Relational Logic with Framing and Hypotheses*

Anindya Banerjee[1], David A. Naumann[2], and Mohammad Nikouei[3]

1 **IMDEA Software Institute, Madrid, Spain**
2 **Stevens Institute of Technology, Hoboken, USA**
3 **Stevens Institute of Technology, Hoboken, USA**

Abstract

Relational properties arise in many settings: relating two versions of a program that use different data representations, noninterference properties for security, etc. The main ingredient of relational verification, relating aligned pairs of intermediate steps, has been used in numerous guises, but existing relational program logics are narrow in scope. This paper introduces a logic based on novel syntax that weaves together product programs to express alignment of control flow points at which relational formulas are asserted. Correctness judgments feature hypotheses with relational specifications, discharged by a rule for the linking of procedure implementations. The logic supports reasoning about program-pairs containing both similar and dissimilar control and data structures. Reasoning about dynamically allocated objects is supported by a frame rule based on frame conditions amenable to SMT provers. We prove soundness and sketch how the logic can be used for data abstraction, loop optimizations, and secure information flow.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Relational Hoare logic, program equivalence, product programs, frame conditions, region logic

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2016.11

1 Introduction

Relational properties are ubiquitous. Compiler optimizations, changes of data representation, and refactoring involve two different programs. Non-interference (secure information flow) is a non-functional property of a single program; it says the program preserves a "low indistinguishability" relation [44]. Many recent works deal with one or more of these applications, using relational logic and/or some form of product construction that reduces the problem to partial correctness, though mostly for simple imperative programs. This paper advances extant work by providing a relational logic for local reasoning about heap data structures and programs with procedures.

To set the stage, first consider the two simple imperative programs:

$$C \mathrel{\widehat{=}} x := 1;\ \mathsf{while}\ y > 0\ \mathsf{do}\ x := x * y;\ y := y - 1\ \mathsf{od}$$

$$C' \mathrel{\widehat{=}} x := 1;\ y := y - 1;\ \mathsf{while}\ y \geq 0\ \mathsf{do}\ x := x * y + x;\ y := y - 1\ \mathsf{od}$$

* A. Banerjee was partially supported by the U.S. National Science Foundation (NSF). D. A. Naumann and M. Nikouei were partially supported by NSF award 1228930. Any opinion, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of NSF.

Both C and C' change x to be the factorial of the initial value of y, or to 1 if y is initially negative. For a context where y is known to be positive and its final value is not used, we could reason that they are interchangeable by showing both

$$C\ :\ y = z \wedge y \geq 0 \ \rightsquigarrow\ x = z! \quad \text{and} \quad C'\ :\ y = z \wedge y \geq 0 \ \rightsquigarrow\ x = z! \tag{1}$$

This is our notation for partial correctness judgments, with evident pre- and postconditions, for C and C'. It is not always easy to express and prove functional correctness, which motivates a less well developed approach to showing interchangeability of the examples. The two programs have a relational property which we write as

$$(C|C')\ :\ \mathbb{B}(y \geq 0) \wedge y \doteq y \ \approx\!> \ x \doteq x \tag{2}$$

This relational correctness judgment says that a pair of terminating executions of C and C', from a pair of states which both satisfy $y \geq 0$ and which agree on the value of y, yields a pair of final states that agree on the value of x. The relational formula $x \doteq x$ says that the value of x in the left state is the same as its value in the right state.

Property (2) is a consequence of functional correctness (1), but there is a direct way to prove it. Any pair of runs, from states that agree on y, can be aligned in such a way that both $x \doteq x$ and $y \doteq y + 1$ hold at the aligned pairs of intermediate states. The alignment is almost but not quite step by step, owing to the additional assignment in C'. The relational property is more complicated than partial correctness, in that it involves pairs of runs. On the other hand the requisite intermediate assertions are much simpler; they do not involve ! which is recursively defined. Prior work showed such assertions are amenable to automated inference (see Section 7).

Despite the ubiquity of relational properties and recent logic-based or product-based approaches to reasoning with them (see Section 7), simple heap-manipulating examples like the following remain out of reach:

$$C'' \mathrel{\hat=} xp := \mathsf{new}\ Int(1);\ \mathsf{while}\ y > 0\ \mathsf{do}\ xp.set(xp.get() * y);\ y := y - 1\ \mathsf{od};\ x := xp.get()$$

This Java-like program uses get/set procedures acting on an object that stores an integer value, and $(C|C'')$ satisfies the same relational specification as (2). This code poses significant new challenges. It is not amenable to product reductions that rely on renaming of identifiers to encode two states as a single state: encoding of two heaps in one can be done, but at the cost of significant complexity [35] or exposing an underlying heap model below the level of abstraction of the programming language. Code like C'' also needs to be linked with implementations of the procedures it calls. For reasoning about two versions of a module or library, relational hypotheses are needed, and calls need to be aligned to enable use of such hypotheses.

Floyd [22] articulates the fundamental method of inductive assertions for partial correctness: establish that certain conditions hold at certain intermediate steps of computation, designating those conditions/steps by associating formulas with control flow points. For relational reasoning, pairs of steps need to be aligned and it is again natural to designate those in terms of points in control flow. Alignment of steps has appeared in many guises in prior work, often implicit in simulation proofs but explicit in a few works [47, 8, 28].

First contribution: In this paper we embody the alignment principle in a formal system at the level of abstraction of the programming language – as Hoare logic does for the inductive assertion method – with sufficient generality to encompass many uses of relational properties for programs including procedures and dynamically allocated mutable objects. Our logic (Section 6) manifests the reasoning principle directly, in structured syntax. It also embodies other reasoning principles, such as frame rules, case analysis, and hypothetical specifications for procedures. The rules encompass relations between both similarly- and differently-structured programs, and handle partially and fully aligned iterations. This achievement brings together ideas from many recent works (Section 7), together with two ingredients we highlight as contributions in their own right.

Second contribution: Our relational assertion language (Section 4) can describe agreement between unbounded pointer structures, allowing for differences in object allocation, as is needed to specify noninterference [4] and for simulation relations [3] in languages like Java and ML where references are abstract. Such agreements are expressed without the need for recursively defined predicates, and the assertion language has a direct translation to SMT-friendly encodings of the heap. (For lack of space we do not dwell on such encodings in this paper, which has a foundational focus, but see [40, 7].)

Third contribution: We introduce a novel form of "biprogram" (Section 5) that makes explicit the reasoner's choice of alignments. A biprogram run models an aligned pair of executions of the underlying programs. The semantics of biprograms involves a number of subtleties: To provide a foundation for extending the logic with encapsulation (based on [5]), we need to use small-step semantics – which makes it difficult to prove soundness of linking, even in the unary case [5]. For this to work we need to keep the semantics deterministic and to deal with semantics of hypotheses in judgments.

Section 2 provides background and Section 3 is an overview of the logic using examples. We have chosen to use the available space to explain fundamental intuitions. An accompanying technical report includes worked proofs of the examples, additional examples like a loop tiling transformation, details of semantics, and the soundness theorem.

