Relational program logics in decomposed style

Lennart Beringer *
Department of Computer Science, Princeton University,
35 Olden Street, Princeton NJ 08540
eberinge@cs.princeton.edu

Abstract

Suppose you have developed a Hoare logic for your favorite programming language. You have justified the logic by defining an operational model of the language and constructing a soundness proof that interprets triples as partial correctness assertions. Now you want to interpret program analyses, validate security properties such as noninterference, or justify program transformations. You observe that all these properties are relational: they are naturally phrased over *pairs* of executions, for programs that are either identical or closely related. Is your program logic up to the task?

This article shows how to formally decompose termination-insensitive relational program logics into judgements from unary logics. We develop relational predicate transformers, present laws that govern their decomposition along the phrase structure, and relate them to their unary counterparts. We apply our findings to justify variants of Benton's Relational Hoare Logic (RHL) for a language with objects, extend the logics to auxiliary state, derive a noninterference analysis in the style of Banerjee-Naumann, and develop relational interpretations of separation logic. As related executions do not have to refer to the same program syntax or employ the same notion of state, decomposition can in principle be applied to cross-language verification problems, as long as suitable one-execution logics exist.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory — Semantics; F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs — Logics of programs; F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Mathematical Logic — Mechanical theorem proving

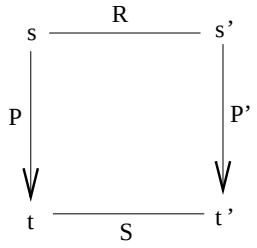
General Terms Languages, Theory, Verification

Keywords Relational program logics, Decomposition, Noninterference, Self-composition, Separation logic, Relational predicate transformers

1. Introduction

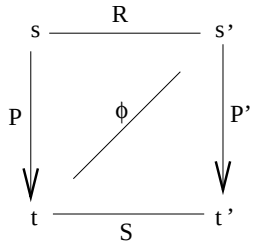
Verification formalisms and tools based on Hoare logics are typically designed with one-execution properties in mind: their partial or total correctness interpretation involves a single operational judgement.

However, many program properties are relational: they are naturally phrased as statements over pairs of executions, stipulating that the terminal states of two runs stand in relation S whenever the initial states are in relation R. Examples include not only "obviously relational" properties such as program transformations or noninterference [29], but also extensional interpretations of types, and program analyses that may be found in almost any compiler [9]. This article shows that unary verification technology nevertheless suffices in many cases for certifying relational properties in the foundational sense [3].



We introduce a novel technique called *relational decomposition* which reduces the task of proving a simulation diagram to the task of proving two separate unary assertions, one for $s \xrightarrow{P} t$ and one for $s' \xrightarrow{P'} t'$.

These assertions arise from witnesses ϕ that relate terminal states of executions to the left with initial states of executions to the right. Witnesses of this form were first introduced by Beringer and Hofmann [12] in order to derive interpretations of type systems for noninterference according to the approach of self-composition [8, 15, 20, 32]. The present paper establishes the general applicability of the technique, and some of its core properties:



1. We show that relational decomposition applies to arbitrary pairs of transition systems, rather than being limited to relational properties of single programs, or relationships between programs from the same language. Indeed, primed states in the above diagram are not required to be of the same type as non-primed states, and P and P' may be from different languages. Any witness that makes the two unary properties valid guarantees the simulation property (soundness).
2. We establish the completeness of decomposition in the general case: witnesses exist whenever the simulation property holds. The space of witnesses ϕ is characterized by an inclusion property between relational predicate transformers. We present laws that relate these transformers to their unary counterparts.

* Supported in part by NSF grant CNS-0910448 and AFOSR award FA9550-09-1-0138.

3. We extend relational decomposition by a generic parameterization mechanism which can be instantiated to communicate arbitrary additional information between pre- and postrelations.
4. We transfer relational decomposition to a setting where the operational model contains explicit error states, based on unary verification technology that exposes such states in assertions.
5. We instantiate relational decomposition to a number of relational and nonrelational formalisms, illustrating the previous items in concrete settings, for a language of simple objects. Specifically, we outline (i) variants of Benton's relational Hoare logic (RHL, [9]) (ii) an interpretation of a type system for noninterference, for a security model in the style of Banerjee-Naumann [5], and (iii) relational interpretations of unary and relational separation logic [26, 27, 36].

The concrete instantiations are derived from suitable unary logics, and combine witnesses associated with subphrases to witnesses for compound phrases exactly as in [12], highlighting that the decomposition is in fact syntax- rather than type-driven.

The formal setup and principle of decomposition are as follows.

1.1 The principle of decomposition

For the purpose of this paper, a transition system $\mathcal{T}$ over state space $\mathcal{S}$ and labels (thought of as program phrases) $\mathcal{P}$ is a ternary relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S}$.

A transition system $\mathcal{T}$ gives rise to a one-execution specification system with assertions $A \in \mathcal{A} \equiv \mathcal{S} \Rightarrow \mathcal{S} \Rightarrow \mathbb{T}$ (where $\mathbb{T}$ is the set of truth values), interpreted by the partial-correctness condition

$$\models^{\mathcal{T}} P : A \quad \equiv \quad \forall\, s\, t.\, (s, P, t) \in \mathcal{T} \rightarrow A\, s\, t.$$

The use of binary relations for one-execution properties follows the VDM discipline [19] and simplifies the treatment of auxiliary variables and quantifiers when compared to separate pre- and postconditions.

The formal notion of simulation is as follows.

DEFINITION 1. *For $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S}$ and $\mathcal{T}' \subseteq \mathcal{S}' \times \mathcal{P}' \times \mathcal{S}'$, let $R, S \subseteq \mathcal{S} \times \mathcal{S}'$. Programs $P \in \mathcal{P}$ and $P' \in \mathcal{P}'$ are $R \Rightarrow S$-similar, notation $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Rightarrow S$, if for all s, s', t, and t' with $(s, P, t) \in \mathcal{T}$ and $(s', P', t') : \mathcal{T}'$, sRs' implies tSt'.*

Unless explicitly remarked otherwise, we follow the convention that nonprimed entities (states, phrases,...) refer to $\mathcal{T}$ and primed entities to $\mathcal{T}'$. Relations from $\mathcal{S} \times \mathcal{S}'$ are often written infix.

Examples of properties of this shape for $\mathcal{S} = \mathcal{S}'$ include determinism (choose R and S to be equality), liveness of variables (choose R and S to be equality on live-in variables) and slicing, program transformations, and termination-insensitive versions of properties considered by [9]. Further instances arise when the condition $\mathcal{S} = \mathcal{S}'$ is dropped, including variations of compiler correctness, refinement, and abstract interpretation.

The core of relational decomposition consists of the unary assertions defined by the operators Dec_{L} and Dec_{R}

$$\begin{aligned} Dec_{\mathsf{L}}\ R\ \phi\ s\ t &= \forall s'.\, sRs' \rightarrow t\phi s' \\ Dec_{\mathsf{R}}\ S\ \phi\ s'\ t' &= \forall t.\, t\phi s' \rightarrow tSt' \end{aligned}$$

Given some $\phi \subseteq \mathcal{S} \times \mathcal{S}'$, operator Dec_{L} constructs a unary assertion for the upper left triangle, i.e. for the execution of P. In fact, the construction uniformly applies for all types $\mathcal{S}'$ subject to $R \subseteq \mathcal{S} \times \mathcal{S}'$. Similarly, Dec_{R} constructs a unary assertion for the lower right triangle (the execution of P'), uniformly for types $\mathcal{S}$ with $S \subseteq \mathcal{S} \times \mathcal{S}'$.

The operators are motivated by the following result.

LEMMA 1. (Soundness) *Suppose $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ \phi$ and $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ \phi$. Then $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Rightarrow S$.*

Thus, the task of verifying $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Rightarrow S$ is reduced to the task of exhibiting an arbitrary ϕ that satisfies the two one-execution properties.

How can one find suitable witnesses ϕ? In [12], we considered the special case of (termination-insensitive) *noninterference* [29], where and R and S are state indistinguishability relations $\sim_X$ with respect to sets X of ("low" security) program variables. We showed that suitable witnesses ϕ can be obtained mechanically from type systems for noninterference from the literature [18, 33], in a rule-by-rule fashion. We also showed the completeness of the characterization, by exhibiting a witness whenever noninterference holds. This constructed witness was phrased in terms of an operational judgement, similar to the standard argument in a proof of (relative) completeness proofs of program logics [21, 25]. However, we did not study whether the witnesses ϕ are unique, and our analysis was restricted to noninterference. Specifically, the development was hampered by the identification of $\mathcal{T}$ with $\mathcal{T}'$, and the fact that $Dec_{\mathsf{L}}\ R\ \phi$ and $Dec_{\mathsf{R}}\ S\ \phi$ were combined to a single assertion.

1.2 Synopsis

Section 2 presents a more detailed analysis of decomposition witnesses, for general simulation diagrams rather than noninterference. We then derive a decomposed version of relational Hoare logic, for an extension of Benton's language with heap objects.

Section 3 introduces some constructions on transition systems; these are subsequently used to derive a parameterization mechanism. An application to noninterference for the language with objects ensues, where partial bijections hide the nondeterminism of object allocation and are communicated from pre- to postrelations.

Section 4 considers simulations for operational semantics with explicit error states. We develop a unary program logic that exposes error states in assertions while at the same time retaining the applicability of decomposition. Having adapted the parameterized logic from Section 3 to this setting, we outline relational interpretations of unary and relational separation logic.

We conclude with a discussion of future and related work. The entire development presented has been formalized in Isabelle/HOL. A snapshot of the development is available at [35].

Before continuing, we emphasize that in contrast to Benton's and Yang's logics, all our interpretations are termination-insensitive: simulations are vacuously fulfilled if either execution fails to terminate. This design decision is motivated by the fact that even in the unary setting, proof techniques for termination (appropriate measures, i.e. variants) are rather different from those for partial-correctness properties (invariants). A second reason is that applications such as compiler verification actually relax termination-sensitivity to at least an asymmetric form [23]. Thus, termination appears sufficiently orthogonal to the functional aspects of relational behaviour to be treated separately. Nevertheless, we acknowledge that (and point out where) our design decision has repercussions on the proof rules we are able to derive.

2. Decomposition of relational Hoare logics

2.1 Relational decomposition operators

We first observe that Dec_{L} is covariant in ϕ and contravariant in R, i.e. for $\phi \subseteq \psi$ and $Q \subseteq R$, $Dec_{\mathsf{L}}\ R\ \phi\ s\ t$ implies $Dec_{\mathsf{L}}\ Q\ \psi\ s\ t$, while Dec_{R} is covariant in S and contravariant in ϕ, i.e. for $\psi \subseteq \phi$ and $S \subseteq T$, $Dec_{\mathsf{R}}\ S\ \phi\ s'\ t'$ implies $Dec_{\mathsf{R}}\ T\ \psi\ s'\ t'$. We also note the identity $Dec_{\mathsf{R}}\ S\ \phi = Dec_{\mathsf{L}}\ \phi^{-1}\ S^{-1}$.

Next, we characterize the witnesses suitable for establishing Lemma 1. To this end, consider the operators

$$\begin{aligned} \phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R &= \{(t, s').\, \exists s.\, (s, P, t) \in \mathcal{T} \wedge sRs'\} \\ \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S &= \{(t, s').\, \forall t'.\, (s', P', t') \in \mathcal{T}' \rightarrow tSt'\} \end{aligned}$$

The former constructs a candidate for ϕ according to the upper left triangle in the diagram, given R and P. The latter constructs a (in general different) candidate ϕ according to the lower right triangle in the diagram, given S and P'.

