Relational decomposition

Lennart Beringer **

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

Abstract. We introduce relational decomposition, a technique for formally reducing termination-insensitive relational program logics to unary logics, that is program logics for one-execution properties. Generalizing the approach of self-composition, we develop a notion of interpolants that decompose along the phrase structure, and relate these interpolants to unary and relational predicate transformers. In contrast to previous formalisms, relational decomposition is applicable across heterogeneous pairs of transition systems. We apply our approach to justify variants of Benton's Relational Hoare Logic (RHL) for a language with objects, and present novel rules for relating loops that fail to proceed in lockstep. We also outline applications to noninterference and separation logic.

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 $s \xrightarrow{P} t$.

However, many program properties are relational: they are naturally phrased as statements over pairs of executions $s \xrightarrow{P} t$ and $s' \xrightarrow{P'} t'$, stipulating that the terminal states are in relation S whenever the initial states are in relation R. Examples include "obviously relational" properties such as program transformations or noninterference [35], but also extensional interpretations of type systems and program analyses [13].

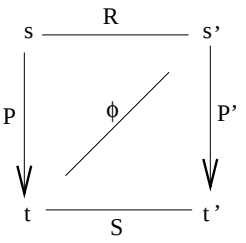


Fig. 1. Relational decomposition of simulation using witness ϕ

In this article, we present *relational decomposition*, a technique for reducing the verification of relational properties to that of unary ones. We demonstrate our technique by deriving a variant of Benton's Relational Hoare Logic (RHL, [13]) from a unary program logic, demonstrating that efforts invested into the construction of semantic models for unary logics can be harnessed for the justification of relational formalisms. We thus open an avenue for integrating relational logics into foundational stacks of verification formalisms [6, 4].

** This work is funded in part by the Air Force Office of Scientific Research (FA9550-09-1-0138) and the National Science Foundation (CNS-0910448).

Relational decomposition reduces the validation of a simulation to the separate verification of unary specifications for the executions $s \xrightarrow{P} t$ and $s' \xrightarrow{P'} t'$. These unary specifications are determined by shared relations ϕ that relate terminal states of (nonprimed) executions to the left with initial states of (primed) executions to the right, in effect witnessing the simulation as indicated in Figure 1. The specification for P then arises from the upper left decomposition triangle (with corners s, s', and t), and the specification for P' arises from the lower right triangle (with corners s', t', and t). Each unary specification universally quantifies over states from the opposite execution.

The present article makes two contributions. First, we present fundamental properties of relational decomposition, in a setting in which the two transition systems involved in a simulation are not necessarily identical. We exploit this flexibility when extending relational decomposition to parametrized simulations, i.e. situations where the relations R and S are parametric over values of some type $\mathcal{Z}$. Second, we present specific relational decompositions of relational program logics, for a concrete language of commands, objects, and loops, thus demonstrating how witness relations may be obtained in a concrete setting. More specifically, we

1. establish soundness of decomposition: any witness ϕ yields unary specifications for the left and right executions that together imply the simulation property (Section 2);
2. establish the formal completeness of decomposition: witnesses exist whenever the simulation property semantically holds. The space of witnesses is characterized by an inclusion property between relational predicate transformers. We present laws that relate these transformers to their unary counterparts (Section 2);
3. derive a termination-insensitive variant of RHL in decomposed style, including novel rules for *dissonant* loops, i.e. loops that do not proceed in lock-step; in particular, proof rules synthesize witnesses in a compositional fashion (Section 3);
4. outline an extension of relational decomposition that deals with parametrized simulations. The resulting logic can be used to justify type systems for noninterference [7] and variants of relational separation logics [40] (Section 4).

All results have been verified using the theorem prover Isabelle/HOL, and the source files are available online [14]. As a consequence, details of most proofs are omitted.

1.1 Related work

Relational decomposition extends the idea of *self-composition* [11]. For the special case of (termination-insensitive) noninterference [35], self-composition establishes the security of a command[1] C by verifying the one-execution property $\{\sim_L\}\ C; C'\{\sim_L\}$ where C' arises from C by replacing all program variables x in C by fresh copies x', and the predicate $\sim_L$ is defined as $\{s \mid \forall x \in L.\ s\ x = s\ x'\}$ for some fixed ("low") subset L of (nonprimed) variables. Self-composition thus reduces relational to unary verification using *syntactic* operations on programs: variable renaming, code duplication, and (sequential) composition. Relational decomposition reveals that the essence

[1] Anticipating the concrete programming language used later in the paper, we let C range over some concrete category of commands, in contrast to the generic labels P.

of self-composition lies neither in the "self" nor in the "composition" aspect, but in the dual use of auxiliary state: related programs do not have to be copies of each other (in fact they need not be syntactically similar at all and may stem from different languages), and no syntactic composition operator is required at their point of interaction. Indeed, the witness relations ϕ can be interpreted as specifications applicable at the point of program composition in a self-composed program, mediating between pre- and postrelations in a style reminiscent of interpolants [17].

Terauchi and Aiken [37] observe that the efficiency of self-composition is improved if phrase-duplication is applied only to small program fragments, but limit their attention largely to noninterference. They demonstrate that type systems for noninterference yield transformation rules that push self-composition towards the leaves of the syntax tree so that the symmetry between C and C' can be better exploited. Our application of relational decomposition is phrased in the opposite direction: we derive a relational logic from a unary one rather than aiming to obtain unary specifications from a given relational specification. The language considered by Terauchi and Aiken [37] is that of simple assignments and while-loops. In particular, heap structures – whose treatment presents a particular challenge due to the fact that differences in location chosen by the allocator in different runs are generally considered unobservable – are not considered.