2 Background: synopsis of region logic

For reasoning about the heap, separation logic is very effective, with modal operators that implicitly describe heap regions. But for relations on unbounded heap structures at the Java/ML level of abstraction we need explicit means to refer to heap regions, as in the dependency logic of Amtoft et al. [2]. Our relational logic is based on an underlying unary logic dubbed "region logic" (***RL***), developed in a series of papers [10, 5, 7] to which we refer for rationale and omitted details. RL is a Hoare logic augmented with some side conditions (first order verification conditions) which facilitate local reasoning about frame conditions [10] in the manner of dynamic frames [27, 31]. In the logic such reasoning hinges on a frame rule. In a verifier, framing can be done by the VC-generator, optionally guided by annotation [40]. Stateful frame conditions also support an approach to encapsulation that validates a second order frame rule (at the cost of needing to use small-step semantics) [5]. Read effects enable the use of pure method calls in assertions and in frame conditions [7] and are useful for proving some equivalences, like commuting assignments, that hold in virtue of disjointness of effects [15].

The logic is formalized for imperative programs with first order procedures and dynamically allocated mutable objects (records), see Fig. 1. As in Java and ML, references are distinct from integers; they can be tested for equality but there is no pointer arithmetic. Typing

$$m \in \mathit{ProcName} \qquad x, y, r \in \mathit{VarName} \qquad f, g \in \mathit{FieldName} \qquad K \in \mathit{DeclaredClassNames}$$

$$\begin{array}{lll}
\text{(Types)} & T & ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{rgn} \mid K \\
\text{(Program Expr.)} & E & ::= x \mid c \mid \mathsf{null} \mid E \oplus E \quad \text{where } c \text{ is in } \mathbb{Z} \text{ and } \oplus \text{ in } \{=, +, -, *, \geq, \wedge, \neg, \ldots\} \\
\text{(Region Expr.)} & G & ::= x \mid \varnothing \mid \{E\} \mid G\text{‘}f \mid G \otimes G \text{ where } \otimes \text{ is in } \{\cup, \cap, \setminus\} \\
\text{(Expressions)} & F & ::= E \mid G \\
\text{(Atomic comm.)} & A & ::= \mathsf{skip} \mid m() \mid x := F \mid x := \mathsf{new}\ K \mid x := x.f \mid x.f := x \\
\text{(Commands)} & C & ::= A \mid \mathsf{let}\ m = C\ \mathsf{in}\ C \mid \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ E\ \mathsf{do}\ C \mid C\,;C \\
\text{(Biprograms)} & CC & ::= (C|C) \mid \lfloor A \rfloor \mid \mathsf{let}\ m = (C|C)\ \mathsf{in}\ CC \mid CC\,;CC \\
 & & \quad \mid \mathsf{if}\ E|E\ \mathsf{then}\ CC\ \mathsf{else}\ CC \mid \mathsf{while}\ E|E \bullet \mathcal{P}|\mathcal{P}\ \mathsf{do}\ CC
\end{array}$$

Figure 1 Programs and biprograms. Assume each class type K has a declared list of fields, $\overline{f} : \overline{T}$. Biprograms are explained in Section 3.

of programs is standard. In specifications we use ghost variables and fields of type rgn. A ***region*** is a set of object references, which may include the improper null reference.

A ***specification*** $P \leadsto Q\ [\varepsilon]$ is comprised of precondition P, postcondition Q, and frame condition ε. Frame conditions include both read and write effects:

$$\varepsilon ::= \mathsf{rd}\, x \mid \mathsf{rd}\, G\text{‘}f \mid \mathsf{wr}\, x \mid \mathsf{wr}\, G\text{‘}f \mid \varepsilon, \varepsilon \mid (\mathit{empty})$$

The form $\mathsf{rd}\, G\text{‘}f$ means the program may read locations $o.f$ where o is a reference in the region denoted by expression G. We write $\mathsf{rw}\, x$ to abbreviate the composite effect $\mathsf{rd}\, x, \mathsf{wr}\, x$, and omit repeated tags: $\mathsf{rd}\, x, y$ abbreviates $\mathsf{rd}\, x, \mathsf{rd}\, y$. Predicate formulas P include standard first order logic with equality, region subset ($G \subseteq G$), and the "points-to" relation $x.f = E$, which says x is non-null and the value of field f equals E. A ***correctness judgment*** has the form $\Phi \vdash C : P \leadsto Q\ [\varepsilon]$ where the ***hypothesis context*** Φ maps procedure names to specifications. In C there may be ***environment calls*** to procedures bound by let inside C, and also ***context calls*** to procedures in Φ. The form $G\text{‘}f$ is termed an ***image expression***. For an example of image expressions, consider this command which sums the elements of a singly-linked null-terminated list, ignoring nodes for which a deletion flag, *del*, has been set.

$$C_1 \mathrel{\widehat{=}}\ s := 0; \mathsf{while}\ p \neq \mathsf{null}\ \mathsf{do}\ \mathsf{if}\ \neg p.\mathit{del}\ \mathsf{then}\ s := s + p.\mathit{val}\ \mathsf{fi};\ p := p.\mathit{nxt}\ \mathsf{od}$$

For its specification we use ghost variable r : rgn to contain the nodes. Its being closed under nxt is expressed by $r\text{‘}\mathit{nxt} \subseteq r$ in this specification:

$$p \in r \wedge r\text{‘}\mathit{nxt} \subseteq r \ \leadsto\ s = \mathit{sum}(\mathit{listnd}(\mathsf{old}(p)))\ [\mathsf{rw}\, s,\ p,\ \mathsf{rd}\, r,\ r\text{‘}\mathit{val},\ r\text{‘}\mathit{nxt},\ r\text{‘}\mathit{del}]$$

The r-value of the image expression $r\text{‘}\mathit{nxt}$ is the set of values of nxt fields of the objects in r. In frame conditions, expressions are used for their l-values. In this case, the frame condition uses image expressions to say that for any object o in r, locations $o.\mathit{val}, o.\mathit{nxt}, o.\mathit{del}$ may be read. The frame condition also says that variables s and p may be both read and written. Let function *listnd* give the mathematical list of non-deleted values.

Some proof rules in RL have side conditions which are first order formulas on one or two states. One kind of side condition, dubbed the "frames judgment", delimits the part of state on which a formula depends (its read effect). RL's use of stateful frame conditions provides for a useful frame rule, and even second order frame rule [37, 5], but there is a price to be paid. Frame conditions involving state dependent region expressions are themselves susceptible to interference by commands. That necessitates side conditions, termed "immunity" and "read-framed", in the proof rules for sequence and iteration [5, 7]. The frame rule allows to infer from $\Phi \vdash C :\ P \leadsto Q\ [\varepsilon]$ the conclusion $\Phi \vdash C :\ P \wedge R \leadsto Q \wedge R\ [\varepsilon]$ provided that

R is framed by read effects η (written η frm R) for locations disjoint from those writable according to ε (written $\eta \cdot/. \varepsilon$).

In keeping with our goal to develop a comprehensive deductive system, our unary and relational logics include a rule for discharging hypotheses, expressed in terms of the linking construct. Here is the special case of a single non-recursive procedure.

$$\textsc{Link}\ \frac{m : R \leadsto S\,[\eta] \vdash C : P \leadsto Q\,[\varepsilon] \vdash B : R \leadsto S\,[\eta]}{\vdash \mathsf{let}\ m = B\ \mathsf{in}\ C : P \leadsto Q\,[\varepsilon]}$$

3 Overview of the relational logic

This section sketches highlights of relational reasoning about a number of illustrative examples, introducing features of the logic incrementally. Some details are glossed over.