By construction, these operators yield valid specifications for their defining triangles, i.e. they satisfy $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R)$ and $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S)$. Furthermore, they are covariant in their second argument, satisfy the laws in Figure 1, and relate to each other by $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ (\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ T) \subseteq \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ (\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ T)$. In particular, $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R$ is the *least* relation satisfying the left triangle, and $\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S$ is the *greatest* relation satisfying the right triangle:

LEMMA 2. *If* $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ \phi$ *then* $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R \subseteq \phi$. *If* $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ \phi$ *then* $\phi \subseteq \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S$.

Thus, any witness ϕ from Lemma 1 is sandwiched between the two operators, i.e. satisfies $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R \subseteq \phi \subseteq \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S$.

Conversely, either operator is suitable as a witness:

LEMMA 3. (Completeness) *Suppose* $\models_{\mathcal{T}}^{\mathcal{T}'} P \sim P' : R \Rightarrow S$. *Then*

- $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R)$ *and* $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R)$
- $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S)$ *and* $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S)$.

Combining Lemmas 2 and 3, we obtain that $\models_{\mathcal{T}}^{\mathcal{T}'} P \sim P' : R \Rightarrow S$ is equivalent to the inclusion $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R \subseteq \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S$.

While the operators are defined from the relational perspective, they are intimately connected with the unary transformers

Strongest postcondition :
$$SP_P^{\mathcal{T}}(X) = \{t.\ \exists\ s \in X.\ (s,P,t) \in \mathcal{T}\}$$
Weakest liberal precondition :
$$WLP_{P'}^{\mathcal{T}'}(Y') = \{s'.\ \forall\ t'.\ (s',P',t') \in \mathcal{T}' \to t' \in Y'\}$$

where X and Y' are state sets from $\mathcal{T}$ and $\mathcal{T}'$, respectively. Each operator arises from the appropriate unary transformer if one keeps the state in the opposite execution invariant:

$$\begin{array}{rcl} \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R & = & \{(t,s').\ t \in SP_P^{\mathcal{T}}(\{s.\ sRs'\})\} \\ \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S & = & \{(t,s').\ s' \in WLP_{P'}^{\mathcal{T}'}(\{t'.\ tSt'\})\}) \end{array}$$

Thus, $R \Rightarrow S$-similarity is soundly and completely characterized by the inclusion[1]

$$\begin{array}{l} \{(t,s').t \in SP_P^{\mathcal{T}}(\{s.sRs'\})\} \\ \qquad \subseteq \{(t,s').s' \in WLP_{P'}^{\mathcal{T}'}(\{t'.tSt'\})\}. \end{array} \quad (1)$$

Finally, we note that the relational transformers

Strongest postrelation :
$$SR_{P,P'}^{\mathcal{T},\mathcal{T}'}(R) = \{(t,t').\ \exists\ s\ s'.\ (s,P,t) \in \mathcal{T} \wedge (s',P',t') \in \mathcal{T}' \wedge sRs'\}$$
Weakest liberal prerelation :
$$WLR_{P,P'}^{\mathcal{T},\mathcal{T}'}(S) = \{(s,s').\ \forall\ t\ t'.\ (s,P,t) \in \mathcal{T} \to (s',P',t') \in \mathcal{T}' \to tSt'\}$$

satisfy the following properties.

[1] Replacing $WLP_{P'}^{\mathcal{T}'}(Y)$ by the weakest precondition operator
$$WP_{P'}^{\mathcal{T}'}(Y') = \{s'.\ \exists\ t' \in Y'.\ (s',P',t') \in \mathcal{T}'\}$$
yields a similar characterization for simulation diagrams where each $\mathcal{T}$-execution implies the existence of a corresponding $\mathcal{T}'$-execution, i.e. the property that $(s,P,t) \in \mathcal{T}$ and sRs' imply the existence of some t' with $(s',P',t') : \mathcal{T}'$ and tSt'. We do not study this variant any further in the present paper but observe that diagrams of this shape are considered in Leroy's CompCert project [23] and in Schmidt's work on relational abstract interpretation [30].

LEMMA 4. *We have*

1. $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ (WLR_{P,P'}^{\mathcal{T},\mathcal{T}'}(S)) \subseteq \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S$
2. $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R \subseteq \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ (SR_{P,P'}^{\mathcal{T},\mathcal{T}'}(S))$
3. $WLR_{P,P'}^{\mathcal{T},\mathcal{T}'}(S) = \{(s,s').\ s' \in WLP_{P'}^{\mathcal{T}'}(\{t'.\ s \in WLP_P^{\mathcal{T}}(\{t.\ tSt'\})\})\} = \{(s,s').\ s \in WLP_P^{\mathcal{T}}(\{t.\ s' \in WLP_{P'}^{\mathcal{T}'}(\{t'.\ tSt'\})\})\}$
4. $SR_{P,P'}^{\mathcal{T},\mathcal{T}'}(R) = \{(t,t').\ t' \in SP_{P'}^{\mathcal{T}'}(\{s'.\ t \in SP_P^{\mathcal{T}}(\{s.\ sRs'\})\})\} = \{(t,t').\ t \in SP_P^{\mathcal{T}}(\{s.\ t' \in SP_{P'}^{\mathcal{T}'}(\{s'.\ sRs'\})\})\}$

2.2 Application: Relational Hoare Logic in decomposed form

We instantiate the above development to the situation where $\mathcal{T}$ and $\mathcal{T}'$ coincide and are equal to the operational judgement of an imperative language with objects.

We assume infinite and distinct categories of variables $x, y \in \mathcal{X}$, field identifiers $f \in \mathcal{F}$, class identifiers $c \in \mathcal{C}$, and locations $\ell \in \mathcal{L}$. The space of finite partial functions from A to B is denoted by $A \rightharpoonup B$, and the space of total functions by $A \Rightarrow B$. Operations and constructions such as update, domain, and range, are defined and denoted in the standard fashion. A value $v \in \mathcal{V}$ is either an integer value i, a location ℓ, or Null. Value expressions $e \in \mathcal{E}$ are value constants, variables, or binary operators, while boolean expressions b are binary predicates over values.

The syntax is given by the grammar

$$\begin{array}{rcl} C \in \mathcal{P} ::= & \mathbf{Skip} \mid x{:=}e \mid x := \mathbf{new}\ c\ \iota \mid x{:=}y.f \mid x.f{:=}e \mid \\ & C;D \mid \mathbf{While}\ b\ \mathbf{do}\ C \mid \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \end{array}$$

where $\iota \in \mathcal{F} \rightharpoonup \mathcal{E}$ specifies the initialization of fields in the absence of a formalized class system.

The operational semantics is defined over the semantic domains of objects, heaps, stores, and states

$$\begin{array}{rclcrcl} o \in \mathcal{O} & \equiv & \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V}) & \qquad & s \in \mathcal{R} & \equiv & \mathcal{X} \Rightarrow \mathcal{V} \\ h \in \mathcal{H} & \equiv & \mathcal{L} \rightharpoonup \mathcal{O} & & \sigma \in \Sigma & \equiv & \mathcal{R} \times \mathcal{H} \end{array}$$

[2] We write $[\![e]\!]_s$ and $[\![b]\!]_s$ for the (heap-independent) evaluation of value and boolean expressions, respectively, and map the former operation over initialization maps in the expected manner.

The transition system $\mathcal{T}_{\mathsf{Obj}} \subseteq \Sigma \times \mathcal{P} \times \Sigma$, with pretty-printed judgements $\sigma \xrightarrow{C} \tau$, is defined essentially as in [34] for the heap-free instruction forms, and by rules such as

$$\textsc{OpNew}\ \frac{\ell \notin locs\ (s,h)}{(s,h) \xrightarrow{x:=\mathbf{new}\ c\ \iota} (s[x \mapsto \ell], h[\ell \mapsto (c, [\![\iota]\!]_s)])}$$

for the object-manipulating instructions, where $locs\ \sigma$ denotes the set of all locations ℓ occurring in σ. For now, attempts to access dangling pointers or undefined fields of allocated objects result in the absence of an execution.

In accordance with the setup of Section 1.1, we introduce a unary logic with judgements of the form $\triangleright C : A$ where $A : \Sigma \Rightarrow \Sigma \Rightarrow \mathbb{T}$. Selected proof rules are given in Figure 2. The logic can be proven sound and complete (the latter relative to the ambient logic HOL) using standard techniques [21, 25].

THEOREM 1. $\triangleright C : A \Longleftrightarrow \models^{\mathcal{T}_{\mathsf{Obj}}} C : A$.

[2] The use of $s, t, \ldots$ for stores as well as for abstract states of transition systems should not lead to confusion, as instantiations to the concrete language are always discussed separately from the abstract treatment.

$$\begin{array}{llll} \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ (R \cap T) & \subseteq & \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R \cap \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ T & \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ (U \cup T) = \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ U \cup \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ T \quad \phi_{\mathsf{L}}^{\mathcal{T}}\ P\ \emptyset = \emptyset \\ \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ (S \cap T) & \supseteq & \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S \cap \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ T & \phi_{\mathsf{R}}^{\mathcal{T}'}\ P\ (U \cup T) \supseteq \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ U \cup \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ T \quad \phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ \emptyset = \{(t,s').\forall t'.(s',P',t') \notin \mathcal{T}'\} \end{array}$$

Figure 1. Interaction between $\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ R$ and $\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ S$ and set operations

$$\frac{}{\rhd x{:=}e : \lambda\,(s,h)\ \tau.\ \tau = (s[x \mapsto [\![e]\!]_s], h)} \qquad \frac{}{\rhd x := \mathbf{new}\ c\ \iota : \lambda\,(s,h)\ \tau.\ \exists\,\ell \notin locs\ (s,h).\ \tau = (s[x \mapsto \ell], h[\ell \mapsto (c, [\![\iota]\!]_s)])}$$

$$\frac{}{\rhd x{:=}y.f : \lambda\,(s,h)\ \tau.\ \exists\,\ell\ c\ F\ v.\quad \begin{array}{l} s\ y = \ell \wedge h\ l = (c,F) \wedge \\ Ff = v \wedge \tau = (s[x \mapsto v], h) \end{array}} \qquad \frac{\rhd C : A \qquad \rhd D : B}{\rhd \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D : \lambda\,(s,h)\ \tau.\ \begin{array}{l} ([\![b]\!]_s \to A\ (s,h)\ \tau) \wedge \\ (\neg [\![b]\!]_s \to B\ (s,h)\ \tau) \end{array}}$$

$$\frac{\rhd C : A \qquad \rhd D : B}{\rhd C;D : \lambda\,\sigma\ \tau.\ \exists\,\rho.\ A\ \sigma\ \rho \wedge B\ \rho\ \tau} \qquad \frac{\rhd C : B \qquad \forall\,s\ h.\ \neg [\![b]\!]_s \to A\ (s,h)\ (s,h) \qquad \forall\,s\ h\ \rho\ \tau.\ [\![b]\!]_s \to B\ (s,h)\ \rho \to A\ \rho\ \tau \to A\ (s,h)\ \tau}{\rhd \mathbf{While}\ b\ \mathbf{do}\ C : \lambda\,\sigma\ t\ k.\ A\ \sigma\ (t,k) \wedge \neg [\![b]\!]_t}$$

Figure 2. Selected proof rules for logic $\rhd C : A$.