Naumann [31] extends Terauchi and Aiken's work to a language with objects, for general relational pre- and postconditions. Indistinguishability of locations is treated using the well-known technique of partial bijections [12, 7]. Naumann's encoding of relational into unary specifications employs ghost fields: each object contains a boolean ghost field indicating whether the object should be interpreted w.r.t. the left or the right execution, and a further ghost field that (if nonnull) refers to the object's "mate" in the opposite execution. From a semantic point of view this encoding is slightly unsatisfactory, as the soundness result is contingent on the condition that *None of the considered relations or programs should depend on these fields except through explicit use in the encoding* ([31], page 9). Arguably, this condition represents an external assumption whose impact on the end-to-end guarantee is not formally modeled, requiring the end-user to trust some additional tool validating (possibly a syntactic approximation of) this condition. In fact, independence is itself a relational concept – and so is arguably the concept of ghost variables: the rules governing their use are virtually identical to those for a high-security variable in noninterference. A practical drawback of Naumann's encoding is that the explicit declaration of ghost fields permeates all classes, potentially limiting the scalability of the approach ([31], page 16).

Beringer-Hofmann [15] and Darvas-Hähnle-Sands [18] formulate self-composition in terms of program logics, but again focus on noninterference. In particular, Beringer and Hofmann [15] show how standard type systems [38, 24] can be formally interpreted in a unary logic, using a type-directed rule-by-rule construction of intermediate formulae ϕ. The witness relations employed in the present paper extend this construction to arbitrary relational simulation properties over possibly distinct transition systems. The present paper highlights that the synthesis of the witnesses proceeds along the phrase structure or the structure of the RHL proof rules, independent of the type structure.

The logics of Benton [13] and Yang [40] provide a blueprint for our relational Hoare logic, but do not support verification across different languages. These logics are jus-

tified by direct recourse to operational semantics rather than being derived from an intermediate unary verification formalism. A further difference consists in our use of a termination-insensitive interpretation of relational judgements: simulations are vacuously fulfilled if either execution fails to terminate. This design decision is motivated by the fact that already in the unary setting, proof techniques for termination (appropriate measures, i.e. variants) are significantly different from those for partial-correctness properties (invariants). A second reason is that applications such as compiler verification often actually relax termination-sensitivity to at least an asymmetric form [28]. 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.

In the area of translation validation, a number of verification approaches have been proposed, some of which include rules for relating loops that fail to proceed in lockstep [32, 20, 39]. In contrast to our proof system, these approaches are typically justified with the help of auxiliary constructs such as program labels and paths, in conflict with the extensional view taken in the present paper and also emphasized by Benton.

2 The principle of decomposition

2.1 Introducing interpolating witnesses

For the purpose of this paper, a transition system $\mathcal{T}$ over state space $\mathcal{S}$ and labels $\mathcal{P}$ is a ternary relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S}$. Contrary to other uses of transition systems, we employ a big-step reading where labels may represent compound program phrases whose cumulative effect is captured in a single transition.

Each transition system $\mathcal{T}$ gives rise to a one-execution specification system where assertions are (curried) binary predicates A over $\mathcal{S}$ that relate initial and final states, similar to postconditions in VDM [25]. We interpret specifications as partial-correctness statements, by writing $\models^{\mathcal{T}} P : A$ whenever $(s, P, t) \in \mathcal{T}$ implies $A\ s\ t$ for all $s, t \in \mathcal{S}$.

The formal notion of simulation employs pre- and postconditions that relate states across two transition systems. In order to clearly distinguish between one- and two-execution specifications we write relational assertions in uncurried, often infix style.

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 $R \Longrightarrow S$-similar, notation $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$, if for all s, s', t, and t' with $(s, P, t) \in \mathcal{T}$ and $(s', P', t') \in \mathcal{T}'$, sRs' implies tSt'.*

Unless explicitly remarked otherwise, we follow the convention that nonprimed entities (states, phrases,...) refer to $\mathcal{T}$ and primed entities to $\mathcal{T}'$.

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, intra-language transformations, and termination-insensitive versions of properties considered by [13]. 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 operators Dec_{L} and Dec_{R}

$$Dec_{\mathsf{L}}\ R\ \phi\ s\ t = \forall s'.\ sRs' \rightarrow t\phi s' \text{ and } Dec_{\mathsf{R}}\ S\ \phi\ s'\ t' = \forall t.\ t\phi s' \rightarrow tSt'. \quad (1)$$

Given $\phi \subseteq \mathcal{S} \times \mathcal{S}'$, operator Dec_{L} constructs a unary assertion for the left decomposition triangle from Figure 1, 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 right decomposition triangle from Figure 1, 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* $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$.

Thus, the task of verifying $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$ is reduced to the task of exhibiting an arbitrary ϕ that satisfies the two one-execution properties. Each suitable witness ϕ is a (relational) interpolant between the relational precondition R and the relational postcondition S, by virtue of constraints (1). Before deriving concrete interpolants that justify RHL in Section 3, we discuss further formal properties of decomposition.

2.2 Properties of 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

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

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'. In point-free notation [22], $\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$ can be written as $\widehat{P'} \backslash S$ where $\widehat{P'}$ denotes the uncurried form of the transition relation for P' and the *weakest prespecification* $X \backslash Y$ is defined as $\overline{\overline{Y}; X^{-1}}$.

By construction, these operators are covariant in their second argument and yield valid specifications for their defining triangles:

Lemma 2. *We have* $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$ *and* $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$.