We write $(C|C') : \mathcal{Q} \approx\!> \mathcal{R}$ to express that a pair of programs C, C' satisfies the relational contract with precondition $\mathcal{Q}$ and postcondition $\mathcal{R}$, leaving aside frame conditions for now. The judgment constrains executions of C and C' from pairs of states related by $\mathcal{Q}$. (For the grammar of relational formulas, see (7) in Section 4.) It says neither execution faults (e.g., due to null dereference), and if both terminate then the final states are related by $\mathcal{R}$. Moreover no context procedure is called outside its precondition. (We call this property the $\forall\forall$ form, for contrast with refinement properties of $\forall\exists$ form.)

Assume f, g are pure functions. The programs

$$C_0 \ \widehat{=}\ x := f(z); y := g(z) \qquad C_0' \ \widehat{=}\ y := g(z); x := f(z)$$

are equivalent. Focusing on relevant variables, the equivalence can be specified as

$$(C_0 \mid C_0') : z \ddot{=} z \approx\!> x \ddot{=} x \land y \ddot{=} y \tag{3}$$

which can be proved as follows. Both C_0 and C_0' satisfy $\mathit{true} \leadsto x = f(z) \land y = g(z)$, which directly entails that $(C_0 \mid C_0') : \mathbb{B}\mathit{true} \approx\!> \mathbb{B}(x = f(z) \land y = g(z))$ by an embedding rule. The general form of embedding combines two different unary judgments, with different specifications, using relational formulas that assert a predicate on just the left ($\triangleleft$) or right ($\triangleright$) state. So $\mathbb{B}P$ is short for $\triangleleft P \land \triangleright P$. Since z is not written by C_0 or C_1, we can introduce $z \ddot{=} z$ using the relational frame rule, to obtain $(C_0 \mid C_0') : z \ddot{=} z \approx\!> \mathbb{B}(x = f(z) \land y = g(z)) \land z \ddot{=} z$. This yields (3) using the relational rule of consequence with the two valid relational assertion schemas $u \ddot{=} u' \land \triangleleft(u = v) \land \triangleright(u' = v') \Rightarrow v \ddot{=} v'$ and $z \ddot{=} z \Rightarrow f(z) \ddot{=} f(z)$.

For the factorial example $(C|C')$ in Section 1, we would like to align the loops and use the simple relational invariant $x \ddot{=} x \land y \ddot{=} y + 1$. We consider the form $(C|C')$ as a biprogram which can be rewritten to equivalent forms using the ***weaving*** relation which preserves the underlying programs but aligns control points together so that relational assertions can be used. (A minor difference from most other forms of product program is that we do not need to rename apart the variables on the left and right.) The weaving relation is given in Section 5. In this case we weave to the form

$$(x := 1|x := 1; y := y - 1); \mathsf{while}\ y > 0 \mid y \geq 0\ \mathsf{do}\ (x := x * y \mid x := x * y + x); \lfloor y := y - 1 \rfloor$$

This enables us to assert the relational invariant at the beginning and end of the loop bodies. Indeed, we can also assert it just before the last assignments to y. The rule for this form of loop requires the invariant to imply equivalence of the two loops' guard conditions, which it does: $x \ddot{=} x \land y \ddot{=} y + 1 \Rightarrow (y > 0 \ddot{=} y \geq 0)$. For a biprogram of the ***split*** form $(C|C')$, the primary reasoning principle is the lifting of unary judgments about C and C'. For an atomic command A, the ***sync*** notation $\lfloor A \rfloor$ is an alternative to $(A|A)$ that indicates its left and right transition are considered together. This enables the use of relational specifications for procedures, and a relational principle for object allocation. For an ordinary assignment, sync merely serves to abbreviate, as in $\lfloor y := y-1 \rfloor$ above.

The next example involves the heap and it also involves a loop that is "dissonant" in the sense that we do not want to align all iterations – that is, *alignment is ultimately about traces,* not program texts. Imagine the command C_1 from Section 2 is run on a list from which secret values have been deleted. To specify that no secrets are leaked, we use the relational judgment $(C_1|C_1) : \mathit{listnd}(p) \ddot{=} \mathit{listnd}(p) \approx\!> s \ddot{=} s$ which says: Starting from any two states containing the same non-deleted values, terminating computations agree on the sums. The judgment can be proved by showing the functional property that s ends up as $\mathit{sum}(\mathit{listnd}(\mathsf{old}(p)))$. But we can avoid reasoning about list sums and prove this relational property by aligning some of the loop iterations in such a way that $\mathit{listnd}(p) \ddot{=} \mathit{listnd}(p) \wedge s \ddot{=} s$ holds at every aligned pair, that is, it is a relational invariant. Not every pair of loop iterations should be aligned: When $p.\mathit{del}$ holds for the left state but not the right, a left-only iteration maintains the invariant, and *mutatis mutandis* when $p.\mathit{del}$ holds only on the right. To handle such non-aligned iterations we use a novel syntactic annotation dubbed ***alignment guards***. The idea is that the loop conditions are in agreement, and thus the iterations are synchronized, unless one of the alignment guards hold – and then that iteration is unsynchronized but the relational invariant must still be preserved. We weave $(C_1|C_1)$ to the form

$$\begin{array}{l}\lfloor s := 0 \rfloor;\ \mathsf{while}\ p \neq \mathsf{null} \mid p \neq \mathsf{null} \bullet \triangleleft (p.\mathit{del}) \mid \triangleright (p.\mathit{del}) \\ \qquad\qquad \mathsf{do\ if}\ \neg p.\mathit{del} \mid \neg p.\mathit{del}\ \mathsf{then}\ \lfloor s := s + p.\mathit{val} \rfloor\ \mathsf{fi};\ \lfloor p := p.\mathit{nxt} \rfloor\ \mathsf{od}\end{array} \tag{4}$$

with alignment guards $\triangleleft p.\mathit{del}$ and $\triangleright p.\mathit{del}$. The rule for the while biprogram has three premises for the loop body: for executions on the left (resp. right) under alignment guard $\triangleleft p.\mathit{del}$ (resp. $\triangleright p.\mathit{del}$) and for simultaneous executions when neither of the alignment guards hold. Each premise requires the invariant to be preserved.