Instantiating the relational decomposition operators to $\mathcal{T} = \mathcal{T}_{\mathsf{Obj}}$ and/or $\mathcal{T}' = \mathcal{T}_{\mathsf{Obj}}$ yields decomposition laws such as

$$\begin{array}{rcl} \phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C;D\ R & = & \phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ D\ (\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ R) \\ \phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C';D'\ S & = & \phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C'\ (\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ D'\ S) \\ WLR_{C;D,C';D'}^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}(S) & = & WLR_{C,C'}^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}((WLR_{D,D'}^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}(S))) \end{array}$$

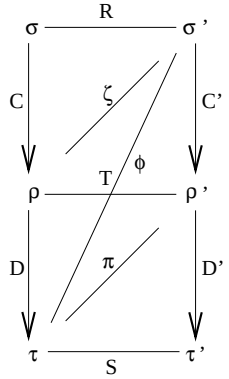
where in the first two cases, the type of the opposite transition system is only constrained by the type of the relations R and S.

Instantiating both transition systems with $\mathcal{T}_{\mathsf{Obj}}$, we now derive a relational logic with judgements $C \sim C' : R \Rightarrow S$, which we interpret (in contrast to Benton [9], but in accordance with Definition 1) in the termination-insensitive style. By virtue of the previous section, several formal interpretations of these judgements are compatible with this interpretation. The derivability from the unary program logic is most explicit if we define $C \sim C' : R \Rightarrow S$ to be a shorthand for

$$\exists\phi.\ \rhd\ C : Dec_{\mathsf{L}}\ R\ \phi \wedge \rhd C' : Dec_{\mathsf{R}}\ S\ \phi \tag{2}$$

and then establish the proof rules as derived lemmas. Figure 3 shows some proof rules for pairs of structurally identical phrases. These are similar to the rules given (for the heap-free fragment of the language) by Benton [9]. As in loc. cit., the loop rule is restricted to the case where both iterations proceed in lock-step. Our rule for conditionals (not shown) allows the executions to proceed along different control paths and consequently has hypotheses for all four possible combinations. The simplified rule that enforces identical evaluation of the branches arises as a special case.

The derivation of the rules exhibits witnesses ϕ as mandated by equation (2). By the results of the previous section, witnesses for the axioms may be chosen as $\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ R$ or $\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C'\ S$, or any relation between the two. Witnesses for compound phrases are constructed from the constituents' witnesses using the same constructions (modulo the addition of the heap) as in [12]. For example, the witness for pairs of compositions arises from the witnesses of the constituent squares as $\phi = \{(\tau,\sigma').\ \exists\,\rho.\ \rho\zeta\sigma' \wedge (\forall\,\rho'.\ \rho T\rho' \to \tau\pi\rho')\}$. The witness for the while rule, $\Phi_{(b',R,\phi)}^{\mathtt{While}}$, is constructed as indicated in Figure 3, where ϕ is the witness of $C \sim C' : U \Rightarrow R$. All proofs for compound phrases respect decomposition in that no proof of a conjunct Dec_{L} . . uses a hypothesis for Dec_{R} . ., and vice versa.



In addition to these rules, we have derived rules where the two phrases may be of different shape, including Benton's rules of falsity, consequence, common branch elimination, and dead code elimination – see Figure 4. Carrying a unary judgement in the hypothesis, the dead-code rule applies to arbitrary phrases C whereas Benton only considers the specializations for assignment and while. Conclusions of DEADL may be promoted to phrase compositions using COMPSKIP. We omit the similar rules for handling dead code and common branches in phrases to the right of $\sim$. Rule UNARY injects a pair of unary judgements into the relational world. This rule is unsound in the termination-sensitive setting of [9]. On the other hand, Benton's rule of transitivity $\dfrac{C \sim C' : R \Rightarrow S \quad C' \sim C'' : R \Rightarrow S \qquad PER(R \Rightarrow S)}{C \sim C'' : R \Rightarrow S}$ where $PER(R \Rightarrow S)$ indicates that the function space $R \Rightarrow S$ is a partial equivalence relation, is unsound in the termination-insensitive setting, as the hypotheses are vacuously satisfied if C' diverges but C and C'' converge. As decomposition witnesses orientate the simulation relation, Benton's rule of symmetry $\dfrac{C \sim C' : R \Rightarrow S \qquad PER(R \Rightarrow S)}{C' \sim C : R^{-1} \Rightarrow S^{-1}}$ can only be derived if we exploit the semantic symmetry of the simulation relation and use the formal completeness of the program logic. An alternative is to modify the interpretation, by conjoining property (2) with

$$\exists\psi.\ \rhd\ C' : Dec_{\mathsf{L}}\ R^{-1}\ \psi \wedge \rhd C : Dec_{\mathsf{R}}\ S^{-1}\ \psi. \tag{3}$$

The resulting interpretation is immediately symmetric and also allows the derivation of the rules above, except for transitivity.

Completeness is also used when deriving rules that contain conclusions with phrases that are subphrases of phrases in hypotheses, thus reversing the standard subphrase orientation that is obeyed by our unary logic. An instance of this phenomenon are **Skip**-elimination rules like $\dfrac{\mathbf{Skip};C \sim C' : R \Rightarrow S}{C \sim C' : R \Rightarrow S}$. An alternative to the use of the formal completeness result would be to work entirely at the level of semantic validity.

THEOREM 2. *The rules in Figures 3 and 4 are derivable as discussed and thus sound with respect to Definition 1.*

$$\frac{R = \{((s,h),(s',h')).\ \forall\, \ell \notin \mathit{locs}\ (s,h).\forall\, \ell' \notin \mathit{locs}\ (s',h').\ (s[x \mapsto \ell], h[\ell \mapsto (c, [\![\iota]\!]_s)])S(s'[x' \mapsto \ell'], h'[\ell' \mapsto (c', [\![\iota']\!]_{s'})])\}}{x := \mathbf{new}\ c\ \iota \sim x' := \mathbf{new}\ c'\ \iota' : R \Rightarrow S}$$

$$\frac{R = \{((s,h),(s',h')).\ \forall\, \ell\ \ell'\ c\ c'\ F\ F'\ v\ v'.\ \ (s\ y = \ell \wedge s'\ y' = \ell' \wedge h\ \ell = (c,F) \wedge h'\ \ell' = (c',F') \wedge F\ f = v \wedge F'\ f' = v') \longrightarrow (s[x \mapsto v], h)S(s'[x' \mapsto v'], h')\}}{x{:=}y.f \sim x'{:=}y'.f' : R \Rightarrow S}$$

$$\frac{C \sim C' : U \Rightarrow R \qquad R = T \cap \{((s,h),(s',h')).\ [\![b]\!]_s = [\![b']\!]_{s'}\} \qquad U = R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \qquad S = R \cap \{((s,h),(s',h')).\ \neg[\![b]\!]_s\}}{\mathbf{While}\ b\ \mathbf{do}\ C \sim \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Rightarrow S}$$

The operator $\Phi^{\mathtt{While}}_{(b',R,\phi)}$ is defined as the least fixed point of the (monotone) functional

$$\psi \mapsto \big\{(\tau,(t',k')).\ \ ([\![b']\!]_{t'} \to (\exists\, \sigma.\ \sigma\phi(t',k') \wedge (\forall\, \sigma'.\ \sigma R\sigma' \to \tau\psi\sigma'))) \wedge (\neg[\![b']\!]_{t'} \to \tau R(t',k'))\ \ \big\}$$

Figure 3. RHL rules for identically shaped phrases (excerpt)

$$\textsc{ComBrL}\ \frac{\begin{array}{cc} C \sim C' : U \Rightarrow S & U = R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \\ D \sim C' : T \Rightarrow S & T = R \cap \{((s,,h),(s',h')).\ \neg[\![b]\!]_s\}\end{array}}{\mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \sim C' : R \Rightarrow S} \qquad \textsc{DeadL}\ \frac{\rhd C : A \quad \forall\, \sigma\ \sigma'\ \tau.\ A\ \sigma\ \tau \to \sigma R\sigma' \to \tau S\sigma'}{C \sim \mathbf{Skip} : R \Rightarrow S}$$

$$\textsc{False}\ \frac{}{C \sim C' : \emptyset \Rightarrow S} \qquad \textsc{CompSkip}\ \frac{C \sim \mathbf{Skip} : R \Rightarrow T \quad D \sim D' : T \Rightarrow S}{C;D \sim D' : R \Rightarrow S} \qquad \textsc{Unary}\ \frac{\begin{array}{c}\rhd C : A \qquad \rhd C' : A' \\ R = \{(\sigma,\sigma').\ \forall\, \tau\ \tau'.\ A\ \sigma\ \tau \to A'\ \sigma'\ \tau' \to \tau S\tau'\}\end{array}}{C \sim C' : R \Rightarrow S}$$

$$\textsc{Sub}\ \frac{C \sim C' : R \Rightarrow S \quad R' \subseteq R \quad S \subseteq S'}{C \sim C' : R' \Rightarrow S'} \qquad \textsc{SetOp}\ \frac{C \sim C' : R \Rightarrow S \quad C \sim C' : T \Rightarrow U \quad R' = R \odot T \quad S' = S \odot U \quad \odot \in \{\cup, \cap\}}{C \sim C' : R' \Rightarrow S'}$$

Figure 4. Nonsynchronous RHL rules (excerpt)

2.3 Discussion: auxiliary state and self-composition

Self-composition [8] solves the certification problem for noninterference of C by reducing it to a one-execution property of the program $C; C'$ where C' arises from C by replacing all program variables x in C by fresh copies x'. Written in pre-/postcondition style, the specification for $C; C'$ then is $\{\sim_L\}\ C; C'\{\sim_L\}$ where for some fixed ("low") subset L of nonprimed variables the predicate $\sim_L$ is defined as $\{s.\ \forall x \in L.\ s\ x = s\ x'\}$. The validity of this approach follows from the observation that the execution of the first half of the self-composed program, i.e. the execution of C, neither modifies nor reads primed variables, whereas the execution of C' neither reads nor modifies nonprimed variables. Comparing self-composition with decomposition, we see that a witness ϕ amounts to an assertion applicable at the point of self-composition, i.e at the state where C has executed but C' has not started yet. The orientation of the witness corresponds to the (in principle arbitrary) order in which the two copies are self-composed. The relations R and S are $\sim_L$, but the entire situation is formulated without duplicating variables or code, purely semantically. This interpretation also explains the inclusion property of witnesses, as equation (1) precisely captures the possible applications of the consequence rule at the point of self-composition.

Terauchi and Aiken observed that the efficiency of verification based on self-composition is improved if the self-composition operation is applied only to small program fragments [32]. They also demonstrated that type systems for non-interference yield appropriate transformation rules. These transformations push self-composition towards the leaves of the syntax tree, in a manner that resembles our decomposition rules for compound phrases.

Decomposition triangles resemble the use of universally quantified state in Kleymann's formal treatment of auxiliary variables [21], an effect that is particularly apparent in the dead-code rule. Formally, our universal quantification concerns states in the opposite execution and occurs inside (relational) assertions whereas in Kleymann's case the quantification concerns a single execution and occurs in the semantic interpretation of judgements. Nevertheless, we will make use of this observation in the following section, where we model auxiliary state (of arbitrary type) by decomposition with respect to a silent instruction.

3. Parameterized simulation and noninterference

Like unary properties, instances of relational simulations often involve pre- and postrelations that are parameterized by some notion of auxiliary "state". This section reduces parametric simulations to the decomposed nonparameterized style and then instantiates the development to noninterference for the language of objects.

3.1 Constructions on transition systems, parameterization

Parameterized relational security concerns the following property.