They also satisfy $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ T) \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ T)$ and set-theoretic laws such as $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ (R \cap T) \subseteq \phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \cap \phi^{\mathcal{T}}_{\mathsf{L}}\ P\ T$. In particular, $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R$ is the *least* relation obeying the left triangle, and $\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$ is the *greatest* relation obeying the right triangle:

Lemma 3. *If* $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ \phi$ *then* $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi$. *If* $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ \phi$ *then* $\phi \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$.

Thus, any witness ϕ from Lemma 1 is sandwiched between the two operators, i.e. satisfies $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$. Conversely, either operator is suitable as a witness:

Lemma 4. (Completeness) *Suppose* $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$. *Then*

1. $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$ *and* $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$
2. $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$ *and* $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$.

Combining the above lemmas, we obtain the following.

Theorem 1. $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow S$ *iff* $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$.

Proof. For the implication from left to right, we apply Lemma 4(1) to obtain $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$. Then, Lemma 3 (part 2) yields $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$. For the opposite implication, we have $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R)$ by Lemma 2, so $\models^{\mathcal{T}} P : Dec_{\mathsf{L}}\ R\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$ by covariance. We also have $\models^{\mathcal{T}'} P' : Dec_{\mathsf{R}}\ S\ (\phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S)$ (again by Lemma 2), hence the result follows by applying Lemma 1 to $\phi := \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$.

The operators are defined from the relational perspective, but are also intimately connected with the unary transformers

$$\begin{array}{l} \textit{Strongest postcondition} : SP^{\mathcal{T}}_{P}(X) = \{t \mid \exists\, s \in X.\ (s, P, t) \in \mathcal{T}\} \\ \textit{Weakest lib. precondition} : WLP^{\mathcal{T}'}_{P'}(Y') = \{s' \mid \forall\, t'.\ (s', P', t') \in \mathcal{T}' \rightarrow t' \in Y'\} \end{array}$$

where X and Y' are state sets from $\mathcal{T}$ and $\mathcal{T}'$, respectively. Indeed, we have

$$\begin{array}{r} \phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R = \{(t, s') \mid t \in SP^{\mathcal{T}}_{P}(\{s \mid sRs'\})\} \\ \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S = \{(t, s') \mid s' \in WLP^{\mathcal{T}'}_{P'}(\{t' \mid tSt'\})\}) \end{array} \tag{2}$$

Substituting these equalities into Theorem 1, we have that $R \Longrightarrow S$-similarity is soundly and completely characterized by the inclusion of the left SP in the right WLP:

$$\{(t, s') \mid t \in SP^{\mathcal{T}}_{P}(\{s \mid sRs'\})\} \subseteq \{(t, s') \mid s' \in WLP^{\mathcal{T}'}_{P'}(\{t' \mid tSt'\})\}. \tag{3}$$

We may also define the relational transformers

$$\begin{array}{l} \textit{Strongest postrelation} : \\ \quad SR^{\mathcal{T},\mathcal{T}'}_{P,P'}(R) = \{(t, t') \mid \exists\, s\ s'.\ (s, P, t) \in \mathcal{T} \wedge (s', P', t') \in \mathcal{T}' \wedge sRs'\} \\ \textit{Weakest lib. prerelation} : \\ \quad WLR^{\mathcal{T},\mathcal{T}'}_{P,P'}(S) = \{(s, s') \mid \forall\, t\ t'.\ (s, P, t) \in \mathcal{T} \rightarrow (s', P', t') \in \mathcal{T}' \rightarrow tSt'\}. \end{array}$$

These satisfy the following properties.

Lemma 5. *We have*

1. $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ (WLR^{\mathcal{T},\mathcal{T}'}_{P,P'}(S)) \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ S$
2. $\phi^{\mathcal{T}}_{\mathsf{L}}\ P\ R \subseteq \phi^{\mathcal{T}'}_{\mathsf{R}}\ P'\ (SR^{\mathcal{T},\mathcal{T}'}_{P,P'}(S))$
3. $\begin{array}{rl} WLR^{\mathcal{T},\mathcal{T}'}_{P,P'}(S) &= \{(s, s') \mid s' \in WLP^{\mathcal{T}'}_{P'}(\{t' \mid s \in WLP^{\mathcal{T}}_{P}(\{t \mid tSt'\})\})\} \\ &= \{(s, s') \mid s \in WLP^{\mathcal{T}}_{P}(\{t \mid s' \in WLP^{\mathcal{T}'}_{P'}(\{t' \mid tSt'\})\})\} \end{array}$
4. $\begin{array}{rl} SR^{\mathcal{T},\mathcal{T}'}_{P,P'}(R) &= \{(t, t') \mid t' \in SP^{\mathcal{T}'}_{P'}(\{s' \mid t \in SP^{\mathcal{T}}_{P}(\{s \mid sRs'\})\})\} \\ &= \{(t, t') \mid t \in SP^{\mathcal{T}}_{P}(\{s \mid t' \in SP^{\mathcal{T}'}_{P'}(\{s' \mid sRs'\})\})\}. \end{array}$

The latter two equations show that $R \Longrightarrow S$-similarity can also be verified by sequentially applying the respective unary liberal precondition operators (item 3) and verifying $R \subseteq WLR^{\mathcal{T},\mathcal{T}'}_{P,P'}(S)$, or by applying the respective unary strongest postcondition operators (item 4) and verifying $SR^{\mathcal{T},\mathcal{T}'}_{P,P'}(R) \subseteq S$. The mixed positive and negative occurrences of interpolants in the definition of Dec_{L} and Dec_{R} highlight that interpolants capture the property applicable at the "point of composition" in self-composition, i.e. at the state where P has executed but P' has not started yet, as captured by equation (3).