The final example is a change of data representation. It illustrates dynamic allocation and frame conditions, as well as procedures and linking. A substantive example of this sort would be quite lengthy, so we contrive a toy example to provide hints of the issues that motivate various elements of our formal development. Our goal is to prove a conditional equivalence between these programs, whose components are defined in due course.

$$C_4 \mathrel{\widehat{=}} \mathsf{let}\ \mathit{push}(x : \mathsf{int}) = B\ \mathsf{in}\ \mathit{Cli} \qquad C_4' \mathrel{\widehat{=}} \mathsf{let}\ \mathit{push}(x : \mathsf{int}) = B'\ \mathsf{in}\ \mathit{Cli}$$

These differ only in the implementations B, B' of the stack interface (here stripped down to a single procedure), to which the client program Cli is linked. For modular reasoning, the unary contract for push should not expose details of the data representation. We also want to avoid reliance on strong functional specifications – the goal is equivalence of the two versions, not functional correctness of the client. The client, however, should respect encapsulation of the stack representation, to which end frame conditions are crucial. A simple pattern is for contracts to expose a ghost variable rep (of type rgn) for the set of objects considered to be owned by a program module. Here is the specification for push, with parts named for later reference. Let size and rep be ***spec-public***, i.e., they can be used in public contracts but not in client code [30].

$$\begin{array}{ll}\mathit{push}(x : \mathsf{int}) :\ R \rightsquigarrow S[\eta]\ \text{where} & R \mathrel{\widehat{=}} \mathit{size} < 100 \\ & S \mathrel{\widehat{=}} \mathit{size} = \mathsf{old}(\mathit{size}) + 1 \\ & \eta \mathrel{\widehat{=}} \mathsf{rw}\, \mathit{rep}, \mathit{size}, \mathit{rep}\text{`}\mathsf{any}\end{array} \tag{5}$$

Variables rep and $size$ can be read and written (keyword rw) by $push$. This needs to be explicit, even though client code cannot access them, because reasoning about client code involves them. The notation $rep\text{`}\mathsf{any}$ designates all fields of objects in rep; these too may be read and written. The specification makes clear that calls to $push$ affect the encapsulated state, while not exposing details. Here is one implementation of $push(x)$.

$$B \mathrel{\widehat{=}} top := \mathsf{new}\ Node(top, x);\ \ rep := rep \cup \{top\};\ \ size\text{++}$$

Variable top is considered internal to the stack module, so it need not appear in the frame condition. The alternate implementation of $push$ replaces top by module variables $free : \mathsf{int};\ slots : String[\,];$.

$$B' \mathrel{\widehat{=}} \mathsf{if}\ slots = \mathsf{null}\ \mathsf{then}\ slots := \mathsf{new}\ String[100];\ \ rep := rep \cup \{slots\};\ \ free := 0\ \mathsf{fi}; \\ slots[free\text{++}] := x;\ \ size\text{++}$$

Correctness of the two versions is proved using module invariants

$$I \mathrel{\widehat{=}} (top = \mathsf{null} \wedge size = 0)\ \vee (top \in rep \wedge rep\text{`}nxt \subseteq rep \wedge size = length(list(top)))$$

$$I' \mathrel{\widehat{=}} (slots = \mathsf{null} \wedge size = 0) \vee (slots \in rep \wedge size = free)$$

Here $list(top)$ is the mathematical list of values reached from top. Recall that in an assertion the expression $rep\text{`}nxt$ is the image of set $rep$ under the $nxt$ field, i.e., the set of values of $nxt$ fields of objects in $rep$. The condition $rep\text{`}nxt \subseteq rep$ says that rep is closed under nxt. This form is convenient in using ghost code to express shapes of data structures without recourse to reachability or other inductive predicates [10, 40].

As a specific Cli, we consider one that allocates and updates a node of the same type as used by the list implementation; this gets assigned to a global variable p.

$$Cli \mathrel{\widehat{=}} push(1);\ \ p := \mathsf{new}\ Node(\mathsf{null}, 2);\ \ p.val := 3;\ \ push(4)$$

Having completed the definitions of C_4, C'_4 we can ask: In what sense are C_4, C'_4 equivalent? A possible specification for $(C_4|C'_4)$ requires agreement on $size$ and ensures agreement on $size$ and on p and $p.val$. However, the latter agreements cannot be literal equality: following the call $push(1)$, one implementation has allocated a $Node$ whereas the array implementation has not. Depending on the allocator, different references may be assigned to p in the two executions. The appropriate relation is "equivalence modulo renaming of references" [2, 3, 4, 16, 17]. For region expression G and field name f, we write $\mathbb{A}G\text{`}f$ for the ***agreement*** relation that says there is a partial bijection on references between the two states, that is total on the region $G$, and for which corresponding $f$-fields are equal. The notation $\mathbb{A}G\text{`}\mathsf{any}$ means agreement on all fields. In the present example, we only need the singleton region $\{p\}$ containing the reference denoted by p.

To prove a relational judgment for $(C_4|C'_4)$ we need suitable relational judgments for $(B|B')$ for the implementations of $push$. It is standard [26] that they should preserve a "coupling relation" that connects the two data representations and also includes the data invariants for each representation. For the example, the connection is that the sequence of elements reached from top, written $list(top)$, is the same as the reversed sequence of elements in $slots[0..free-1]$. Writing rev for reversal, we define the coupling and specification

$$\mathcal{L} \mathrel{\widehat{=}} \triangleleft I \wedge \triangleright I' \wedge LtR \qquad LtR \mathrel{\widehat{=}} list(top) \ddot{=} rev(\langle\,\rangle\ \mathsf{if}\ slots = \mathsf{null}\ \mathsf{else}\ slots[0..free-1])$$

$$(C_4|C'_4) :\ \mathbb{B}(size = 0) \wedge \mathcal{L} \ \approx\!\!> \ p \ddot{=} p \wedge size \ddot{=} size \wedge \mathbb{A}\{p\}\text{`}\mathsf{any} \wedge \mathcal{L} \tag{6}$$

We now proceed to sketch a proof of (6). First, we weave $(C_4|C_4')$ to let $push(x : \mathsf{int}) = (B|B')$ in $\llbracket Cli \rrbracket$. Here $\llbracket Cli \rrbracket$ abbreviates the fully aligned biprogram $\lfloor push(1) \rfloor; \lfloor p := \mathsf{new}\ Node(\mathsf{null}, 2) \rfloor; \lfloor p.val := 3 \rfloor; \lfloor push(4) \rfloor$. This biprogram simultaneously links the procedure bodies on left and right, and aligns the client. Using $\lfloor p := \mathsf{new}\ Node(\mathsf{null}, 2) \rfloor$ enables use of a relational postcondition that says the objects are in agreement. Using $\lfloor push(4) \rfloor$ enables use of *push*'s relational specification.

Like in unary RL, the proof rule for linking has two premises: one says the bodies $(B|B')$ satisfy their specification, the other says $\llbracket Cli \rrbracket$ satisfies the overall specification under the hypothesis that *push* satisfies its spec (see rLink in Fig. 2). This hypothesis context gives *push* a relational specification, using $\mathbb{A}x$ as sugar for $x \ddot{=} x$:

$$\Phi \mathrel{\widehat{=}} push(x) : \quad \mathbb{B}R \wedge \mathbb{A}size \wedge \mathbb{A}x \wedge \mathcal{L} \ \approx\!> \ \mathbb{B}S \wedge \mathbb{A}size \wedge \mathcal{L} \ [\eta, \mathsf{rw}\ top \mid \eta, \mathsf{rw}\ slots, free]$$

Here η is the effect $\mathsf{rw}\ rep, size, rep`\mathsf{any}$ in the original specification (5) of *push*.

The specification in Φ is not simply a relational lift of *push*'s public specification (5). Invariants I and I' on internal data structures should not appear in *push*'s API: they should be hidden, because the client should not touch the internal state on which they depend. Effects on module variables (like *top*) should also be hidden. This kind of reasoning is the gist of second order framing [37, 5]. The relational counterpart is a relational second order frame rule which says that any client that respects encapsulation will preserve $\mathcal{L}$. Hiding is the topic of another paper, for which this one is laying the groundwork (see Section 8).