DEFINITION 2. *For transition systems $\mathcal{T}$ and $\mathcal{T}'$ as before, type $\mathcal{Z}$ of auxiliary "states", and parameterized relations $R : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$, we write $\models^{(\mathcal{T},\mathcal{T}')}_{\mathtt{Par}}\ P \sim P' : R \Rightarrow S$ if for all z, s, s', t, and t' with $(s,P,t) \in \mathcal{T}$ and $(s',P',t') : \mathcal{T}'$, sR_zs' implies tS_zt', where R_z denotes the application of R to parameter z.*

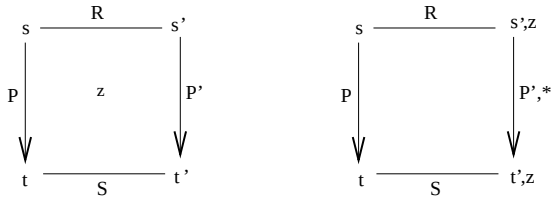
Exploiting the above connection between simulation diagrams and auxiliary state, we reduce parameterized simulation to the nonparameterized form, using two constructions on transition systems. The product transition system $\mathcal{T} \times \mathcal{T}'$ is given by $\{((s,s'),(P,P'),(t,t')).\ (s,P,t) \in \mathcal{T} \wedge (s',P',t') \in \mathcal{T}'\}$ and thus internalizes the two-execution nature of simulations. The identity transition system for parameters $\mathcal{Z}$ is denoted by $\mathcal{I}_{\mathcal{Z}}$ and given by $\{(z,*,y).\ z = y\}$ where $*$ is the unique value of some singleton set of labels. These constructions yield further laws regarding the extremal witnesses – see Figure 5.

For parameterized relation $R : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$ we write $\vec{R}$ for the relation $\{(s,(s',z)).\ (s,s') \in R\,z\}$. The following currying lemma justifies the above constructions on transition systems by

$$
\begin{array}{rcl}
\phi_{\mathsf{L}}^{\mathcal{T}\times\mathcal{T}'}\ (P,P')\ R & = & \{((t,t'),t'').\ (t,(t',t''))\in\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ \{(s,(r',r'')).\ (r',r'')\in\phi_{\mathsf{L}}^{\mathcal{T}'}\ P'\ \{(s',s'').\ ((s,s'),s'')\in R\}\}\}\\
\phi_{\mathsf{R}}^{\mathcal{T}\times\mathcal{T}'}\ (P,P')\ S & = & \{(s'',(s,s')).\ ((s'',s),s')\in\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ \{((t'',t),t').\ (t'',t)\in\phi_{\mathsf{R}}^{\mathcal{T}}\ P\ \{(r'',r).\ (r'',(r,t'))\in S\}\}\}\\
\phi_{\mathsf{L}}^{\mathcal{T}\times\mathcal{I}_{\mathcal{Z}}}\ (P,*)\ R & = & \{((t,z),t').\ (t,(z,t'))\in\phi_{\mathsf{L}}^{\mathcal{T}}\ P\ \{(s,(y,r')).\ ((s,y),r')\in R\}\}\\
\phi_{\mathsf{R}}^{\mathcal{T}'\times\mathcal{I}_{\mathcal{Z}}}\ (P',*)\ S & = & \{(s,(s',z)).\ (s,s')\in\phi_{\mathsf{R}}^{\mathcal{T}'}\ P'\ \{(t,t').\ (t,(t',z))\in S\}\}\\
\models^{\mathcal{T}}\ P : Dec_{\mathsf{L}}\ R\,\phi & = & \models^{\mathcal{T}\times\mathcal{I}_{\mathcal{Z}}}\ (P,*) : \{((s,s'),(t,t')).\ s'=t'\wedge(sRs'\to t\phi t')\}\\
\models^{\mathcal{T}\times\mathcal{I}_{\mathcal{Z}}}\ (P,*) : Dec_{\mathsf{R}}\ R\,\phi & = & \models^{\mathcal{T}}\ P : Dec_{\mathsf{R}}\ \{((s,z),s').\ sR(s',z)\}\ \{((s,z),s').\ s\phi(s',z)\}
\end{array}
$$

Figure 5. Product decomposition laws

relating $\mathcal{Z}$-parameterized behaviour over $\mathcal{T}\times\mathcal{T}'$ to nonparameterized behaviour over $\mathcal{T}\times(\mathcal{T}'\times\mathcal{I}_{\mathcal{Z}})$.



LEMMA 5. *For $R,S:\mathcal{Z}\Rightarrow(\mathcal{S}\times\mathcal{S}')$ we have $\models_{\mathsf{Par}}^{(\mathcal{T},\mathcal{T}')}\ P\sim P' : R\Rightarrow S$ exactly iff $\models_{\mathcal{T}}^{(\mathcal{T}'\times\mathcal{I}_{\mathcal{Z}})}\ P\sim(P',*):\vec{R}\Rightarrow\vec{S}$.*

The product $(P',*)$ is again in the form of Definition 1, resulting in a form of "internal notion of transitivity". For general triples of transitions, we remark that the weakest liberal prerelation decomposes in the two following ways

$$
\begin{array}{l}
WLR_{P,(P',P'')}^{\mathcal{T},(\mathcal{T}'\times\mathcal{T}'')}(S)\\
\quad=\{(s,x).\ x\in WLR_{P',P''}^{\mathcal{T}',\mathcal{T}''}(\{y.\ s\in WLP_{P}^{\mathcal{T}}(\{t.\ tSy\})\})\}\\
\quad=\{(s,x).\ s\in WLP_{P}^{\mathcal{T}}(\{t.\ x\in WLR_{P',P''}^{\mathcal{T}',\mathcal{T}''}(\{y.\ tSy\})\})\}.
\end{array}
$$

Comparing the latter equation with Lemma 4(3) highlights the correspondence between program pairs as considered in Definition 1 and formal products of transition systems.

3.2 Application: auxiliary state for language with objects

Specializing Lemma 5 to $\mathcal{T}'=\mathcal{T}_{\mathsf{Obj}}$ yields identities such as

$$
\begin{array}{l}
\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}\times\mathcal{I}_{\mathcal{Z}}}\ (x':=e',*)\ S\\
\quad=\{(\tau,((s',h'),z)).\ \tau S((s'[x'\mapsto[\![e']\!]_{s'}],h'),z)\}\\
\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}\times\mathcal{I}_{\mathcal{Z}}}\ (C;D,*)\ S=\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}\times\mathcal{I}_{\mathcal{Z}}}\ (C,*)\ (\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}\times\mathcal{I}_{\mathcal{Z}}}\ (D,*)\ S)
\end{array}
$$

and similarly for the other instruction forms. Together with the (unmodified) laws for $\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ R$, these may be used to derive inclusions $\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ \vec{R}\subseteq\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}\times\mathcal{I}_{\mathcal{Z}}}\ (C',*)\ \vec{S}$, i.e. properties in the shape of Lemma 4(1). Proceeding in parallel with the previous section, we may also derive a logic $\vdash_{\mathsf{Par}}C\sim C':R\Rightarrow S$ using the interpretation

$$\exists\,\phi.\ \triangleright C:Dec_{\mathsf{L}}\ \vec{R}\ \phi\wedge\triangleright C':Dec_{\mathsf{R}}\ (\vec{S})^{\sharp}\ \phi^{\sharp}$$

where the operation $\psi^{\sharp}\equiv\{((x,z),x').(x,(x',z))\in\psi\}$ shifts the auxiliary value z to the left component, so that it is not affected by the execution of C'.

By construction, we have that $\vdash_{\mathsf{Par}}C\sim C':R\Rightarrow S$ implies $\models_{\mathsf{Par}}^{(\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}})}\ C\sim C':R\Rightarrow S$. Moreover, the interpretation admits the derivation of similar proof rules as in the previous section, using essentially the same constructions of witnesses. Two exemplary rules are shown in Figure 6, highlighting the formal similarity to the earlier rules. In both cases, the given prerelations R indeed coincide with $WLR_{C,(C',*)}^{\mathcal{T}_{\mathsf{Obj}},(\mathcal{T}_{\mathsf{Obj}}\times\mathcal{I}_{\mathcal{Z}})}(\vec{S})$.

3.3 Application: noninterference for simple objects

Following Banerjee-Naumann [5], our model of noninterference employs indistinguishability relations between states that are parameterized by partial bijections over locations, i.e. sets $\beta\subseteq\mathcal{L}\times\mathcal{L}$ satisfying $(\ell=\ell_1)\Leftrightarrow(\ell'=\ell_1')$ for any $(\ell,\ell')\in\beta$ and $(\ell_1,\ell_1')\in\beta$. These bijections serve two purposes: first, they allow both executions to allocate or otherwise use private objects, i.e. objects that have no correspondence in the opposite execution. Second, for those objects that do have a correspondent in the opposite execution, they hide the precise locations holding either object. Thus, locations behave anonymously.

The formal setup is based on indistinguishability relations for values, stores, and heaps, given fixed associations Γ and Ξ that associate observation levels $l\in\{high,low\}$ to variables and field names, respectively. The two levels correspond to the observation capability of private and public agents, respectively. For values, the indistinguishability relation $v\sim_{\beta}v'$ is defined by

$$\frac{i=i'}{i\sim_{\beta}i'}\qquad\frac{}{\mathsf{Null}\sim_{\beta}\mathsf{Null}}\qquad\frac{(\ell,\ell')\in\beta}{\ell\sim_{\beta}\ell'}$$

For stores and objects, we define

$$
\begin{array}{rcl}
s\sim_{\beta}s' & \equiv & \forall\,x.\,\Gamma(x)=low\to(s\,x)\sim_{\beta}(s'\,x)\\
(c,F)\sim_{\beta}(c',F') & \equiv & c=c'\wedge LFlds_{\Xi}(F)=LFlds_{\Xi}(F')\\
 & \wedge & \forall\,f.\,\Xi(f)=low\to(F\,f)\sim_{\beta}(F'\,f)
\end{array}
$$

where $LFlds_{\Xi}(F)=\{f.\ \exists\,v.\ F\,f=v\ \wedge\ \Xi(f)=low\}$ denotes the public nonempty fields of an object. Finally, indistinguishability of heaps and states is defined as

$$
\begin{array}{rcl}
h\sim_{\beta}h' & \equiv & dom\,\beta\subseteq dom\,h\wedge rng\,\beta\subseteq dom\,h'\\
 & \wedge & \forall\,(\ell,\ell')\in\beta.\ h\,\ell\sim_{\beta}h'\,\ell'\\
(s,h)\sim_{\beta}(s',h') & \equiv & s\sim_{\beta}s'\ \wedge\ h\sim_{\beta}h'
\end{array}
$$

Noninterference then requires that β-indistinguishable initial states yield γ-indistinguishable final states, for some $\gamma\supseteq\beta$:

DEFINITION 3. *C is* noninterferent *if for any $\sigma\sim_{\beta}\sigma'$ with $\sigma\xrightarrow{C}\tau$ and $\sigma'\xrightarrow{C}\tau'$, there exists some $\gamma\supseteq\beta$ with $\tau\sim_{\gamma}\tau'$.*