3 Application: decomposed justification of Relational Hoare Logic

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

3.1 Language definition and unary program logic

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 of commands is

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

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 objects, heaps, stores, and states

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

[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 as a big-step relation, with nondeterministic allocation

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

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

($locs\ \sigma$ denotes the set of all locations ℓ occurring in σ) and field modification rule

$$\textsc{OpPut}\ \frac{s\,x = \ell \quad h\,\ell = (c, F)}{(s,h) \xrightarrow{x.f:=e} (s, h[\ell \mapsto (c, F[f \mapsto [\![e]\!]_s])])}.$$

The semantics does not model error states or stuck executions explicitly: attempts to access dangling pointers, Null, or undefined fields of allocated objects result in the absence of a formal derivation.

In accordance with the setup of Section 2.1, we have derived a unary logic with judgements of the form $\rhd C : A$ where A are curried relations over Σ. The proof rules are essentially those given in [15], plus rules for object allocation

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

and for the field accessing instructions (omitted). Using standard techniques [26, 33], we have proven the logic sound and complete, relative to the ambient logic HOL:

Theorem 2. *$\rhd C : A$ holds if and only if $\models^{\mathcal{T}_{\mathsf{Obj}}} C : A$.*

3.2 Derivation of relational proof rules

Instantiating $\mathcal{T} = \mathcal{T}_{\mathsf{Obj}}$ and/or $\mathcal{T}' = \mathcal{T}_{\mathsf{Obj}}$ yields laws that decompose the operators $\phi_{\mathsf{L}}^{\mathcal{T}_{\mathsf{Obj}}}\ C\ R$ and $\phi_{\mathsf{R}}^{\mathcal{T}_{\mathsf{Obj}}}\ C'\ S$ along the phrase structure, in accordance with the characterizing equations (2). Examples for such laws are

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

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 proof rules for judgements $C \sim C' : R \Longrightarrow S$. In contrast to Benton [13], but in accordance with Definition 1, we interpret these in the termination-insensitive style. By virtue of the previous section, several formal interpretations of these judgements are compatible with this reading. The derivability from the unary program logic is most explicit if we define $C \sim C' : R \Longrightarrow S$ to be a shorthand for

$$\exists\phi.\ \rhd\ C : Dec_{\mathsf{L}}\ R\ \phi \wedge \rhd C' : Dec_{\mathsf{R}}\ S\ \phi \tag{4}$$

and then establish the proof rules as derived lemmas. Figure 2 shows selected proof rules for pairs of structurally identical phrases, namely the rule for related object allocations (representative of all rules for relating pairs of atomic instructions) and rules for compound phrases. These rules are similar to the rules given (for the heap-free fragment of the language) by Benton [13]. As is the case in loc. cit., the loop rule is restricted to situations where both iterations proceed in lock-step. Our rule for conditionals allows the executions to proceed along different control paths and consequently has hypotheses for all four possible combinations of branch outcomes.

$$\textsc{RhlNew}\ \frac{R = WLR^{\mathcal{T}_{\mathsf{Obj}},\mathcal{T}_{\mathsf{Obj}}}_{x:=\mathbf{new}\ c\ \iota,x':=\mathbf{new}\ c'\ \iota'}(S)}{x := \mathbf{new}\ c\ \iota \sim x' := \mathbf{new}\ c'\ \iota' : R \Longrightarrow S}$$

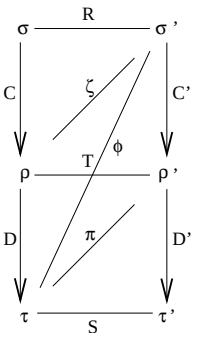
$$\textsc{RhlComp}\ \frac{C \sim C' : R \Longrightarrow T \qquad D \sim D' : T \Longrightarrow S}{C; D \sim C'; D' : R \Longrightarrow S}$$

$$\textsc{RhlIff}\ \frac{\begin{array}{c} C \sim C' : R \cap \{((s,h),(s',h')) \mid [\![b]\!]_s \wedge [\![b']\!]_{s'}\} \Longrightarrow S \\ D \sim D' : R \cap \{((s,h),(s',h')) \mid \neg[\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \Longrightarrow S \\ C \sim D' : R \cap \{((s,h),(s',h')) \mid [\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \Longrightarrow S \\ D \sim C' : R \cap \{((s,h),(s',h')) \mid \neg[\![b]\!]_s \wedge [\![b']\!]_{s'}\} \Longrightarrow S \end{array}}{\mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \sim \mathbf{If}\ b'\ \mathbf{then}\ C'\ \mathbf{else}\ D' : R \Longrightarrow S}$$

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

Fig. 2. RHL rules for identically shaped phrases (excerpt)

The derivation of the rules exhibits witnesses as mandated by equation (4). By the results of the previous section, witnesses for the atomic instruction forms may be chosen as $\phi^{\mathcal{T}_{\mathsf{Obj}}}_{\mathsf{L}}\ C\ R$ or $\phi^{\mathcal{T}_{\mathsf{Obj}}}_{\mathsf{R}}\ C'\ S$, or any relation sandwiched between the two. Witnesses for compound phrases are synthesized from the witnesses of the constituents, generalizing the noninterference-specific construction from [15]. For example, the witness for the conclusion of rule RHLCOMP is given by $\phi = \{(\tau,\sigma').\ \exists\,\rho.\ \rho\zeta\sigma' \wedge (\forall\,\rho'.\ \rho T\rho' \rightarrow \tau\pi\rho')\}$ where ζ and π denote the witnesses of the hypotheses, as illustrated on the right. The witness for while rule, $\Phi^{\mathtt{While}}_{(b',R,\phi)}$, is constructed as the least fixed point of the functional



$$\psi \mapsto \left\{ (\tau,(t',k')) \;\middle|\; \begin{array}{l} ([\![b']\!]_{t'} \rightarrow (\exists\,\sigma.\ \sigma\phi(t',k') \wedge (\forall\,\sigma'.\ \sigma R\sigma' \rightarrow \tau\psi\sigma'))) \\ \wedge\ (\neg[\![b']\!]_{t'} \rightarrow \tau R(t',k')) \end{array} \right\}$$