4 Relational formulas

The relational assertion language is essentially syntax for a first order structure comprised of the variables and heaps of two states, together with a refperm connecting the states.

$$\mathcal{P} ::= \quad F \ddot{=} F \mid \mathbb{A}G`f \mid \Diamond\mathcal{P} \mid \triangleleft P \mid \triangleright P \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \Rightarrow \mathcal{P} \mid \forall x|x' : K.\, \mathcal{P} \qquad (7)$$

A ***refperm*** is a type-respecting partial bijection from references allocated in one state to references allocated in the other state. For use with SMT provers, a refperm can be encoded by a pair of maps with universal formulas stating they are inverse [7]. The syntax for relations caters for dynamic allocation by providing primitives such as $F \ddot{=} F'$ that says the value of F in the left state equals that of F' in the right state, modulo the refperm. In case of integer expressions, this is ordinary equality. For reference expressions, it means the two values are related by the refperm. For region expressions, $G \ddot{=} G'$ means the refperm forms a bijection between the reference set denoted by G in the left state and G' in the right state (ignoring *null*). The agreement formula $\mathbb{A}G`f$ says, of a pair of states, that the refperm is total on the set denoted by G in the left state, and moreover the f-field of each object in that set has the same value, modulo refperm, as the f-field of its corresponding object in the right state.

For commands that allocate, the postcondition needs to allow the refperm to be extended, which is expressed by the modal operator $\Diamond$ (read "later"): $\Diamond\mathcal{P}$ holds if there is an extension of the refperm with zero or more pairs of references for which $\mathcal{P}$ holds. For example, after the assignment to p in the stack example, the relational rule for allocation yields postcondition $\Diamond(p \ddot{=} p \wedge \mathbb{A}\{p\}`\mathsf{any})$. Aside from the left and right embeddings of unary predicates ($\triangleleft P$ and $\triangleright P$), the only other constructs are the logical ones (conjunction, implication, quantification over values).

Let $\Box\mathcal{P} \mathrel{\widehat{=}} \neg\Diamond\neg\mathcal{P}$. Validity of $\mathcal{P} \Rightarrow \Box\mathcal{P}$ is equivalent to $\mathcal{P}$ being ***monotonic***, i.e., not falsified by extension of the refperm. Here are some valid schemas: $\mathcal{P} \Rightarrow \Diamond\mathcal{P}$, $\Diamond\Diamond\mathcal{P} \Rightarrow \Diamond\mathcal{P}$, and $\Diamond(\mathcal{P} \wedge \mathcal{Q}) \Rightarrow \Diamond\mathcal{P} \wedge \Diamond\mathcal{Q}$. The converse of the latter is not valid. For framing, a key property is that $\diamond\mathcal{P} \wedge \mathcal{Q} \Rightarrow \diamond(\mathcal{P} \wedge \mathcal{Q})$ is valid if $\mathcal{Q}$ is monotonic. In practice, $\diamond$ is only needed in postconditions, and only at the top level. Owing to $\diamond\diamond\mathcal{P} \Rightarrow \diamond\mathcal{P}$, this works fine with sequenced commands. Many useful formulas are monotonic, including $\mathbb{A}G`f$ and $F \doteqdot F'$, but not $\neg(F \doteqdot F')$.

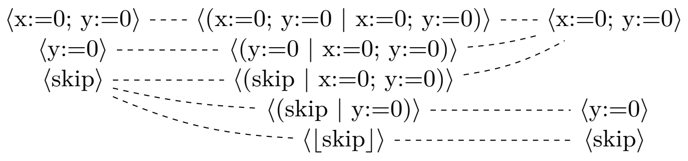
5 Biprograms

A biprogram CC (Fig. 1) represents a pair of commands, which are given by syntactic projections defined by clauses including the following: $\overleftarrow{(C|C')} \mathrel{\hat=} C$, $\overrightarrow{(C|C')} \mathrel{\hat=} C'$, $\overleftarrow{\lfloor A \rfloor} \mathrel{\hat=} A$, $\overleftarrow{\mathsf{if}\ E|E'\ \mathsf{then}\ BB\ \mathsf{else}\ CC} \mathrel{\hat=} \mathsf{if}\ E\ \mathsf{then}\ \overleftarrow{BB}\ \mathsf{else}\ \overleftarrow{CC}$, and $\overleftarrow{\mathsf{let}\ m = (C|C')\ \mathsf{in}\ CC} \mathrel{\hat=} \mathsf{let}\ m = C\ \mathsf{in}\ \overleftarrow{CC}$. The weaving relation has clauses including the following.

$(A|A) \hookrightarrow \lfloor A \rfloor$ (for atomic commands A)
$(C; D \mid C'; D') \hookrightarrow (C|C'); (D|D')$
$(\mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D \mid \mathsf{if}\ E'\ \mathsf{then}\ C'\ \mathsf{else}\ D') \hookrightarrow \mathsf{if}\ E|E'\ \mathsf{then}\ (C|C')\ \mathsf{else}\ (D|D')$
$(\mathsf{while}\ E\ \mathsf{do}\ C \mid \mathsf{while}\ E'\ \mathsf{do}\ C') \hookrightarrow \mathsf{while}\ E|E' \bullet \mathcal{P}|\mathcal{P}'\ \mathsf{do}\ (C|C')$ (for any $\mathcal{P}, \mathcal{P}'$)

Additional clauses are needed for congruence, e.g., $CC \hookrightarrow DD$ implies $BB; CC \hookrightarrow BB; DD$. The loop weaving introduces chosen alignment guards. The ***full alignment*** of a command C is written $\llbracket C \rrbracket$ and defined by $\llbracket A \rrbracket \mathrel{\hat=} \lfloor A \rfloor$, $\llbracket C; D \rrbracket \mathrel{\hat=} \llbracket C \rrbracket; \llbracket D \rrbracket$, $\llbracket \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D \rrbracket \mathrel{\hat=} \mathsf{if}\ E|E\ \mathsf{then}\ \llbracket C \rrbracket\ \mathsf{else}\ \llbracket D \rrbracket$, $\llbracket \mathsf{while}\ E\ \mathsf{do}\ C \rrbracket \mathrel{\hat=} \mathsf{while}\ E|E \bullet \mathsf{false}|\mathsf{false}\ \mathsf{do}\ \llbracket C \rrbracket$, etc. Note that $(C|C) \hookrightarrow^* \llbracket C \rrbracket$ for any C.

Commands are deterministic (modulo allocation), so termination-insensitive noninterference and equivalence properties can be expressed in a simple $\forall\forall$ form described at the start of Section 3, rather than the $\forall\exists$ form needed for refinement and for possibilistic noninterference ("for all runs . . . there exists a run . . . "). The transition rules for biprograms must ensure that the behavior is compatible with the underlying unary semantics, while enforcing the intended alignment. That would still allow some degree of nondeterminacy in biprogram transitions. However, we make biprograms deterministic (modulo allocation), because it greatly simplifies the soundness proofs. Rather than determinize by means of a scheduling oracle or other artifacts that would clutter the semantics, we build determinacy into the transition semantics. Whereas the syntax aligns points of interest in control flow, biprogram traces explicitly represent aligned pairs of executions. We make the arbitrary choice of left-then-right semantics for the split form. In a trace of $(C|C')$, every step taken by C is effectively aligned with the initial state for C'. This is followed by the steps of C', each aligned with the final state of C. To illustrate the idea, here is a sketch of the trace of a split biprogram (center column) and its alignment with left and right unary traces.