Instead of understanding this condition as requiring the terminal states to be in some postrelation that is indexed by γ, we employ the reading that they are again in a β-indexed relation that internally quantifies over γ. This allows the partial bijection β to be treated as the auxiliary parameter: for

$$
\begin{array}{rcl}
R_{\mathsf{NI}} & = & \lambda\,\beta.\,\{(\sigma,\sigma').\ \sigma\sim_{\beta}\sigma'\}\\
S_{\mathsf{NI}} & = & \lambda\,\beta.\,\{(\sigma,\sigma').\ \exists\,\gamma\supseteq\beta.\ \sigma\sim_{\gamma}\sigma'\}
\end{array}
$$

$$\frac{R = \lambda\, z.\, \{((s,h),(s',h')).\, (s[x \mapsto [\![e]\!]_s], h)(S_z)(s'[x' \mapsto [\![e']\!]_{s'}], h')\}}{\vdash_{\mathsf{Par}} x{:=}e \sim x'{:=}e' : R \Rightarrow S}$$

$$\frac{R = \lambda\, z.\, \left\{ \begin{array}{l} ((s,h),(s',h')).\, \forall\, \ell\, c\, F\, v\, \ell'\, c'\, F'\, v'.\, s\, y = \ell \to h\, \ell = (c,F) \to F\, f = v \to \\ \quad s'\, y' = \ell' \to h'\, \ell' = (c',F') \to F'\, f' = v' \to (s[x \mapsto v], h)(S_z)(s'[x' \mapsto v'], h') \end{array} \right\}}{\vdash_{\mathsf{Par}} x{:=}y.f \sim x'{:=}y'.f' : R \Rightarrow S}$$

Figure 6. Example proof rules for parameterized relational logic $\vdash_{\mathsf{Par}} C \sim C' : R \Rightarrow S$

noninterference coincides with $\models_{\mathsf{Par}}^{(\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}})} C \sim C : R_{\mathsf{NI}} \Rightarrow S_{\mathsf{NI}}$ and, in fact, also with $\models_{\mathsf{Par}}^{(\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}})} C \sim C : S_{\mathsf{NI}} \Rightarrow S_{\mathsf{NI}}$.

This motivates the definition of the judgement forms

$$\begin{array}{rcl} \mathit{LOW}(C) & \equiv & \vdash_{\mathsf{Par}} C \sim C : S_{\mathsf{NI}} \Rightarrow S_{\mathsf{NI}} \\ \mathit{HIGH}(C) & \equiv & \vdash_{\mathsf{Par}} C \sim \mathbf{Skip} : R_{\mathsf{NI}} \Rightarrow R_{\mathsf{NI}} \end{array}$$

which interpret the type judgements $[low] \vdash C$ and $[high] \vdash C$, respectively.

Figure 7 shows the (flow-insensitive) typing rules that have been derived in this fashion. In the rules for assignment and field modification, we write $\models e : low$ if $s \sim_\beta s'$ implies $[\![e]\!]_s \sim_\beta [\![e]\!]_{s'}$. The similar notation for boolean expressions used in the low rules for conditionals and loops additionally requires that the resulting values are identical. These semantic guarantees are implied by the auxiliary derivation systems for expressions that are typically part of type systems for noninterference (see [5, 29, 33]).

3.4 Discussion: ghost variables

JML and similar specification methodologies support ghost variables, i.e. variables that may be assigned language- or specification-level expressions but which influence neither the content of ordinary program variables nor the control flow. Ghost variables (and extensions of this concept such as ghost methods, fields, classes,...) are used to simplify the verification process and also as an instrumentation technique for specifying extra-functional behaviour such as resource consumption. The use of ghost variables is a typical instance of simulation. It may be formulated either as the comparison of programs from different languages over different state spaces (instrumented language over extended states versus erased language), or as the comparison of two executions of the same annotated program, but for pre- and postrelations that ignore the content of ghost variables. The latter view suggests that ghost variables conceptually coincide with high-security variables - a view that is consistent with the admitted flow direction from ordinary into ghost variables and the independence of control flow from ghost variables. Under this view, the erasure of ghost variables finds its counterpart in the intimate relationship between noninterference and program slicing. The former view corresponds arguably more closely to Hofmann and Pavlova's technique for formally eliminating ghost variables, where derivations in a program logic for the ghost-annotated language are transformed into derivations for equivalent specifications in a logic for the erased language [17].

4. Error states and separation logic

Occasionally, verification formalisms are formulated for operational models that include explicit error states. An important aspect of these formalisms then is their capability to demonstrate that error states are in fact unreachable. Well-known instances of this approach are Reynolds' and Yang's presentations of unary [27] and relational [36] separation logics (SL). The operational semantics underlying these works contain explicit error states called abort and fault, respectively. These are entered whenever an attempt is made to dereference a dangling pointer, and their absence from executions of well-specified programs is a core aspect of SL.

In this section, we extend decomposition to models with error states, based on a unary logic that exposes error states in assertions. We then derive variants of unary and relational separation logics. The choice of operational model, and the use of separate pre- and postconditions in SL judgements, were guided by the desire to stay close to the original presentations.

4.1 Decomposition of language with explicit error states

We consider syntactically the same language as before, but employ an operational semantics with states of type $\Sigma\ option$, ranged over by $\mu, \nu, \ldots$. We write $\bot$ for the error state and $\lfloor\sigma\rfloor$ for the injection of $\sigma \in \Sigma$. The operational semantics $\mathcal{T}_{\mathsf{Err}}$ is obtained by

- injecting the rules of $\mathcal{T}_{\mathsf{Obj}}$, lifting initial states, and lifting final states as appropriate;
- adding behaviour to the object-manipulating instructions such that attempts to dereference a dangling pointer yield $\bot$, and likewise attempts to read from a field that is not in an (allocated) object's field map. We also add erroneous behaviour if the object variable contains an integer value or[3] Null. Figure 8 shows the resulting rules for getfield and putfield.
- adding the rule $\frac{}{\bot \xrightarrow{C} \bot}$. Equivalently, one could propagate error states by appropriately modifying the rules for sequential composition and loops.

Traditionally, program logics for languages with error states build the avoidance of errors into the interpretation of judgements: the satisfaction of the precondition by an initial state guarantees that the execution will not end in $\bot$. In contrast, our unary logics exposes error states explicitly in the assertions, by employing assertions of type $\mathcal{A} \equiv \Sigma\ option \Rightarrow \Sigma\ option \Rightarrow \mathbb{T}$. Both the format of judgements $\vdash^{\mathsf{Err}} C : A$, and their partial-correctness interpretation $\models^{\mathsf{Err}} C : A \equiv \forall\, \mu\, \nu.\, (\mu, C, \nu) \in \mathcal{T}_{\mathsf{Err}} \to A\, \mu\, \nu$ are in accordance with the generic setup of Section 1.1. Once the programmer has been exposed to the existence of error states by their occurrence in the operational model, this approach appears more flexible, as the programmer may explicitly reason about the presence or absence of error states. In the relational setting, coincident occurrence of error states can be expressed directly, as can asymmetric variations of this discipline. Preconditions remain redundant in a termination-insensitive setting, and their use in total-correctness logics remains orthogonal to the issue of error states.

The proof rules (see Figure 9) reflect the changes in the operational semantics but are otherwise similar to the earlier rules. They allow us to specify situations where, for example, the branches of a conditional differ in their error behaviour. Soundness and completeness, i.e. the equivalence between $\vdash^{\mathsf{Err}} C : A$ and $\models^{\mathsf{Err}} C : A$, are proven in the same way as Theorem 1.

Some decomposition laws for $\phi_{\mathsf{L}}^{\mathsf{Err}}\ C\ R$ are given in Figure 10. Subsequently, the proof rules for relational logics without (cf. Figures 3 and 4) and with (cf. Figure 6) auxiliary state can be derived

[3] The fact that type errors should arguably be kept separate from dangling-pointer errors is orthogonal to our argument.

$$\frac{}{[l] \vdash \mathbf{Skip}} \quad \frac{\models e : low}{[low] \vdash x{:=}e} \quad \frac{\Gamma(x) = high}{[high] \vdash x{:=}e} \quad \frac{\Gamma(x) = low \quad \forall\,(f,e) \in \iota.\ \models e : low}{[low] \vdash x := \mathbf{new}\ c\ \iota} \quad \frac{\Gamma(x) = high}{[high] \vdash x := \mathbf{new}\ c\ \iota} \quad \frac{\Gamma(y) = low \quad \Xi(f) = low}{[low] \vdash x{:=}y.f}$$

$$\frac{\Gamma(y) = high}{[high] \vdash x{:=}y.f} \quad \frac{\Gamma(x) = low \quad \Xi(f) = low \quad \models e : low}{[low] \vdash x.f{:=}e} \quad \frac{\Xi(f) = high}{[high] \vdash x.f{:=}e} \quad \frac{[low] \vdash C \quad [low] \vdash D \quad \models b : low}{[low] \vdash \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D}$$

$$\frac{[high] \vdash C \quad [high] \vdash D}{[high] \vdash \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D} \quad \frac{[l] \vdash C \quad [l] \vdash D}{[l] \vdash C;D} \quad \frac{[low] \vdash C \quad \models b : low}{[low] \vdash \mathbf{While}\ b\ \mathbf{do}\ C} \quad \frac{[high] \vdash C}{[high] \vdash \mathbf{While}\ b\ \mathbf{do}\ C} \quad \frac{[high] \vdash C}{[low] \vdash C}$$

Figure 7. Derived typing rules for noninterference

$$\frac{s\ y = \ell \quad h\ \ell = (c,F) \quad F\ f = v}{\lfloor (s,h) \rfloor \xrightarrow{x:=y.f} \lfloor (s[x \mapsto v], h) \rfloor} \qquad \frac{(\exists\, i.\ s\ y = i) \vee s\ y = \mathsf{Null} \vee (\exists\, \ell.\ s\ y = \ell \wedge (\ell \notin dom\, h \vee (\exists\, c\ F.\ h\ \ell = (c,F) \wedge f \notin dom\, F)))}{\lfloor (s,h) \rfloor \xrightarrow{x:=y.f} \bot}$$

$$\frac{s\ x = \ell \quad h\ \ell = (c,F)}{\lfloor (s,h) \rfloor \xrightarrow{x.f:=e} \lfloor (s, h[\ell \mapsto (c, F[f \mapsto [\![e]\!]_s])]) \rfloor} \qquad \frac{(\exists\, i.\ s\ x = i) \vee s\ x = \mathsf{Null} \vee (\exists\, \ell.\ s\ x = \ell \wedge \ell \notin dom\, h)}{\lfloor (s,h) \rfloor \xrightarrow{x.f:=e} \bot}$$