(which is monotone in ϕ and ψ), where ϕ is the witness of $C \sim C' : U \Longrightarrow R$. As a by-product of our generalization, the proofs for the compound phrases reveal a discipline that is not apparent in our earlier noninterference-specific formulation [15]: proofs of the $Dec_{\mathsf{L}}\ .\ .$-conjuncts only use $Dec_{\mathsf{L}}\ .\ .$-clauses of the hypotheses, and proofs of the $Dec_{\mathsf{R}}\ .\ .$-conjuncts only use $Dec_{\mathsf{R}}\ .\ .$-clauses. Thus, the proof system separates into subsystems with specifications $Dec_{\mathsf{L}}\ .\ .$ and $Dec_{\mathsf{R}}\ .\ .$.

In addition to the rules in Figure 2, 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 3. 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. On the other hand, Benton's rule of transitivity $\dfrac{C \sim C' : R \Longrightarrow S \quad C' \sim C'' : R \Longrightarrow S \qquad PER(R \Rightarrow S)}{C \sim C'' : R \Longrightarrow S}$ where $PER(R \Rightarrow S)$ indicates that the function space $R \Rightarrow S$ is a partial equivalence relation[3], is unsound in the termination-insensitive setting: 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 $\dfrac{C \sim C' : R \Longrightarrow S \qquad PER(R \Rightarrow S)}{C' \sim C : R^{-1} \Longrightarrow S^{-1}}$ can be derived if we exploit the semantic symmetry of the simulation relation and use the formal completeness of the program logic, i.e. the reverse direction of Theorem 2. An alternative is to modify the interpretation of judgements, by conjoining (4) with

$$\exists \psi.\ \triangleright\ C' : Dec_{\mathsf{L}}\ R^{-1}\ \psi \wedge \triangleright\ C : Dec_{\mathsf{R}}\ S^{-1}\ \psi. \tag{5}$$

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

$$\textsc{ComBrL}\ \frac{\begin{array}{c} C \sim C' : U \Longrightarrow S \qquad U = R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \\ D \sim C' : T \Longrightarrow S \qquad T = R \cap \{((s,,h),(s',h')).\ \neg[\![b]\!]_s\} \end{array}}{\mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \sim C' : R \Longrightarrow S}$$

$$\textsc{DeadL}\ \frac{\triangleright\ C : Dec_{\mathsf{L}}\ R\ S}{C \sim \mathbf{Skip} : R \Longrightarrow S} \qquad \textsc{CompSkip}\ \frac{C \sim \mathbf{Skip} : R \Longrightarrow T \quad D \sim D' : T \Longrightarrow S}{C;D \sim D' : R \Longrightarrow S}$$

$$\textsc{False}\ \frac{}{C \sim C' : \emptyset \Longrightarrow S} \qquad \textsc{Unary}\ \frac{\begin{array}{c} \triangleright\ C : A \qquad \triangleright\ C' : A' \\ R = \{(\sigma,\sigma').\ \forall\ \tau\ \tau'.\ A\ \sigma\ \tau \rightarrow A'\ \sigma'\ \tau' \rightarrow \tau S \tau'\} \end{array}}{C \sim C' : R \Longrightarrow S}$$

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

Fig. 3. Nonsynchronous RHL rules (excerpt)

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. For example, the proof of the **Skip**-elimination rule $\dfrac{\mathbf{Skip};C \sim C' : R \Longrightarrow S}{C \sim C' : R \Longrightarrow S}$ employs completeness to deduce $\triangleright\ C : Dec_{\mathsf{L}}\ R\ \phi$ from $\triangleright\ \mathbf{Skip};C : Dec_{\mathsf{L}}\ R\ \phi$. An alternative to the use of the formal completeness result would be to work directly at the level of semantic validity, i.e. replace all judgements of the form $\triangleright\ C : A$ in (4) or (5) by $\models^{\mathcal{T}_{\mathsf{Obj}}} C : A$.

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

[3] In Benton's setting $R \Rightarrow S$ and $R \Longrightarrow S$ coincide.

3.3 New rules for dissonant loops

Like the rules of Benton [13] and Yang [40], rule RHLWHL from Figure 2 requires the iterations to proceed in lock-step. We have derived two novel rules that overcome this limitation. Our first rule requires both bodies to preserve the invariant individually, decoupling the loops based on a similar motivation as the dead code rules:

$$\frac{\begin{array}{c} C \sim \mathbf{Skip} : (R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\}) \Longrightarrow R \\ \mathbf{Skip} \sim C' : (R \cap \{((s,h),(s',h')).\ [\![b']\!]_{s'}\}) \Longrightarrow R \\ S = R \cap \{((s,h),(s',h')).\ \neg[\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \end{array}}{\mathbf{While}\ b\ \mathbf{do}\ C \sim \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Longrightarrow S}$$

The second rule splits the invariant into preconditions appropriate for synchronized iterations and autonomous iterations.