This pattern is also typical for "high conditionals" in noninterference proofs, where different branches may be taken (cf. rule rIf4). Here is the sync'd version in action.

```
⟨x:=0; y:=0⟩ ---- ⟨⌊x:=0⌋; ⌊y:=0⌋⟩ ---- ⟨x:=0; y:=0⟩
   ⟨y:=0⟩ ----------- ⟨⌊y:=0⌋⟩ ----------- ⟨y:=0⟩
   ⟨skip⟩ ------------ ⟨⌊skip⌋⟩ ------------ ⟨skip⟩
```

$$\text{rLink}\ \frac{m : \mathcal{R} \approx\!> \mathcal{S}\,[\eta] \vdash \llbracket C \rrbracket : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon] \qquad \vdash (B|B') : \mathcal{R} \approx\!> \mathcal{S}\,[\eta]}{\vdash \mathsf{let}\ m = (B|B')\ \mathsf{in}\ \llbracket C \rrbracket : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]}$$

$$\text{rIf4}\ \frac{\begin{array}{ll}\Phi \vdash (C|C') : \mathcal{P} \land \triangleleft E \land \triangleright E' \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon'] & \Phi \vdash (C|D') : \mathcal{P} \land \triangleleft E \land \triangleright \neg E' \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon'] \\ \Phi \vdash (D|C') : \mathcal{P} \land \triangleleft \neg E \land \triangleright E' \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon'] & \Phi \vdash (D|D') : \mathcal{P} \land \triangleleft \neg E \land \triangleright \neg E' \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']\end{array}}{\Phi \vdash (\mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ D | \mathsf{if}\ E'\ \mathsf{then}\ C'\ \mathsf{else}\ D') : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon, \mathit{ftpt}(E) | \varepsilon', \mathit{ftpt}(E')]}$$

$$\text{rIf}\ \frac{\begin{array}{c}\mathcal{P} \Rightarrow E \doteq E' \\ \Phi \vdash CC : \mathcal{P} \land \triangleleft E \land \triangleright E' \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon'] \qquad \Phi \vdash DD : \mathcal{P} \land \triangleleft \neg E \land \triangleright \neg E' \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']\end{array}}{\Phi \vdash \mathsf{if}\ E|E'\ \mathsf{then}\ CC\ \mathsf{else}\ DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon, \mathit{ftpt}(E) | \varepsilon', \mathit{ftpt}(E')]}$$

$$\text{rWeave}\ \frac{\begin{array}{c}\Phi \vdash DD : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon'] \\ CC \hookrightarrow DD \quad \mathit{unaryOnly}(\Phi) \quad \mathit{terminates}(\overleftarrow{\mathcal{P}}, \overleftarrow{DD}) \quad \mathit{terminates}(\overrightarrow{\mathcal{P}}, \overrightarrow{DD})\end{array}}{\Phi \vdash CC : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']}$$

Figure 2 Selected relational proof rules.

The relational correctness judgment has the form $\Phi \vdash CC : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']$. The hypothesis context Φ maps some procedure names to their specifications: $\Phi(m)$ may be a unary specification as before or else a relational one of the form $\mathcal{R} \approx\!> \mathcal{S}\,[\varepsilon|\varepsilon']$. Frame conditions retain their meaning, separately for the left and the right side. In case ε is the same as ε', the judgment or specification is abbreviated as $\mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon]$.

The semantics of biprograms uses small steps, which makes alignments explicit. A configuration is comprised of a biprogram, two states, and two environments for procedures. The transition relation depends on a semantic interpretation for each procedure in the hypothesis context Φ. Context calls, i.e., calls to procedures in the context, take a single step in accord with the interpretation. For the sake of determinacy, this is formalized in the semantics of relational correctness by quantifying over deterministic "interpretations" of the specifications (as in [7]), rather than a single nondeterministic transition rule (as in [5, 37]).

Let us sketch the semantic consistency theorem, which confirms that executions of a biprogram from a pair of states correspond to pairs of executions of the underlying commands, so that judgments about biprograms represent relational properties of the underlying commands. Suppose $\Phi \vdash (C|C') : \mathcal{P} \approx\!> \mathcal{Q}\,[\varepsilon|\varepsilon']$ is valid and Φ has only unary specifications. Consider any states σ, σ' that are related by $\mathcal{P}$ (modulo some refperm). Suppose C and C', when executed from σ, σ', reach final states τ, τ'. (In the formal semantics, transitions are defined in terms of interpretations φ that satisfy the specifications Φ, so this is written $\langle C,\ \sigma \rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau \rangle$ and $\langle C',\ \sigma' \rangle \overset{\varphi}{\longmapsto}{}^* \langle \mathsf{skip},\ \tau' \rangle$.) Then τ, τ' satisfy $\mathcal{Q}$.

6 Relational region logic

Selected proof rules appear in Fig. 2.

For linking a procedure with its implementation, rule rLink caters for a client program C related to itself, in such a way that its executions can be aligned to use the same pattern of calls. The procedure implementations may differ, as in the stack example, Section 3. The rule shown here is for the special case of a single procedure, and the judgment for $(B|B')$ has empty hypothesis context, to disallow recursion. We see no difficulty to add mutually recursive procedures, as done for the unary logic in [5], but have not yet included that in a detailed soundness proof. The soundness proof is basically an induction on steps as in [5] but with the construction of an interpretation as in the proof of the linking rule in [7]. The general rule also provides for un-discharged hypotheses for ambient libraries used in the client and in the procedure implementations [5].

Rule rIf4 is the obvious rule that considers all paths for a conditional not aligned with itself (e.g., for "high branches"), whereas rIf leverages the alignment designated by the biprogram form. The disjunction rule – i.e., from $\Phi \vdash CC : \mathcal{P}_0 \approx> \mathcal{Q} [\varepsilon|\varepsilon']$ and $\Phi \vdash CC : \mathcal{P}_1 \approx> \mathcal{Q} [\varepsilon|\varepsilon']$ infer $\Phi \vdash CC : \mathcal{P}_0 \vee \mathcal{P}_1 \approx> \mathcal{Q} [\varepsilon|\varepsilon']$ – serves to split cases on the initial states, allowing different weavings to be used for different circumstances, which is why there is no notion like alignment guards for the biprogram conditional. The obvious conjunction rule is sound. It is useful for deriving other rules. For example, we have this simple axiom for allocation: $\vdash \lfloor x := \mathsf{new}\ K \rfloor : \mathit{true} \approx> \diamond(x \doteq x)$ [$\mathsf{wr}\, x, \mathsf{rw\, alloc}$]. Using conjunction, embedding, and framing, one can add postconditions like $\mathbb{A}\{x\}$'f and freshness of x.