Figure 8. Operational semantics with error states (excerpt)

$$\frac{}{\vdash^{\mathsf{Err}} x{:=}e : \lambda\,\mu\,\nu.\ (\exists\, s\ h.\ \mu = \lfloor (s,h) \rfloor \wedge \nu = \lfloor (s[x \mapsto [\![e]\!]_s], h) \rfloor) \vee (\mu = \nu = \bot)} \qquad \frac{\vdash^{\mathsf{Err}} C : A \quad \vdash^{\mathsf{Err}} D : B}{\vdash^{\mathsf{Err}} C;D : \lambda\,\mu\,\nu.\ \exists\,\rho.\ A\,\mu\,\rho \wedge B\,\rho\,\nu}$$

$$\frac{}{\vdash^{\mathsf{Err}} x := \mathbf{new}\ c\ \iota : \lambda\,\mu\,\nu.\ (\exists\, s\ h\ \ell.\ \mu = \lfloor (s,h) \rfloor \wedge \ell \notin locs(s,h) \wedge \nu = (s[x \mapsto \ell], h[\ell \mapsto (c, [\![\iota]\!]_s)])) \vee (\mu = \nu = \bot)}$$

$$A = \lambda\,\mu\,\nu. \left(\begin{array}{l} \exists\, s\ h.\ \mu = \lfloor (s,h) \rfloor \wedge \left(\begin{array}{l} (\exists\, i.\ s\ y = i \wedge \nu = \bot) \vee (s\ y = \mathsf{Null} \wedge \nu = \bot) \vee \\ \left(\exists\, \ell.\ s\ y = \ell \wedge \left(\begin{array}{l} (\ell \notin dom\, h \wedge \nu = \bot) \vee \\ \left(\exists\, c\ F.\ h\ \ell = (c,F) \wedge \left(\begin{array}{l} (\exists\, v.\ F\ f = v \wedge \nu = (s[x \mapsto v], h)) \\ \vee\ (f \notin dom\, F \wedge \nu = \bot) \end{array} \right) \right) \end{array} \right) \end{array} \right) \\ \vee (\mu = \nu = \bot) \end{array} \right)$$

$$\frac{}{\vdash^{\mathsf{Err}} x{:=}y.f : A}$$

$$\frac{\vdash^{\mathsf{Err}} C : A \quad \vdash^{\mathsf{Err}} D : B}{\vdash^{\mathsf{Err}} \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D : \lambda\,\mu\,\nu.\ (\exists\, s\ h.\ \mu = \lfloor (s,h) \rfloor \wedge ((\![b]\!]_s \to A\,\mu\,\nu) \wedge (\neg [\![b]\!]_s \to B\,\mu\,\nu))) \vee (\mu = \nu = \bot)}$$

$$\frac{\begin{array}{c} \vdash^{\mathsf{Err}} C : B \qquad \forall\, s\ h.\ \neg [\![b]\!]_s \to A\ \lfloor (s,h) \rfloor\ \lfloor (s,h) \rfloor \\ \forall\, s\ h.\ [\![b]\!]_s \to ((B\ \lfloor (s,h) \rfloor\ \bot \to A\ \lfloor (s,h) \rfloor\ \bot) \wedge (\forall\, \sigma\ \nu. B\ \lfloor (s,h) \rfloor\ \lfloor \sigma \rfloor \to A\ \lfloor \sigma \rfloor\ \nu \to A\ \lfloor (s,h) \rfloor\ \nu)) \end{array}}{\vdash^{\mathsf{Err}} \mathbf{While}\ b\ \mathbf{do}\ C : \lambda\,\mu\,\nu.\ (\exists\, s\ h.\ \mu = \lfloor (s,h) \rfloor \wedge A\,\mu\,\nu \wedge (\forall\, t\ k.\ \nu = \lfloor (t,k) \rfloor \to \neg [\![b]\!]_t)) \vee (\mu = \nu = \bot)}$$

Figure 9. Unary program logic with error states (excerpt).

$$\begin{array}{rcl} \phi_{\mathsf{L}}^{\mathsf{Err}}\ x{:=}e\ R & = & \{(\nu,\mu').(\exists\, h\ s.\ \nu = \lfloor (s[x \mapsto [\![e]\!]_s], h) \rfloor \wedge \lfloor (s,h) \rfloor R \mu') \vee (\nu = \bot \wedge \bot R \mu')\} \\ \phi_{\mathsf{L}}^{\mathsf{Err}}\ x := \mathbf{new}\ c\ \iota\ R & = & \{(\nu,\mu').(\exists\, s\ h\ \ell.\ \nu = \lfloor (\rfloor s[x \mapsto \ell], h[\ell \mapsto (c, [\![\iota]\!]_s)]) \wedge \ell \notin locs(s,h) \wedge \lfloor (s,h) \rfloor R \mu') \vee (\nu = \bot \wedge \bot R \mu')\} \\ \phi_{\mathsf{L}}^{\mathsf{Err}}\ \mathbf{While}\ b\ \mathbf{do}\ C\ R & = & \{(\nu,\mu').\ (b, C, R, \nu, \mu') \in \Phi_{\mathsf{L}}^{\mathsf{while}}\} \end{array}$$

where $\Phi_{\mathsf{L}}^{\mathsf{while}}$ is inductively defined by (i.e. the smallest set closed under) the three rules

$$\frac{(\bot,\mu') \in \phi_{\mathsf{L}}^{\mathsf{Err}}\ C\ (R \cap \{(\mu,\mu').\ \forall\, s\ h.\ \mu = \lfloor (s,h) \rfloor \to [\![b]\!]_s\})}{(b, C, R, \bot, \mu') \in \Phi_{\mathsf{L}}^{\mathsf{while}}} \qquad \frac{\lfloor (t,k) \rfloor R \mu' \quad \neg [\![b]\!]_t}{(b, C, R, \lfloor (t,k) \rfloor, \mu') \in \Phi_{\mathsf{L}}^{\mathsf{while}}}$$

$$\frac{(b, C, \phi_{\mathsf{L}}^{T_{\mathsf{Err}}}\ C\ (R \cap \{(\mu,\mu').(\forall\, s\ h.\ \mu = \lfloor (s,h) \rfloor \to [\![b]\!]_s)\}), \nu, \mu') \in \Phi_{\mathsf{L}}^{\mathsf{while}}}{(b, C, R, \nu, \mu') \in \Phi_{\mathsf{L}}^{\mathsf{while}}}$$

Figure 10. Laws for $\phi_{\mathsf{L}}^{\mathsf{Err}}\ C\ R$ (excerpt)

$$\frac{R = \lambda\, z.\ \{(\mu,\mu').\ \forall\, \nu\ \nu'.\ \left(\begin{array}{l} (\forall\, s\ h.\ \mu = \lfloor (s,h) \rfloor \to \nu = \lfloor (s[x \mapsto [\![e]\!]_s], h) \rfloor) \wedge (\mu = \bot \to \nu = \bot) \wedge \\ (\forall\, s'\ h'.\ \mu' = \lfloor (s',h') \rfloor \to \nu' = \lfloor (s'[x' \mapsto [\![e']\!]_{s'}], h') \rfloor) \wedge (\mu' = \bot \to \nu' = \bot) \end{array} \right) \to \nu(S_z)\nu'\}}{\vdash_{\mathsf{Par}}^{\mathsf{Err}} x{:=}e \sim x'{:=}e' : R \Rightarrow S}$$

Figure 11. Error-aware parameterized relational logic (Assign-Assign rule)

in the same style as in the previous section. Figure 11 shows an exemplary rule from the latter. The executions may exhibit different behaviour with respect to the error-status of terminal states.

4.2 Application: derivation of unary separation logic

From the logic $\vdash^{\mathsf{Err}}_{\mathsf{Par}} C \sim C' : R \Rightarrow S$ we derive a unary separation logic. In order to stay close to Reynolds' original presentation [27], we employ judgements with separate pre- and postconditions and explicit auxiliary variables. However, locations remain anonymous, and we do not support deallocation.

In addition to serving as a test case for the decomposed relational logics, our encoding emphasizes the relational nature of (even unary) separation logic: similar to the embedding of low proof rules for noninterference, we interpret a separation logic judgement for C as a relational specification $\vdash^{\mathsf{Err}}_{\mathsf{Par}} C \sim C : R \Rightarrow S$ i.e. as an assertion about two executions of C. The relations R and S arise from the separation logic pre- and postconditions by incorporating the effect of future applications of the frame rule, in a style that is reminiscent of Birkedal et al.'s Kripke-resource extension [13] (see also [11],[4]). Thus, the encoding highlights that separation logic, like noninterference, concerns indistinguishability where states are partitioned into public components on which states are equal (the footprint) and nonpublic components (the frames)[4].

We first outline a simplified encoding for the allocation-free language fragment, before integrating allocation in Section 4.2.2.

4.2.1 Unary separation-logic for allocation-free language

SL assertions concern lifted states that have been extended by an additional store holding the values of auxiliary variables (denoted as upper-case letters $X, Y, \ldots$). In order to admit a formulation of separating conjunction at the granularity level of fields, we follow Parkinson [26] and employ *flat* heaps $\kappa \in (\mathcal{L} \times \mathcal{F}) \rightharpoonup (\mathcal{C} \times \mathcal{V})$. The flattening of heap h, denoted $h^\flat$, is given pointwise by

$$h^\flat(\ell, f) = (c, v) \iff \exists\, F.\, h\, \ell = (c, F) \wedge F\, f = v$$

and is injective up to the identification of unallocated objects with objects without fields. Assertions $M, N, \ldots$ are sets of triples (r, s, κ) where $r \in \mathcal{R}$ is the auxiliary store and s is the ordinary store. Thus, assertions in separation logic cannot talk about error states explicitly, in accordance with Reynolds and Parkinson.

The interpretation of a triple mediates between error-free states in assertions and error-aware states in the operational model, and also between flat and nonflat heaps. A separation logic specification is *valid*, notation $\models_{\mathsf{SL}} \{M\}\ C\ \{N\}$, if whenever $(r, s, h^\flat) \in M$ and $\lfloor (s, h) \rfloor \xrightarrow{C} \nu$ then there exist some t and k such that $\nu = \lfloor (t, k) \rfloor$ and $(r, t, k^\flat) \in N$. Thus, any terminating execution starting in a state satisfying M yields a lifted state satisfying N.

Some simple operators of separation logic are given in Figure 12. Contrary to Parkinson we do not model Java-style garbage collection and consequently use classic rather than intuitionistic separation logic. The singleton mapsto assertions for ordinary and auxiliary variables are formulated on a per-field basis.

We say that assertion M is independent from the variables V, notation $\mathsf{indep}_V M$, if for all $x \in V$ and all v, $(r, s, \kappa) \in M$ implies $(r, s[x \mapsto v], \kappa) \in M$.

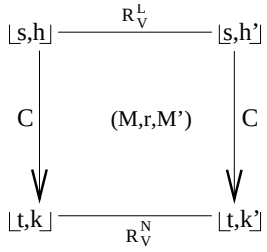
The relational interpretation of assertion N with respect to variable set V yields a relation over error-aware states that is parameterized in the sense of the previous section by triples (M, r, M'). The assertions M and M' are separatingly conjoined with N in the left and right execution, respectively, while r is an auxiliary store. The definition

$$\mathsf{R}^N_V\ (M, r, M') \equiv \left\{ \begin{array}{l} (\lfloor (s,h) \rfloor, \lfloor (s',h') \rfloor).\ s = s' \wedge \\ \quad \mathsf{indep}_V M \wedge (r, s, h^\flat) \in N * M \wedge \\ \quad \mathsf{indep}_V M' \wedge (r, s, h'^\flat) \in N * M' \end{array} \right\}$$

yields pairs of lifted states with identical stores but possibly different heaps, such that the appropriately r-extended states satisfy $N * M$ (left) and $N * M'$ (right), provided that the frames M and M' are independent from V. Frames that depend on V yield the empty relation, and no error state occurs in the relation.

With these definitions in place, we introduce the judgement form $\vdash_{\mathsf{SL}} \{L\}\ C\ \{N\}\ [V]$ as an abbreviation of the self-simulation statement $\vdash^{\mathsf{Err}}_{\mathsf{Par}} C \sim C : \mathsf{R}^L_V \Rightarrow \mathsf{R}^N_V$.

We thus specify that the terminal states of runs starting in R^L_V-related states are related by R^N_V. In particular, both final states are lifted, and the parameterization mechanism propagates the frames from initial to final states.



The following result proves the faithfulness of our embedding with respect to the semantic guarantee and follows from the soundness of the logic $\vdash^{\mathsf{Err}}_{\mathsf{Par}} C \sim C : R \Rightarrow S$ by instantiating both frame parameters by emp.

