})^1\}$ where Δ is $\Gamma \uplus \Gamma'$. Instead of directly verifying this, we want S^* such that $\Delta \models \{(\mathsf{Indd}^{vis})^1\}\ S^*\ \{(\mathsf{Indd}^{vis})^1\}$ implies $\Delta \models \{(\mathsf{Indd}^{vis})^1\}\ S;S'\ \{(\mathsf{Indd}^{vis})^1\}$. The requisite transformation can be expressed by relational correctness judgements: the relations express both the notion of equivalence (e.g., modulo renaming) and the conditions under which the transformation is valid (e.g., known initial values, or final values that aren't used). Here is an example judgement that transforms $x := y$ by renaming and exploiting a precondition:

$$x,y : \mathsf{int}\,|\,x',y' : \mathsf{int} \models\ x := y\ \sim\ x' := 0\ : (y = 0 \wedge y = y') \longrightarrow\!\!\triangleright (x = x' \wedge y = y')$$

The situation of interest is complicated by the fact that the program $S;S'$ to be transformed already acts on two copies of Γ. The rule we need for loop transformation includes an antecedent of the form $\Delta|\Delta \models E \sim E' : \mathscr{R} \longrightarrow\!\!\triangleright \ldots$ where $\mathscr{R}$ expresses that the dashed and undashed copies of E have the same value.

To establish the antecedents, the idea of Terauchi and Aiken is to use type inference to find a less precise property of S, namely $\Gamma|\Gamma \models S \sim S : \mathsf{Ind}^V \longrightarrow\!\!\triangleright \mathsf{Ind}^V$ for some $V \subseteq vis$. Type inference would yields this property for all constituent parts including the loop guard E (if it is low; otherwise no transformation is needed). This is now lifted to Δ by a construction applicable to any context Γ: the cartesian square of a predicate, intersected with the identity. For any $\mathscr{P} \subseteq [\![\Gamma]\!]$, define $\mathscr{P} \otimes \mathscr{P} \subseteq [\![\Gamma]\!] \times [\![\Gamma]\!]$ by $(\mathscr{P} \otimes \mathscr{P})\, s\, s' \iff s = s' \wedge \mathscr{P}\, s$.

Taking $\mathscr{R}$ to be $\mathsf{Indd}^V \otimes \mathsf{Indd}^V$, the analysis-based transformations yield $\Delta|\Delta \models S;S' \sim S^* : \mathscr{R} \longrightarrow\!\!\triangleright \mathscr{R}$. Now the desired judgement $\Delta \models \{(\mathsf{Indd}^{vis})^1\}\ S;S'\ \{(\mathsf{Indd}^{vis})^1\}$ (which itself encodes a relation!) is lifted to the level of relations in the squared form $\Delta|\Delta \models S;S' \sim S;S' : (\mathsf{Indd}^{vis})^1 \otimes (\mathsf{Indd}^{vis})^1 \longrightarrow\!\!\triangleright (\mathsf{Indd}^{vis})^1 \otimes (\mathsf{Indd}^{vis})^1$. This can now be composed with the transformation $\Delta|\Delta \models S;S' \sim S^* : \mathscr{R} \longrightarrow\!\!\triangleright \mathscr{R}$ by general transitivity (that is: $\Gamma|\Gamma \models S0 \sim S1 : \mathscr{R}0 \longrightarrow\!\!\triangleright \mathscr{S}0$ and $\Gamma|\Gamma \models S1 \sim S2 : \mathscr{R}1 \longrightarrow\!\!\triangleright \mathscr{S}1$ imply $\Gamma|\Gamma \models S0 \sim S2 : (\mathscr{R}0;\mathscr{R}1) \longrightarrow\!\!\triangleright (\mathscr{S}0;\mathscr{S}1)$). The outcome is a judgement involving composed relations like $(\mathsf{Indd}^{V} \otimes \mathsf{Indd}^{V});((\mathsf{Indd}^{vis})^{1} \otimes (\mathsf{Indd}^{vis})^{1})$. For this process to be useful, the composed relations must boil down to the original noninterference property, which they do owing to a general result about the relational logic.