A consequence of our design decisions is "one-sided divergence" of biprograms, which comes into play with weaving. For example, assuming *loop* diverges, $(y := 0; z.f := 0 \mid \mathit{loop}; x := 0)$ assigns $z.f$ before diverging. But it weaves to $(y := 0|\mathit{loop}); (z.f := 0|x := 0)$ which never assigns $z.f$. This biprogram's executions do not cover all executions of the underlying unary programs. The phenomenon becomes a problem for code that can fault (e.g., if z is null). Were the correctness judgments to assert termination, this shortcoming would not be an issue, but in this paper we choose the simplicity of partial correctness. Rule rWeave needs to be restricted to prevent one-sided divergence of the premise biprogram DD from states where CC in the conclusion terminates. For simplicity in this paper we assume given a termination check: $\mathit{terminates}(P, C)$ means that C faults or terminates normally, from any initial state satisfying P, This is about unary programs, so the condition can be discharged by standard means.

The relational frame rule is a straightforward extension of the unary frame rule. From a judgment $\Phi \vdash CC : \mathcal{P} \approx> \mathcal{Q} [\varepsilon|\varepsilon']$ it infers $\Phi \vdash CC : \mathcal{P} \wedge \mathcal{R} \approx> \mathcal{Q} \wedge \mathcal{R} [\varepsilon|\varepsilon']$ provided that $\mathcal{R}$ is framed by read effects (on the left and right) that are disjoint from the write effects in $\varepsilon|\varepsilon'$.

To prove a judgment $\Phi \vdash \mathsf{while}\ E|E' \bullet \mathcal{P}|\mathcal{P}'\ \mathsf{do}\ CC : \mathcal{Q} \approx> \mathcal{Q} [\varepsilon, \mathit{ftpt}(E)|\varepsilon', \mathit{ftpt}(E')]$, the rule has three main premises: $\Phi \vdash (\overleftarrow{CC}|\mathsf{skip}) : \mathcal{Q} \wedge \mathcal{P} \wedge \triangleleft E \approx> \mathcal{Q} [\varepsilon|\,]$ for left-only execution of the body, $\Phi \vdash (\mathsf{skip}|\overrightarrow{CC}) : \mathcal{Q} \wedge \mathcal{P}' \wedge \triangleright E' \approx> \mathcal{Q} [\,|\varepsilon']$ for right-only, and $\Phi \vdash CC : \mathcal{Q} \wedge \neg\mathcal{P} \wedge \neg\mathcal{P}' \wedge \triangleleft E \wedge \triangleright E' \approx> \mathcal{Q} [\varepsilon|\varepsilon']$ for aligned execution. A side condition requires that the invariant $\mathcal{Q}$ implies these cases are exhaustive: $\mathcal{Q} \Rightarrow E \doteq E' \vee (\mathcal{P} \wedge \triangleleft E) \vee (\mathcal{P}' \wedge \triangleright E')$. Additional side conditions require the effects to be self-immune, just as in unary RL [10, 7]. Finally, the formulas $\diamond\mathcal{P} \Rightarrow \mathcal{P}$ and $\diamond\mathcal{P}' \Rightarrow \mathcal{P}'$ must be valid; this says the alignment guards are refperm-independent, which is needed because refperms are part of the semantics of judgments but are not part of the semantics of biprograms.

The above rule is compatible with weaving a loop body, as in (4). The left and right projections $\overleftarrow{CC}$ and $\overrightarrow{CC}$ undo the weaving and take care of unaligned iterations.

There are many other valid and useful rules. Explicit frame conditions are convenient, both in tools and in a logic, in part because they compose in simple ways. This may lose precision, but that can be overcome using postconditions to express, e.g., that $x := x$ does not observably write x. This is addressed, in unary RL, by a rule to "mask" write effects [10]. Similarly, the relational logic supports a rule to mask read effects. There is a rule of transitivity along these lines: from $(B|C) : \mathcal{P} \approx> \mathcal{Q}$ and $(C|D) : \mathcal{R} \approx> \mathcal{S}$ infer $(B|D) : \mathcal{P};\mathcal{R} \approx> \mathcal{Q};\mathcal{S}$ where $(;)$ denotes composition of relations. A special case is where the pre-relations (resp. post-relations) are the same, transitive, relation. The rule needs to take care about termination of C.

7 Related work

Benton [15] introduced relational Hoare logic, around the same time that Yang was developing relational separation logic [45]. Benton's logic does not encompass the heap. Yang's does; it features separating conjunction and a frame rule. Pointers are treated concretely in [45]; agreement means identical addresses, which suffices for some low level C code. Neither work includes procedures. Beringer [18] reduces relational verification to unary verification via specifications and uses that technique to derive rules of a relational Hoare logic for programs including the heap (but not procedures). Whereas the logics of Benton, Yang, and others provide only rules for synchronized alignment of loops, Beringer derives a rule that allows for unsynchronized ("dissonant") iterations; our alignment guards are similar to side conditions of that rule. RHTT [34] implements a relational program logic in dependent type theory (Coq). The work focuses on applications to information flow. It handles dynamically allocated mutable state and procedures, and both similar and dissimilar control structures. Like the other relational logics it does not feature frame conditions. RHTT is the only prior relational logic to include both the heap and procedures, and the only one to have a procedure linking rule. It is also the only one to address any form of encapsulation; it does so using abstract predicates, as opposed to hiding [5, 37].

Several works investigate construction of product programs that encode nontrivial choices of alignment [38, 42, 46, 11, 12, 13]. In particular, our weaving relation was inspired by [11, 13] which address programs that differ in structure. In contrast to the 2-safety properties for deterministic programs considered in this paper and most prior work, Barthe et al. [12] handle properties of the form "for all traces . . . there exists a trace . . . " which are harder to work with but which encompass notions of refinement and continuity. Relational specifications of procedures are used in a series of papers by Barthe et al. (e.g.,[14]) for computer-aided cryptographic proofs. Sousa and Dillig [41] implement a logic that encompasses k-ary relations, e.g., the 3-safety property that a binary method is computing a transitive relation; their verification algorithm is based on an implicit product construction. None of these works address the heap or the linking of procedure implementations. Several works show that syntactic heuristics can often find good weavings in the case of similarly-structured programs not involving the heap [28, 32, 41]. Mueller et al. [32] use a form of product program and a relational logic to prove correctness of a static analysis for dependency, including procedures but no heap.

Works on translation validation and conditional equivalence checking use verification conditions (VCs) with implicit or explicit product constructions [46, 47]. Godlin and Strichman formulate and prove soundness of rules for proving equivalence of programs with similar control structure [23]. They use one of the rules to devise an algorithm for VCs using uninterpreted functions to encode equivalence of called procedures, which has been implemented in two prototype tools for equivalence checking [24]. (Pointer structures are limited to trees, i.e., no sharing.) Hawblitzel et al. [25] and Lahiri et al. [29] use relational procedure summaries for intra- and inter-procedural reasoning about program transformations. The heap is modeled by maps. These and related works report good experimental results using SMT or SAT solvers to discharge VCs. Felsing et al. [21] use Horn constraint solving to infer coupling relations and relational procedure summaries, which works well for similarly structured programs; they do not deal with the heap. The purpose of our logic is not to supplant VC-based tools approaches but rather to provide a foundation for them. Our biprograms and relational assertions are easily translated to SMT-based back ends like Boogie and Why3.