THEOREM 3. *If* $\vdash_{\mathsf{SL}} \{L\}\ C\ \{N\}\ [V]$ *then* $\models_{\mathsf{SL}} \{L\}\ C\ \{N\}$.

Figure 13 collects the derived proof rules, where program variables possibly modified by C are collected in set V. Due to the Kripke-resource-style relational interpretation of judgements, the proof of the frame rule essentially amounts to transferring the frame L between the public part (the assertions in the concluding judgement) and the private part (the parameter of the hypothetical judgement). The legality of this operation follows from the associativity of $*$, the compatibility of $\mathsf{indep}_V M$ with $*$, and the contravariance of $\mathsf{indep}_V M$ with respect to V.

4.2.2 Allocation

The interpretation R^N_V does not admit the derivation of a proof rule for object allocation: the allocation nondeterminism results in the conjunct $s = s'$ of the invariant not being preserved. Also, the location chosen in one execution may be already in use in the opposite execution, either in the heap region specified by the footprint assertion or in the invisible frame assertion. Motivated by Section 3.3, we outline a remedy using partial bijections. The modified interpretation

$$\mathsf{S}^N_V\ (M, r, r', M') \equiv \left\{ \begin{array}{l} (\lfloor (s,h) \rfloor, \lfloor (s',h') \rfloor).\ \exists\, \beta. \\ \quad s \sim_\beta s' \wedge \mathit{locs}\ r \subseteq \mathit{dom}\, h \wedge \\ \quad r \sim_\beta r' \wedge \mathit{locs}\ r' \subseteq \mathit{dom}\, h' \wedge \\ \quad \mathsf{indep}_V M \wedge (r, s, h^\flat) \in N * M \wedge \\ \quad \mathsf{indep}_V M' \wedge (r', s', h'^\flat) \in N * M' \end{array} \right\}$$

replaces $s = s'$ with $s \sim_\beta s'$ for some partial bijection β, and extends this condition to the (now) two versions of the auxiliary store. The duplication of the auxiliary store is necessitated by the introduction of $s \sim_\beta s'$, since otherwise assertions like $X = x$ which relate auxiliary to ordinary variables could never hold in related states. In addition, auxiliary variables should only mention allocated locations, for either execution. All indistinguishability relations are for the context $\Gamma = \lambda x.\, \mathit{low}$: we designate all (including auxiliary) variables to be relevant. Our interpretation does not require the heaps h and h' to be related by β.

[4] The analogy is not perfect, however: in separation logic, the program operating on the footprint may neither influence nor be influenced by the private component of states, whereas noninterference allows public data to flow into private variables (but no flow in the opposite direction).

$$
\begin{array}{rcl}
M * N & \equiv & \{(r,s,\kappa).\ \exists\, d_1\ d_2.\ \ d_1 \uplus d_2 = dom\,\kappa \wedge (r,s,\kappa{\downarrow_{d_1}}) \in M \wedge (r,s,\kappa{\downarrow_{d_2}}) \in N\ \ \} \\
\mathsf{emp} & \equiv & \{(r,s,\kappa).\ dom\,\kappa = \emptyset\} \\
X.f \mapsto (c,v) & \equiv & \{(r,s,\kappa).\ \exists\, \ell.\ r\ X = \ell \wedge dom\,\kappa = \{(\ell,f)\} \wedge \kappa\,(\ell,f) = (c,v)\} \\
x.f \mapsto (c,v) & \equiv & \{(r,s,\kappa).\ \exists\, \ell.\ s\ x = \ell \wedge dom\,\kappa = \{(\ell,f)\} \wedge \kappa\,(\ell,f) = (c,v)\}
\end{array}
$$

Figure 12. Separation logic operators

$$
\frac{x \in V}{\vdash_{\mathsf{SL}} \{Y = [\![e]\!]_s\}\ x{:=}e\ \{x = Y\}\ [V]} \qquad \frac{y \in V}{\vdash_{\mathsf{SL}} \{X = x \wedge X.f \mapsto (c,Y)\}\ y{:=}x.f\ \{Y = y \wedge X.f \mapsto (c,Y)\}\ [V]}
$$

$$
\frac{}{\vdash_{\mathsf{SL}} \{M\}\ \mathbf{Skip}\ \{M\}\ [V]} \qquad \frac{}{\vdash_{\mathsf{SL}} \{\exists\, v.\ Y = [\![e]\!]_s \wedge x.f \mapsto (c,v)\}\ x.f{:=}e\ \{x.f \mapsto (c,Y)\}\ [V]} \qquad \frac{\vdash_{\mathsf{SL}} \{M \wedge [\![b]\!]_s\}\ C\ \{M\}\ [V]}{\vdash_{\mathsf{SL}} \{M\}\ \mathbf{While}\ b\ \mathbf{do}\ C\ \{M \wedge \neg[\![b]\!]_s\}\ [V]}
$$

$$
\frac{\vdash_{\mathsf{SL}} \{M\}\ C\ \{L\}\ [V] \quad \vdash_{\mathsf{SL}} \{L\}\ D\ \{N\}\ [V]}{\vdash_{\mathsf{SL}} \{M\}\ C;D\ \{N\}\ [V]} \qquad \frac{\vdash_{\mathsf{SL}} \{M \wedge [\![b]\!]_s\}\ C\ \{N\}\ [V] \quad \vdash_{\mathsf{SL}} \{M \wedge \neg[\![b]\!]_s\}\ D\ \{N\}\ [V]}{\vdash_{\mathsf{SL}} \{M\}\ \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D\ \{N\}\ [V]}
$$

$$
\frac{M \subseteq M' \qquad N' \subseteq N \quad \vdash_{\mathsf{SL}} \{M'\}\ C\ \{N'\}\ [V]}{\vdash_{\mathsf{SL}} \{M\}\ C\ \{N\}\ [V]} \qquad \frac{V \subseteq W \qquad \mathsf{indep}_W L \quad \vdash_{\mathsf{SL}} \{M\}\ C\ \{N\}\ [V]}{\vdash_{\mathsf{SL}} \{M * L\}\ C\ \{N * L\}\ [W]}
$$

Figure 13. Derived rules for unary separation logic. All assertions have formally the form $\{(r,s,h).\ \ldots\}$. Upper-case letters denote auxiliary variables and are interpreted in r. Lower-case variables $x, y, \ldots$ are interpreted in s.

The judgement for unary SL with allocation is defined in a similar way as before, by introducing $\vdash_{\mathsf{SLA}} \{L\}\ C\ \{N\}\ [V]$ as an abbreviation for $\vdash^{\mathsf{Err}}_{\mathsf{Par}} C \sim C : \mathsf{S}^L_V \Rightarrow \mathsf{S}^N_V$. In contrast to Section 3.3, the partial bijection in the postrelation is not required to be related to that in the prerelation, hence we do not propagate β via the parameter mechanism. This may be understood as expressing that our SL concerns in-space rather than in-place properties.

The soundness result of this logic guarantees the same property as the allocation-free logic, but requires that auxiliary variables in the precondition concern at most allocated locations.

THEOREM 4. *If* $\vdash_{\mathsf{SLA}} \{L\}\ C\ \{N\}\ [V]$ *and all* $(r,s,h^\flat) \in L$ *satisfy locs* $r \subseteq dom\,h$ *then* $\models_{\mathsf{SL}} \{L\}\ C\ \{N\}$.

The additional condition enforces the reasonable discipline that an auxiliary variable cannot specify objects outside of the footprint, in particular precluding the possibility that an auxiliary variable freezes the initial value of a location that is in fact allocated by C. The condition is, however, an artifact of the use of partial bijections, and will be avoided by a third interpretation below.

The proof rules from Figure 13 largely remain sound, but the rules for conditionals and loops require the condition $\models b : low$. By enforcing that these boolean expressions do not involve any constant locations, we ensure that the two executions follow the same control flow path. In addition, the allocation rule

$$
\frac{x \in V \quad N = *_{\iota(f_i)=e_i} x.f_i \mapsto (c, Y_i)}{\vdash_{\mathsf{SLA}} \{\mathsf{emp} \wedge \textstyle\bigwedge_{\iota(f_i)=e_i} Y_i = [\![e_i]\!]_s\}\ x := \mathbf{new}\ c\ \iota\ \{N\}\ [V]}
$$

can now be derived, where the Y_i are assumed to be distinct.

4.3 A relational separation logic

Turning to the relational case, we employ assertions $P, Q, \ldots$ that are relations over flat states. We say that a quadruple is valid, notation $\models_{\mathsf{RSL}} \{P\}\ C \sim C'\ \{Q\}$, if whenever $((r,s,h^\flat),(r',s',h'^\flat)) \in P$ and $\lfloor (s,h) \rfloor \xrightarrow{C} \nu$ and $\lfloor (s',h') \rfloor \xrightarrow{C'} \nu'$, then $\nu = \lfloor (t,k) \rfloor$ and $\nu' = \lfloor (t',k') \rfloor$ for some t, k, t', and k' with $((r,t,k^\flat),(r',t',k'^\flat)) \in Q$. We thus continue to work in the termination-insensitive setting but require that both executions are error-avoiding.

Figure 14 presents the adaptation of the basic SL primitives, the modified definition of relational independence, and an interpretation of assertions that again yields sets of pairs of non-error states. A parameter contains two (unrelated) auxiliary stores and a relational frame. Our interpretation differs from Yang's in two respects. First, we incorporate (relational) Kripke-resource extension, thus preparing for the integration of procedures and admitting a similarly simple proof of the frame rule as in the unary case. In contrast, Yang's interpretation quantifies universally over pairs of heap extensions. Second, we leave the stores of related states unconstrained rather than interpreting both programs over a joint store (as Yang) or imposing indistinguishability. As the executions of C and C' are operationally unrelated, Yang's condition that requires related phrases to employ distinct variables becomes redundant.

We introduce the judgement $\vdash_{\mathsf{RSL}} \{P\}\, C \sim C'\, \{Q\}\ (V,V')$ as a shorthand for $\vdash^{\mathsf{Err}}_{\mathsf{Par}} C \sim C' : \mathsf{T}^P_{(V,V')} \Rightarrow \mathsf{T}^Q_{(V,V')}$.

THEOREM 5. *If* $\vdash_{\mathsf{RSL}} \{P\}\, C \sim C'\, \{Q\}\ \ (V,V')$ *then* $\models_{\mathsf{RSL}} \{P\}\, C \sim C'\, \{Q\}$.

Figure 15 shows the resulting one-sided rules for object creation, field access and modification, and the frame rule. The expected rules for pairs of e.g. field updates may either be derived separately or may be obtained from the given rules, their symmetric counterparts for the phrases in the right phrase, and the **Skip**-elimination rules – see also [36]. The rules for composition, conditionals, loops, etc. are inherited from the host logic $\vdash^{\mathsf{Err}}_{\mathsf{Par}} C \sim C' : R \Rightarrow S$, modulo the addition of variable sets V and V', and are hence omitted.

Regarding logical rules, Yang includes an embedding rule for pairs of (total-correctness) unary SL judgements. However, a proof attempt for the termination-insensitive variant

$$
\frac{\vdash_{\mathsf{SL}} \{M\}\ C\ \{N\}\ [V] \qquad \vdash_{\mathsf{SL}} \{M'\}\ C'\ \{N'\}\ [V']}{\vdash_{\mathsf{RSL}} \{M \times M'\}\, C \sim C'\, \{N \times N'\}\ (V,V')}
$$

fails, and likewise an attempt for the alternative unary judgement form from Section 4.2.2. Both unary interpretations use Kripke-extension over unary predicates, whereas the interpretation of the concluding judgement requires us to prove a property for *relational* Kripke-extensions. As not every relational frame splits into a pair of unary frames, the hypotheses do not apply.