Theorem 2. *Let Δ abbreviate $\Gamma \uplus \Gamma'$. Suppose $\Delta|\Delta \models S \sim S^{*} : \mathscr{R} \longrightarrow\!\!\triangleright \mathscr{S}$ for some S and S^{*}, where $\mathscr{R}$ and $\mathscr{S}$ are symmetric. Let $\mathscr{P}$ and $\mathscr{Q}$ be predicates on Δ such that $\mathscr{R};(\mathscr{P} \otimes \mathscr{P}) = \mathscr{P} \otimes \mathscr{P}$ and $\mathscr{P} \otimes \mathscr{P} = (\mathscr{P} \otimes \mathscr{P});\mathscr{R}$ (and* mutatis mutandis *for $\mathscr{Q}$). Then $\Delta \models \{\mathscr{P}\}\ S^{*}\ \{\mathscr{Q}\}$ implies $\Delta \models \{\mathscr{P}\}\ S\ \{\mathscr{Q}\}$.*

This can be instantiated with $S := (S;S')$ with S' being a renamed copy of S; moreover $\mathscr{P}$ and $\mathscr{Q}$ encode the desired noninterference property based on Ind^{vis} and $\mathscr{R},\mathscr{S}$ encode the result of type based analysis, e.g., $\mathscr{P}$ is $\mathsf{Indd}^{vis} \otimes \mathsf{Indd}^{vis}$, $\mathscr{R}$ is $\mathsf{Indd}^{V} \otimes \mathsf{Indd}^{V}$ and $\mathscr{Q},\mathscr{S}$ correspond to $\mathscr{P},\mathscr{R}$ but with extended bijections as usual. In the situation described above with $V \subseteq vis$ we have $\mathsf{Indd}^{vis}_{\tau} \subseteq \mathsf{Indd}^{V}_{\tau}$ because larger vis is more restrictive. This yields the requisite absorption properties, e.g., $\mathscr{R};(\mathscr{P} \otimes \mathscr{P}) = \mathscr{P} \otimes \mathscr{P}$. So the Theorem justifies the use of transformations that are sound for Indd^{V} to prove noninterference with respect to *vis*.

8 Discussion

We defined a novel encoding to support self-composition in programs acting on the heap. The encoding is expressed in terms of auxiliary state, specifically ghost fields which are available in specification languages like JML. Theorem 1 says that a relational property is equivalent to a corresponding partial correctness property of a self-composed program. The notion of relational property is general enough to encompass rich declassification policies and also to be used to reason about program transformations. Theorem 2 justifies the use of transformations like those of Aiken and Terauchi [30] which are needed to make the self-composed version amenable to off-the-shelf program verifiers (automatic or interactive), in particular to bring allocations together by merging loops. Preliminary experiments indicate that the encoding and mechanical transformations work smoothly with extant tools.

For modular verification of commands with method calls, it is desirable to transform the program so that a duplicated pair of calls can be brought together and even replaced by a single invocation of a suitably transformed version of the method (like `pair_m` in the example). The transformed version can be proved to satisfy its specification using verification, if necessary, or by security type checking.

The fact that type checking can be used to justify transformations does not mean that the verification technique achieves nothing beyond what can be type checked. Transformation is not always necessary, as illustrated by Fig. 3; similarly, method calls need not be transformed if adequate functional specifications are available. The properties needed to justify transformations can themselves be proved by verification instead of type checking.

Related work. Self-composition is essentially an application of Reynolds' method for proving data refinements [25,13], but data refinement with general heaps has only recently been studied [3] and not yet using this method. Barthe, D'Argenio, and Rezk [7] develop a general theory of self-composition. They sketch a treatment of the heap using separation logic semantics; indistinguishability of abstract lists is used to avoid the issue of pointer renaming. Terauchi and Aiken [30] note that self-composition generalizes to general relational properties (of a single program, in their case). They introduce the transformations we studied in this paper and report good experimental results for deterministic simple imperative programs using the BLAST tool [17]. Correctness of the transformations is proved under reasonable assumptions typical of type systems [30]. But their formulation seems rather specific and it is unclear how to extend it to richer semantics, e.g., where equality may only be up to renaming.

Benton [9] develops relational Hoare logic for a deterministic imperative language and applies it to analysis-based compiler optimizations. Yang [32] develops a relational Separation Logic [26]. The secure flow logic of Amtoft and Banerjee [2] can be seen as a relational Hoare logic for the special case of noninterference; it is extended to heaps in [1] and applied to declassification in [5]. The focus of Amtoft et al. is automated static analysis; an abstract interpretation for the heap is presented in "logical form".