$$\frac{\begin{array}{ll} C \sim C' : U \Longrightarrow R & U \subseteq R \cap \{((s,h),(s',h')).\ [\![b]\!]_s \wedge [\![b']\!]_{s'}\} \\ C \sim \mathbf{Skip} : V \Longrightarrow R & V \subseteq R \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \\ \mathbf{Skip} \sim C' : W \Longrightarrow R & W \subseteq R \cap \{((s,h),(s',h')).\ [\![b']\!]_{s'}\} \\ R \subseteq U \cup V \cup W \cup S & S = R \cap \{((s,h),(s',h')).\ \neg[\![b]\!]_s \wedge \neg[\![b']\!]_{s'}\} \\ W \cap \{((s,h),(s',h')).\ [\![b]\!]_s\} \subseteq U & V \cap \{((s,h),(s',h')).\ [\![b']\!]_{s'}\} \subseteq U \end{array}}{\mathbf{While}\ b\ \mathbf{do}\ C \sim \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Longrightarrow S}$$

This rule is interderivable with the variant where the last two side-conditions (the inclusions $\ldots \subseteq U$) are omitted, for the price of replacing $U \subseteq \ldots$ by $U = \ldots$ in the fist side condition. The earlier rule RHLWHL arises from this variant by setting $V = W = \emptyset$.

The decomposed derivation of the new loop rules employs fixed-point-interpolants similar to $\Phi^{\mathtt{While}}_{(b',R,\phi)}$ above. For the details, see [14].

As an example for the application of these rules, consider the programs

$$C \equiv r{:=}0; i{:=}0; \mathbf{While}\ i < n\ \mathbf{do}\ (r{:=}r+i; i{:=}i+1)$$
$$C' \equiv r{:=}0; i{:=}0; \mathbf{While}\ i < n\ \mathbf{do}\ (r{:=}r+i; i{:=}i+1; r{:=}r+i; i{:=}i+1).$$

The equivalence between the C and its unrolling C' for even n may be formulated as the relational specification $C \sim C' : T_N \Longrightarrow S_N$ for any $N \geq 0$ and

$$T_N \equiv \{((s,h),(s',h')).\ [\![n]\!]_s = [\![n]\!]_{s'} = 2N\}$$
$$S_N \equiv \{((s,h),(s',h')).\ [\![n]\!]_s = [\![n]\!]_{s'} = [\![i]\!]_s = [\![i]\!]_{s'} = 2N \wedge [\![r]\!]_s = [\![r]\!]_{s'} = 2N^2 - N\}.$$

A proof for this specification using the rule for independent loops instantiates R to

$$T_N \cap \left\{ ((s,h),(s',h')) \;\middle|\; \begin{array}{r} \exists\ I\ I'\ k.\ [\![i]\!]_s = I \wedge [\![i]\!]_{s'} = I' \wedge 0 \leq I \leq 2N \wedge 0 \leq I' \leq 2N \\ \wedge\ 2[\![r]\!]_s = I(I-1) \wedge 2[\![r]\!]_{s'} = I'(I'-1) \wedge I' = 2k \end{array} \right\}$$

where each conjunct applies to either the primed or the non-primed state.

Alternatively, the same specification may be proven using the rule for partially synchronized loops, using the instantiation $W = \emptyset$,

$$R \equiv T_N \cap \left\{ ((s,h),(s',h')) \;\middle|\; \begin{array}{l} \exists\, I\, I'.\ [\![i]\!]_s = I \wedge [\![i]\!]_{s'} = I' \wedge 0 \leq I, I' \leq 2N \\ \quad \wedge\ 2[\![r]\!]_s = I(I-1) \wedge 2[\![r]\!]_{s'} = I'(I'-1) \\ \quad \wedge\ ((I < N \wedge I' = 2I) \vee (N \leq I \wedge I' = 2N)) \end{array} \right\}$$
$$U \equiv R \cap \{((s,h),(s',h')).\ [\![i]\!]_s < 2N \wedge [\![i]\!]_{s'} < 2N\}$$
$$V \equiv R \cap \{((s,h),(s',h')).\ N \leq [\![i]\!]_s < 2N\},$$

based on the intuition that the first N iterations proceed synchronously, followed by N additional unilateral iterations of the left loop. The entanglement surfaces in the disjunctive final clause in the definition of R.

The above specifications universally quantify over the meta-variable N at Isabelle-level. Using the parametrization mechanism below, we have also performed verifications where N is part of the specification, and shared between pre- and postconditions.

4 Extensions and applications

We briefly sketch some extensions of our formal framework, and motivating applications. Details of the development are available in [14].

Parametrized simulations Often, simulations are of interest where the pre- and post-relations employ auxiliary state. We model this situation by endowing the relations with additional arguments, similar to Kleymann's [26] treatment for unary logics.

Definition 2. *For transition systems $\mathcal{T}$ and $\mathcal{T}'$ as before, type $\mathcal{Z}$ of auxiliary states, and parametrized relations $R : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$, we write $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow_{\mathcal{Z}} S$ 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.*

Parametrized simulation can be reduced to nonparametrized simulation using two constructions on transition systems, as follows. The first construction, the product

$$\mathcal{T} \times \mathcal{T}' \equiv \{((s, s'), (P, P'), (t, t')) \mid (s, P, t) \in \mathcal{T} \wedge (s', P', t') \in \mathcal{T}'\}$$

internalizes the two-execution nature of simulations. Second, we define the identity transition system for parameters $\mathcal{Z}$, denoted by $\mathcal{I}_{\mathcal{Z}}$, by $\{(z, *, z) \mid z \in \mathcal{Z}\}$ where $*$ is the unique value of some singleton set of labels.

The following lemma justifies these constructions by relating $\mathcal{Z}$-parametrized behaviour over $\mathcal{T} \times \mathcal{T}'$ to nonparametrized behaviour over $\mathcal{T} \times (\mathcal{T}' \times \mathcal{I}_{\mathcal{Z}})$, where $\overrightarrow{R}$ denotes the relation $\{(s, (s', z)).\ (s, s') \in R\, z\}$ for any $R : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$.