Amtoft et al. [2] introduce a logic for information flow in object-based programs, using abstract locations to specify agreements in the heap. It was proposed in [8] to extend this approach to more general relational specifications, for fine-grained declassification policies. Banerjee et al. [9] showed how region-based reasoning including a frame rule can be encoded, using ghost code, with standard FOL assertions instead of an ancillary notion of abstract region. This evolved to the logic in Section 6.

Relational properties have been considered in the context of separation logic: [19] and [43] both give relational interpretations of unary separation logic that account for representation independence, using second order framing [19] or abstract predicates [43]. Extension of this work to a relational logic seems possible, but the semantics does not validate the rule of conjunction so it may not be a good basis for verification tools. Tools often rely heavily on splitting conjunctions in postconditions.

Ahmed et al. [1] address representation independence for higher order code and code pointers, using a step-indexed relational model, and prove challenging instances of contextual equivalence. Based on that work, Dreyer et al. [20] formulate a relational modal logic for proving contextual equivalence for a language that has general recursive types and general ML-style references atop System F. The logic serves to abstract from details of semantics in ways likely to facilitate interactive proofs of interesting contextual equivalences, but it includes intensional atomic propositions about steps in the transition semantics of terms. Whereas contextual equivalence means equivalent in all contexts, general relational logics can express equivalences conditioned on the initial state. For example, the assignments $x := y.f$ and $z.f := w$ do not commute, in general, because their effects can overlap. But they do commute under the precondition $y \neq z$. We can easily prove equivalence judgments such as $(x := y.f; z.f := w \mid z.f := w; x := y.f) : \mathbb{B}(y \neq z) \wedge \mathbb{A}\{y\}`f \wedge w \ddot{=} w \approx\!> x \ddot{=} x \wedge \mathbb{A}\{z\}`f$. By contrast with [1, 34], we do not rely on embedding in higher-order logic.

8 Conclusion

We provide a general relational logic that encompasses the heap and includes procedures. It handles both similarly- and differently-structured programs. We use small-step semantics with the goal to leverage, in future work, our prior work on SMT-friendly heap encapsulation [40, 5, 7] for representation independence, which is not addressed in prior relational logics.[1]

As articulated long ago by Hoare [26] but never fully formalized in a logic of programs, reasoning about change of data representation is based on simulation relations on encapsulated state, which are necessarily preserved by client code in virtue of encapsulation. For functional correctness this corresponds to “hiding” of invariants on encapsulated data, i.e., not including the invariant in the specification used by a client. O’Hearn et al. [37] formalize this as a hypothetical or second order framing rule (which has been adapted to RL [5]). In ongoing work, the logic presented here has been extended to address encapsulation and provides a relational second order frame rule which embodies Reynolds’ abstraction theorem [39]. Whereas framing of invariants relies on write effects, framing of encapsulated relations also relies on read effects. Our ongoing work also addresses observational purity, which is known to be closely related to representation independence [26, 36].

Although we can prove equivalence for loop tiling, some array-oriented loop optimizations seem to be out of reach of the logic as currently formulated. Loop interchange changes

[1] With the partial exception of [1], see Section 7. Although there has been some work on observational equivalence for higher order programs, we are not aware of work dealing with general relational judgments for higher order programs.

matrix row to column order, reordering unboundedly many atomic assignments, as does loop fusion/distribution. Most prior work does not handle these examples; [47] does handle them, with a non-syntactic proof rule that involves permutations on transition steps, cf. [33].

References

1 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages*, 2009.

2 T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *ACM Symposium on Principles of Programming Languages*, 2006.

3 Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005.

4 Anindya Banerjee and David A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

5 Anindya Banerjee and David A. Naumann. Local reasoning for global invariants, part II: Dynamic boundaries. *Journal of the ACM*, 60(3):19:1–19:73, 2013.

6 Anindya Banerjee and David A. Naumann. A logical analysis of framing for specifications with pure method calls. In *Verified Software: Theories, Tools and Experiments*, volume 8471 of *LNCS*, 2014.

7 Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. A logical analysis of framing for specifications with pure method calls. Under review for publication. Extended version of [6]. `http://www.cs.stevens.edu/~naumann/pub/readRL.pdf`, 2015.

8 Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, 2008.

9 Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, 2008.

10 Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Local reasoning for global invariants, part I: Region logic. *Journal of the ACM*, 60(3):18:1–18:56, 2013.

11 Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Formal Methods*, volume 6664 of *LNCS*, 2011.

12 Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Logical Foundations of Computer Science, International Symposium*, volume 7734 of *LNCS*, 2013.

13 Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *J. Logical and Algebraic Methods in Programming*, 2016. To appear.

14 Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3):9, 2013.

15 Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM Symposium on Principles of Programming Languages*, 2004.

16 Nick Benton, Martin Hofmann, and Vivek Nigam. Abstract effects and proof-relevant logical relations. In *ACM Symposium on Principles of Programming Languages*, 2014.

17 Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *International Symposium on Principles and Practice of Declarative Programming*, 2007.

18 Lennart Beringer. Relational decomposition. In *Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*, 2011.

19 Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. *Logical Methods in Computer Science*, 4(2), 2008. `doi:10.2168/LMCS-4(2:6)2008`.

20 Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful ADTs. In *ACM Symposium on Principles of Programming Languages*, 2010.

21 Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *International Conference on Automated Software Engineering*, 2014.

22 Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.

23 Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6):403–439, 2008.

24 Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.

25 Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. Towards modularly comparing programs using automated theorem provers. In *International Conference on Automated Deduction*, 2013.

26 C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

27 Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, 2011. doi:10.1007/s00165-010-0152-5.

28 Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *ACM Conference on Computer and Communications Security*, 2013.

29 Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, 2013.

30 Gary T. Leavens and Peter Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering*, 2007.

31 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, LNCS, 2010.

32 Christian Mueller, Máté Kovács, and Helmut Seidl. An analysis of universal information flow based on self-composition. In *IEEE Computer Security Foundations Symposium*, 2015.

33 Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. In *Static Analysis Symposium*, volume 9837 of *LNCS*, 2016.

34 Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Trans. Program. Lang. Syst.*, 35(2):6, 2013. doi:10.1145/2491522.2491523.

35 David A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security*, volume 4189 of *LNCS*, 2006.

36 David A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205–224, 2007.

37 Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Transactions on Programming Languages and Systems*, 31(3):1–50, 2009.

38 John C. Reynolds. *The Craft of Programming.* Prentice-Hall, 1981.

39 John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 1983*, pages 513–523. North-Holland, 1984.

40 Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Decision procedures for region logic. In *Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, 2012.

41 Marcelo Sousa and Isil Dillig. Cartesian Hoare logic for verifying k-safety properties. In *ACM Conf. on Program. Lang. Design and Implementation*, 2016.

42 Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *International Static Analysis Symposium*, volume 3672 of *LNCS*, 2005.

43 Jacob Thamsborg, Lars Birkedal, and Hongseok Yang. Two for the price of one: Lifting separation logic assertions. *Logical Methods in Computer Science*, 8(3), 2012.

44 Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

45 Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

46 Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *Formal Methods*, volume 5014 of *LNCS*, 2008.

47 Lenore D. Zuck, Amir Pnueli, Benjamin Goldberg, Clark W. Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design*, 27(3):335–360, 2005.