We are not aware of any discussion of this issue in the literature, but offer the following solution. Motivated by the shape of the rules in Figure 15, we interpret unary SL judgements in terms of the RSL-judgement form and thus formally quantify over relational

$$
\begin{array}{rcl}
\mathsf{emp} & \equiv & \{((r,s,\kappa),(r',s',\kappa')).\ dom\,\kappa = \emptyset \wedge dom\,\kappa' = \emptyset\} \\
P * Q & \equiv & \left\{((r,s,\kappa),(r',s',\kappa')).\ \exists\, d_1\ d_2\ d_1'\ d_2'.\ \begin{array}{l} d_1 \uplus d_2 = dom\,\kappa \wedge d_1' \uplus d_2' = dom\,\kappa' \wedge \\ ((r,s,\kappa\lfloor_{d_1}),(r',s',\kappa'\lfloor_{d_1'})) \in P \wedge ((r,s,\kappa\lfloor_{d_2}),(r',s',\kappa'\lfloor_{d_2'})) \in Q \end{array}\right\} \\
\mathsf{rindep}_V^{V'}\ P & \equiv & \forall\, r\ s\ \kappa\ r'\ s'\ \kappa'.\ ((r,s,\kappa),(r',s',\kappa')) \in P \longrightarrow \left(\begin{array}{l} (\forall\, x\ v.\ x \in V \longrightarrow ((r,s[x \mapsto v],\kappa),(r',s',\kappa')) \in P) \wedge \\ (\forall\, x'\ v'.\ x' \in V' \longrightarrow ((r,s,\kappa),(r',s'[x' \mapsto v'],\kappa')) \in P) \end{array}\right) \\
\mathsf{T}^P_{(V,V')}\ (r,Q,r') & \equiv & \{(\lfloor(s,h)\rfloor,\lfloor(s',h')\rfloor).\ \mathsf{rindep}_V^{V'}\ Q \wedge ((r,s,h^\flat),(r',s',h'^\flat)) \in P * Q\}
\end{array}
$$

Figure 14. Relational separation logic operators and interpretation of relational assertions

$$
\frac{M = (X = x \wedge X.f \Mapsto (c,Y)) \quad y \in V \quad N = (Y = y \wedge X.f \Mapsto (c,Y))}{\vdash_{\mathsf{RSL}} \{M \times A'\}\, y{:=}x.f \sim \mathbf{Skip}\, \{N \times A'\}\ (V,V')}
\qquad
\frac{M = (\exists\, v.\ Y = [\![e]\!]_s \wedge x.f \mapsto (c,v))}{\vdash_{\mathsf{RSL}} \{M \times A'\}\, x.f{:=}e \sim \mathbf{Skip}\, \{x.f \mapsto (c,Y) \times A'\}\ (V,V')}
$$

$$
\frac{\begin{array}{c} M = \mathsf{emp} \wedge \bigwedge_{\iota(f_i)=e_i} Y_i = [\![e_i]\!]_s) \\ N = *_{\iota(f_i)=e_i} x.f_i \mapsto (c,Y_i) \qquad x \in V \end{array}}{\vdash_{\mathsf{RSL}} \{M \times A'\}\, x := \mathbf{new}\ c\ \iota \sim \mathbf{Skip}\, \{N \times A'\}\ (V,V')}
\qquad
\frac{\begin{array}{c} V \subseteq W \qquad \mathsf{rindep}_W^{W'}\ R \qquad V' \subseteq W' \\ \vdash_{\mathsf{RSL}} \{P\}\, C \sim C'\, \{Q\}\ (V,V') \end{array}}{\vdash_{\mathsf{RSL}} \{P * R\}\, C \sim C'\, \{Q * R\}\ (W,W')}
$$

Figure 15. Selected rules of relational separation logic: object-related instructions, FRAME

frames, but use relations that contain emp in the appropriate components of assertions and compare the subject phrase with **Skip** rather than itself. With $\vdash_{\mathsf{USL}} \{M\}\ C\ \{N\}\ [V]$ as a shorthand for [5]

$$
\begin{array}{c} \vdash_{\mathsf{RSL}} \{M \times \mathsf{emp}\}\, C \sim \mathbf{Skip}\, \{N \times \mathsf{emp}\}\ (V,\emptyset) \\ \wedge\ \vdash_{\mathsf{RSL}} \{\mathsf{emp} \times M\}\, \mathbf{Skip} \sim C\, \{\mathsf{emp} \times N\}\ (\emptyset,V) \end{array} \tag{4}
$$

we obtain a proof system that satisfies the following properties. First, the above embedding rule becomes derivable - in particular, $\vdash_{\mathsf{USL}} \{M\}\ C\ \{N\}\ [V]$ implies

$$
\vdash_{\mathsf{RSL}} \{M \times M\}\, C \sim C\, \{N \times N\}\ (V,V)
$$

and thus indeed guarantees a self-simulation property for C. Second, soundness holds in the sense of Theorem 3. Third, all rules in Figure 13 are validated, and so is the allocation rule from Section 4.2.2. As the interpretation does not involve any partial bijections the proof rules do not need the additional side conditions from Section 4.2.2, and soundness formally holds without a restriction on the locations held in auxiliary variables.

5. Discussion

Relational decomposition is a technique for integrating relational logics into unary verification frameworks. We established soundness and completeness of decomposition for general simulations, introduced relational variants of predicate transformers, and studied their relationship to unary transformers. We applied our findings to derive a number of analyses, including separation logics. Our development is backed up by a formalization in Isabelle/HOL.

While all our instantiations concerned a simple imperative language of objects, we pointed out the conceptual applicability of decomposition across language boundaries. We are currently extending our formalization to a bytecode-level variant of the language, aiming to formally relate the two language levels in a subsequent step. Both language levels should also be extended with language features such as arrays, exceptions, and methods. Our previous treatment of decomposition for noninterference [12] supports a restricted notion of (possibly recursive) procedures, but the transfer of this development to virtual methods and non-lock-step-occurrences of method invocations is an open topic.

A long-term goal is the transfer of our techniques to other mainstream languages such as C, and their integration into appropriate verification infrastructures and certified compilation environments. While the integration into production-strength infrastructures for the entire language – as developed in the Concurrent Cminor and CompCert projects – appears a daunting task, language fragments such as Spark/Ada may represent a realistic testbed for relational concepts that is both industrially relevant and formally tractable.

Clearly, even the applications that we touched upon provide ample room for further development, for example concerning extensional notions of declassification [6] and conditional information flow [2], the integration of separation disciplines and noninterference along the lines of Amtoft et al.'s relational logic [1], and a principled treatment of ghost variables and program instrumentation. Magill et al.'s two-step isolation of numeric program abstractions for reasoning about data structures may provide orientation how the latter topics interact with separation aspects [24].

While some of these applications (including program instrumentation) cope well with the condition that loops proceed in lockstep, this aspect is probably the most severe limitation of our development, and also of the logics by Benton and Yang. The necessity to appeal to the rules for injecting pairs of unary judgements (rule Unary, or the embedding rule of relational SL) arguably weakens the appeal of relational logics, and hampers the systematic verification of loop-reordering transformations typically found in modern compilers. A topic for future work is thus to investigate how formal arguments that support such transformations, for example in the setting of translation validation [7], can be rephrased as suitable relational invariants, possibly based on data structures such as Tate et al.'s program equivalence graphs [31] or by embedding aspects of Rhodium's transformation rules [22].

Saabas and Uustalu show how type derivations yield semantics-preserving proof transformations between pairs of judgements of unary Hoare logics [28].

Our interpretations of SL have been rather elementary, despite the integration of Kripke-extensions: concepts such as permissions, separation algebras, and local actions have not been considered [14]. Local actions have an immediate relational reading, which suggests that it might be possible to consider a more fine-

[5] By the relational frame rule, the definition coincides with

$$
\begin{array}{l} \forall A'\ V'.\ \vdash_{\mathsf{RSL}} \{M \times A'\}\, C \sim \mathbf{Skip}\, \{N \times A'\}\ (V,V') \\ \wedge\, \forall A\ V'.\ \vdash_{\mathsf{RSL}} \{A \times M\}\, \mathbf{Skip} \sim C\, \{A \times N\}\ (V',V). \end{array}
$$

grained decomposition of local reasoning, for example by interpreting them as preservation conditions on relations [10].

Separation logic is also treated as a further instance of self-composition by Barthe et al. [8], but in a termination- and error-insensitive manner. Indeed, breaking down error-avoidance of self-composed programs into *equi*-error-avoidance of its constituents would require redoing the soundness analysis of the syntactic manipulations, including the analysis of the Terauchi-Aiken optimizations. On the other hand, the application of self-composition to concurrent systems is at present beyond the capability of our logic-based analysis, and we are unaware of previous attempts to develop relational logics for concurrent languages.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Principles of Programming Languages*, pages 91–102. ACM Press, 2006.

[2] T. Amtoft, J. Hatcliff, E. Rodriguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow. In *15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 229–245. Springer, May 2008.

[3] A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, pages 247–258. IEEE Press, June 2001.

[4] A. W. Appel and S. Blazy. Separation logic for small-step cminor. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference (TPHOLs 2007), Proceedings*, volume 4732 of *LNCS*, pages 5–21. Springer, 2007.

[5] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, Mar. 2005. Special Issue on Language-Based Security.

[6] A. Banerjee, D. A. Naumann, and S. Rosenberg. Towards a logical account of declassification. In M. W. Hicks, editor, *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 61–66. ACM Press, 2007.

[7] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, (CAV 2005), Proceedings*, volume 3576 of *LNCS*, pages 291–295. Springer, 2005.

[8] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM Symposium on Principles of Programming Languages, POPL'04, Venice, Italy*, pages 14–25. ACM Press, 2004.

[10] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In M. Leuschel and A. Podelski, editors, *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2007)*, pages 87–96. ACM Press, 2007.

[11] L. Beringer. Relational bytecode correlations. *Journal of Logic and Algebraic Programming*. To appear.

[12] L. Beringer and M. Hofmann. Secure information flow and program logics. In *IEEE Computer Security Foundations Symposium*, pages 233–248. IEEE Press, 2007.

[13] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 260–269. IEEE Press, 2005.

[14] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), Proceedings*, pages 366–378. IEEE Press, 2007.

[15] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[16] M. V. Hermenegildo and J. Palsberg, editors. *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. ACM Press, 2010.

[17] M. Hofmann and M. Pavlova. Elimination of ghost variables in program logics. In *Trustworthy Global Computing: Revised Selected Papers from the Third Symposium TGC 2007*, number 4912 in LNCS, pages 1–20. Springer, 2008.

[18] S. Hunt and D. Sands. On flow-sensitive security types. In J. G. Morrisett and S. L. P. Jones, editors, *Principles of Programming Languages*, pages 79–90. ACM Press, 2006.

[19] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.

[20] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

[21] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.

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

[23] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[24] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In Hermenegildo and Palsberg [16], pages 211–222.

[25] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

[26] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Press.

[28] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1-2):131–154, 2008.

[29] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, Jan. 2003.

[30] D. A. Schmidt. Structure-preserving binary relations for program abstraction. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, volume 2566 of *LNCS*, pages 245–268. Springer, 2002.

[31] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In Hermenegildo and Palsberg [16], pages 389–402.

[32] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.

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

[34] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[35] `www.cs.princeton.edu/∼eberinge`.

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