Lemma 6. *For $R, S : \mathcal{Z} \Rightarrow (\mathcal{S} \times \mathcal{S}')$ we have $\models^{\mathcal{T}'}_{\mathcal{T}} P \sim P' : R \Longrightarrow_{\mathcal{Z}} S$ exactly iff $\models^{(\mathcal{T}' \times \mathcal{I}_{\mathcal{Z}})}_{\mathcal{T}} P \sim (P', *) : \overrightarrow{R} \Longrightarrow \overrightarrow{S}$.*

Instantiating the parametrization mechanism to our language with objects, we may derive proof rules for judgements $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ formally defined as

$$\exists\, \phi.\, \triangleright\, C : \mathit{Dec}_{\mathsf{L}}\ \overrightarrow{R}\ \phi \wedge \triangleright\, C' : \mathit{Dec}_{\mathsf{R}}\ (\overrightarrow{S})^{\sharp}\ \phi^{\sharp}.$$

Here, the operation $\psi^{\sharp} \equiv \{((x, z), x') \mid (x, (x', z)) \in \psi\}$ shifts the auxiliary value z to the left component, so that it is not affected by the execution of C'. By construction, $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ implies $\models^{\mathcal{T}_{\mathsf{Obj}}}_{\mathcal{T}_{\mathsf{Obj}}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$. The proof rules for the system $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ are essentially the same as in Section 3, and are derived by incorporating the operators $(.)^{\sharp}$ and $\overrightarrow{(.)}$ into the construction of witnesses.

Noninterference for objects A typical use case for the parametrization mechanism consists of noninterference. Following Banerjee-Naumann [7], we consider a notion of indistinguishability that prevents an attacker from observing the precise location chosen during an allocation, and also allow each execution to allocate objects that have no counterpart in the opposite execution. Formally, this is modeled by parametrizing the relation $\sim$ by partial bijections over locations, i.e. sets $\beta \subseteq \mathcal{L}^2$ satisfying $(\ell = \ell_1) \Leftrightarrow (\ell' = \ell'_1)$ for any $(\ell, \ell') \in \beta$ and $(\ell_1, \ell'_1) \in \beta$.

Naturally, the bijections evolve throughout program execution according to the allocation of fresh objects, but in a conservative manner: the partial bijection relating the final states should be an extension of the one relating the initial states. We therefore parametrize the simulations by bijections, communicating the initial bijection to the postrelation. Indeed, for

$$R_{\mathsf{NI}} = \lambda\, \beta.\, \{(\sigma, \sigma').\, \sigma \sim_\beta \sigma'\} \qquad S_{\mathsf{NI}} = \lambda\, \beta.\, \{(\sigma, \sigma').\, \exists\, \gamma \supseteq \beta.\, \sigma \sim_\gamma \sigma'\}$$

noninterference coincides with $\models^{\mathcal{T}_{\mathsf{Obj}}}_{\mathcal{T}_{\mathsf{Obj}}} C \sim C : R_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} S_{\mathsf{NI}}$ and, in fact, also with $\models^{\mathcal{T}_{\mathsf{Obj}}}_{\mathcal{T}_{\mathsf{Obj}}} C \sim C : S_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} S_{\mathsf{NI}}$. This motivates the definition of the derived forms

$$\begin{aligned} \mathit{LOW}(C) &\equiv \vdash_{\mathsf{Par}} C \sim C : S_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} S_{\mathsf{NI}} \\ \mathit{HIGH}(C) &\equiv \vdash_{\mathsf{Par}} C \sim \mathbf{Skip} : R_{\mathsf{NI}} \Longrightarrow_{\mathcal{L}^2} R_{\mathsf{NI}}, \end{aligned}$$

that interpret, respectively, the judgements for noninterferent and publically unobservable code fragments. As the semantic interpretations S_{NI} and R_{NI} are transparent, the derived rules can be combined with direct uses of the underlying rules for $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ to integrate type-based with logical reasoning.

Error behaviour and separation logic We have also derived proof rules of unary and relational separation logic, including the appropriate frame rules. The derivations make crucial use of the parametrization mechanism, by instantiating $\mathcal{Z}$ to the type of (relational) assertions. This allows frame assertions to be joined onto the pre- and postconditions in a style reminiscent of Birkedal et al.'s Kripke resource extension [16]. Our encoding is derived from a variant of $\vdash_{\mathsf{Par}} C \sim C' : R \Longrightarrow_{\mathcal{Z}} S$ for a language where null dereferences and attempts to access undefined fields result in a fault/error state. The faultiness of states is exposed in the specifications of the unary and derived relational logics, enabling the interpretation of separation logic judgements to specify equi-fault-avoidance of the two phrases. We include the Isabelle-files of this development in [14] but are prevented from a detailed exposition by page limitations.

5 Discussion

Relational decomposition is a technique for integrating relational logics into stacks of unary verification frameworks [6, 4]. 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 relational program logics, and sketched applications to noninterference and separation logics. Our development is backed up by a formalization in Isabelle/HOL.

The formulation across different transition systems was crucial for our derivation of parametrized simulations. Future work will seek to exploit this flexibility for the verification of refinement and compiler correctness. Work on a relational logic for a bytecode-like language is under way, with a system for formally relating the two language levels as an intended subsequent step. Later, one might aim to support features such as arrays, exceptions, and methods. Our treatment of noninterference in [15] already supports parameterless but possibly recursive procedures, but transferring this development to virtual methods and non-lockstep method invocations is future work.