Darvas, Hähnle, and Sands [11] use the KeY tool for interactive verification of noninterference. It uses a dynamic logic for Java, which is more expressive than ordinary partial correctness assertions, allowing in particular existential quantification over weakest-precondition statements. For nondeterministic S, the self-composed version $S;S'$ does not capture relational properties, but they can be captured using the conjugate predicate transformer $\neg\mathsf{wlp}\neg$ [16]. This suggests it would be interesting to explore the use of dynamic logic for relational properties of nondeterministic programs.

Dufay, Felty, and Matwin [15] use the self-composition technique to check noninterference for data mining algorithms implemented in Java. They use the Krakatoa tool, based on the Coq theorem prover and using JML [19]. They extend JML with special notations to refer to the two copies of program state and extend Krakatoa to generate special verification conditions. The paper does not give much detail about the heap encoding. To prove that noninterference is enforced by their security type inference system for an ML-like language, Pottier and Simonet [24] extend the language with a pairing construct and semantics that encodes two runs of a program as one.

Future work. Although our small experiments worked without difficulty, there is an impediment to scaling up the idea. The mating condition appears in preconditions and postconditions of every method, so effectively it is an object invariant. But in general it needs to be fully ramified to all fields of all reachable objects. This is tricky because in languages like JML specifications must respect the visibility rules of the language and therefore cannot refer to "all" fields. One possibility is to define the `mate` predicate as a pure method, overridden in each class—it constrains the fields visible in that class, by invoking `mate` on class type fields and invoking `super.mate` for inherited ones. Care is needed, owing to cycles in the heap; moreover reasoning about pure methods in specifications is not well supported in current verifiers [12]. Another approach is to formulate mating in a decentralized way using explicit object invariants, which are a topic of active research on modular reasoning [20]. The mating invariant is incompatible with ownership-based invariants (or model fields) but it is compatible with friendship-based invariants [23]; friendship is slated to be added to Spec# and is available as an undocumented feature in the tool of Cees Pierik (www.cs.uu.nl/groups/IS/vft/).

Theorem 2 characterizes the useful transformations and some examples have been mentioned, but it remains to develop a full set of transformations. Benton's proof system could be extended to incorporate the heap and also method calls, and syntax added to manipulate the embeddings. The idea is to derive specialized transformations like those needed for self-composition from very powerful general rules that can be formulated in a relational setting.

Self-composition generalizes to simulations for data abstraction. In particular, we plan to investigate use of the encoding for establishing the antecedent in the representation independence property [3].

Acknowledgement. Thanks to Dustin Long for help with the ESC/Java experiments. Thanks to Dustin and to Mike Barnett for adapting the Java experiments to Spec# and trying them with the Boogie verifier. Insightful comments from reviewers significantly improved the presentation.

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.
2. T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *Static Analysis Symposium (SAS)*, 2004.
3. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, Nov. 2005.
4. A. Banerjee and D. A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.
5. A. Banerjee and D. A. Naumann. A logical account of secure declassification (extended abstract). Submitted, 2006.
6. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004), Revised Selected Papers*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
7. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 100–114, 2004.
8. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
9. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–25, 2004.
10. E. S. Cohen. Information transmission in sequential programs. In A. K. J. Richard A. DeMillo, David P. Dobkin and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
11. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.

12. A. Darvas and P. Müller. Reasoning about method calls in JML specifications. In ECOOP workshop on *Formal Techniques for Java-like Programs*, 2005.
13. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
14. D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
15. G. Dufay, A. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In *Conference on Automated Deduction (CADE)*, 2005.
16. D. Gries. Data refinement and the tranform. In M. Broy, editor, *Program Design Calculi*. Springer, 1993. International Summer School at Marktoberdorf.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 58–70, 2002.
18. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Computing Science Institute, University of Nijmegen, 2003. In *International Symposium on Software Security*, volume 3233, of *LNCS*, pages 134–153. Springer, 2003.
19. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284. Springer, 2003.
20. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, Department of Computer Science, ETH Zurich, 2004.
21. A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.
22. D. A. Naumann. Verifying a secure information flow analyzer. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics TPHOLS*, volume 3603 of *LNCS* pages 211–226. Springer, 2005.
23. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. To appear in *Theoretical Computer Science*, 2006.
24. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.
25. J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.
26. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Logic in Computer Science (LICS)*, pages 55–74, 2002.
27. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
28. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-order and Symbolic Computation*, 14(1):59–91, 2001.
29. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2005.
30. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *12th International Static Analysis Symposium (SAS)*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
31. D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, volume 1214 in *LNCS*, pages 607–621. Springer, 1997.
32. H. Yang. Relational separation logic. *Theoretical Comput. Sci.*, 2004. To appear.