Concrete relational verification might benefit from formulating relational decomposition more algorithmically, so that the traversal of a program pair emits unary verification tasks, along the line of Terauchi and Aiken's work. Hints for the discovery of relational invariants may potentially arise from Amtoft et al.'s preconditions for conditional information flow [2], Barthe et al.'s product programs [10], from Rhodium's transformation rules [27], or from Tate et al.'s program equivalence graphs [36]. It would also be interesting to compare the expressiveness and usability of our rules for dissonant loops with the rules from translation validation [20], and to investigate how the latter can be justified in a more semantics-oriented fashion.

Natural extensions of noninterference include extensional notions of declassification [8], conditional information flow [3], and the explicit integration of noninterference and separation disciplines, following the work of Amtoft et al. [1]. Magill et al.'s two-step abstractions for reasoning about data structures may provide orientation how ghost variables and program instrumentation interact with separation aspects [29].

A more abstract treatment of our operators can be obtained using relational algebra. As pointed out by a referee, uncurrying $Dec_{\mathsf{L}}\ R\ \phi$ yields $(R \backslash \phi)^{-1}$ while uncurrying $Dec_{\mathsf{R}}\ S\ \phi$ yields the *weakest postspecification* S/ϕ given by $\overline{\phi^{-1}; \overline{S}}$. Extending the work of [22, 23], Gardiner [19] explores connections between these operators and predicate transformers to study a variation of bisimulation called power simulation. In contrast to our work, predicates and relations are formulated over a single universe.

Barthe et al.'s article [11] includes a self-composed treatment of separation, but restricted to a (termination- and) error-insensitive case and without a fine-grained object control via partial bijections. Reducing error-avoidance of self-composed programs to *equi*-error-avoidance of C and C' appears difficult as the execution of C' is conditional on the nonfaultiness of C's final state.

Saabas and Uustalu show how type derivations yield semantics-preserving proof transformations between pairs of judgements of unary Hoare logics [34].

A long-term goal is the integration of our techniques into verification infrastructures for mainstream languages such as the Verified Software Toolchain for C [5]. As a stepping stone towards this goal, fragments of C such as Spark/Ada [9] may represent a realistic testbed that is both industrially relevant and formally tractable.

Acknowledgments Andrew Appel encouraged me to revisit the earlier article with Martin Hofmann. The PL group at Princeton and the SAnToS group at Kansas State University provided numerous comments. I am grateful to Dave Naumann and the anonymous referees for their detailed feedback and several additional pointers to the literature.

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In Morrisett and Jones [30], pages 91–102.
2. T. Amtoft, J. Hatcliff, and E. Rodríguez. Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In A. D. Gordon, editor, *Programming Languages and Systems: Proceedings of the European Symposium on Programming (ESOP'10)*, volume 6012 of *LNCS*, pages 43–63. Springer, 2010.
3. 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.
4. A. W. Appel. Foundational high-level static analysis. In *Proceedings of the CAV 2008 Workshop on Exploiting Concurrency Efficiently and Correctly (EC^2)*, July 2008.
5. A. W. Appel. Verified software toolchain. In G. Barthe, editor, *Programming Languages and Systems: Proceedings of the European Symposium on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 1–17. Springer, Apr. 2011. Invited talk.
6. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411–445, 2007.
7. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, Mar. 2005.
8. 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.
9. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2006.
10. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. See `http://software.imdea.org/~ckunz/rellog/long-rellog.pdf`, 2011.
11. 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.
12. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Types in Language Design and Implementation*, pages 103–112. ACM Press, 2005.
13. 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)*, pages 14–25. ACM Press, 2004.
14. L. Beringer. Relational decomposition – Isabelle/HOL sources. Available at `www.cs.princeton.edu/~eberinge/RelDecompITP2011.tar.gz`, 2011.
15. L. Beringer and M. Hofmann. Secure information flow and program logics. In *IEEE Computer Security Foundations Symposium*, pages 233–248. IEEE Press, 2007.
16. 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.
17. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.
18. 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.
19. P. Gardiner. Power simulation and its relation to traces and failures refinement. *Theoretical Computer Science*, 309(1-3):157–176, 2003.

20. B. Goldberg, L. D. Zuck, and C. W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, 2005.
21. M. V. Hermenegildo and J. Palsberg, editors. *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL'10)*. ACM Press, 2010.
22. C. A. R. Hoare and J. He. The weakest prespecification. *Information Processing Letters*, 24(2):127 – 132, 1987.
23. C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71 – 76, 1987.
24. S. Hunt and D. Sands. On flow-sensitive security types. In Morrisett and Jones [30], pages 79–90.
25. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.
26. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.
27. 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 Symposium on Principles of Programming Languages (POPL'05)*, pages 364–377. ACM Press, 2005.
28. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
29. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In Hermenegildo and Palsberg [21], pages 211–222.
30. J. G. Morrisett and S. L. P. Jones, editors. *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*. ACM Press, 2006.
31. D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *Computer Security — ESORICS 2006, Proceedings of the 11th European Symposium on Research in Computer Security*, number 4189 in LNCS, pages 279–296. Springer, 2006.
32. G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, 2000.
33. 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.
34. A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1-2):131–154, 2008.
35. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, Jan. 2003.
36. R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In Hermenegildo and Palsberg [21], pages 389–402.
37. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis: Proceedings of the 12th International Symposium (SAS'05)*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
38. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
39. A. Voronkov and I. Narasamdya. Inter-program properties. In J. Palsberg and Z. Su, editors, *Proceedings of the 16th International Symposium on Static Analysis (SAS'09)*, volume 5673 of *LNCS*, pages 343–359. Springer, 2009.
40. H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.