A machine-checked framework for relational separation logic

Juan Manuel Crespo[1] and César Kunz[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] Universidad Politécnica de Madrid

Abstract. Relational methods are gaining growing acceptance for specifying and verifying properties defined in terms of the execution of two programs—notions such as simulation, observational equivalence, non-interference, and continuity can be elegantly casted in this setting. In previous work, we have proposed *program product construction* as a technique to reduce relational verification to standard verification. This method hinges on the ability to interpret relational assertions as traditional predicates, which becomes problematic when considering assertions from relational separation logic. We report in this article an alternative method that overcomes this difficulty, defined as a relational weakest precondition calculus based on separation logic and formalized in the Coq proof assistant. The formalization includes an application to the formal verification of the Schorr-Waite graph marking algorithm. We discuss additional variants of relational separation logic inspired by the standard notions of partial and total correctness, and extensions of the logic to handle non-structurally equivalent programs.

1 Introduction

Separation logic [16, 24, 25] is a formalism devised to verify pointer programs using local reasoning; its extensions and variants have been used successfully in a variety of large scale programs [31] and smaller but challenging examples [18], including lock-free algorithms [14].

Relational reasoning, on the other hand, provides an effective means to understand program behavior: in particular, it allows one to establish that the same program behaves similarly on two different runs, or that two programs execute in a related fashion. Relational judgments are often formalized by quadruples $\{\varphi\}\, c_1 \sim c_2\, \{\psi\}$, denoting that every pair of executions of c_1 and c_2 with initial states related by φ returns with final states related by ψ. Prime examples of relational properties include notions of simulation and observational equivalence, and 2-properties, such as non-interference and continuity.

Syntactic methods [7] have been developed to support relational reasoning. In particular, relational separation logic [30] is a variant of separation logic that supports reasoning about two pointer programs; it embodies the conventional wisdom that casting program correctness as an equivalence between two programs is often more beneficial than functional verification. More concretely, relational separation logic is intended to prove program correctness by showing the equivalence between the program to be verified and a reference implementation: e.g. Yang [30] provides an elegant proof in

relational separation logic that the Schorr-Waite graph marking algorithm is equivalent to depth-first search.

However, these syntactic methods suffer from two important caveats: on the one hand, these logics confine reasoning to structurally equivalent programs with equivalent guards; on the other hand, tool support is negligible, with the exception of recent work by Aleks Nanevski *et al* [23]—which focuses mainly on the specification and proof of a rich set of security policies and its static enforcement. Although the relational postconditions used to describe such policies can be arbitrary relations between pairs of initial, final heaps and results, this tool seems to be specially tailored to reason about two runs of the same program, rather than about two different programs. To some extent it is possible to circumvent such restriction by casting two different programs `P` and `P'` as a single program with a guard deciding which program to execute, i.e. `if x then P else P'`. However, this approach seems a bit awkward and it is not at all clear whether doing this can enable reasoning in terms of relational invariants—which is essential to keep invariants simple.

In recent work [4] we propose a technique—*product program construction*—that reduces relational program reasoning to traditional program reasoning—even for non-structurally equivalent programs. Perhaps more importantly, it enables the use of traditional verification tools, circumventing two of the main issues of techniques supporting relational reasoning. However, this method relies on the ability to interpret relational assertions (predicates on two states) as traditional assertions (predicates on one state), but this is not straightforward when using assertions from relational separation logic. More precisely, an issue arises when trying to interpret the relational assertion

$$R = \begin{pmatrix} p \\ q \end{pmatrix}$$

(two heaps h_1 and h_2 are related by R if p holds in h_1 and q holds in h_2) as a predicate $p \star q$. Note that this interpretation induces a loss of information: predicate R holds for a fixed partition of the heap while the latter holds for any partition of the heap. This loss of information renders our method unsound. Indeed, $\{P \star \mathsf{emp}\}\mathsf{skip};\mathsf{skip}\{\mathsf{emp} \star P\}$ is a valid separation logic judgment for all P, whereas the following relational judgment is not:

$$\begin{pmatrix} P \\ \mathsf{emp} \end{pmatrix} \mathsf{skip} \sim \mathsf{skip} \begin{pmatrix} \mathsf{emp} \\ P \end{pmatrix}$$

We present in this article an alternative approach that overcomes the difficulties of relational verification by product construction, based on a *weakest precondition calculus* for relational separation logic. The calculus is complete and formalized in the Coq proof assistant, and can be regarded as a first step towards providing tool support for relational methods that enables reasoning about heap manipulating programs.

The formalization provides a framework to reason about a small imperative language —using a deep embedding—with heap manipulating instructions very similar to the one described in Yang's article. We have formalized its semantics and provided a soundness proof of the relational weakest precondition. Local reasoning is supported by proving that the calculus is compatible with the frame rule. Also, we have defined an alternative calculus ensuring total relational correctness relying on *variants* (or ranking functions) defining a well-founded order on states.

$$
\begin{array}{rl}
\textbf{(integer expressions)} & E ::= 0 \mid 1 \mid E + E \mid E \times E \mid E - E \mid \mathsf{hd}(L) \\
\textbf{(boolean expressions)} & B ::= \mathsf{false} \mid B \Rightarrow B \mid E = E \mid E < E \mid L = L \\
\textbf{(list expressions)} & L ::= \alpha \mid \epsilon \mid E{::}L \mid \mathsf{tl}(L) \\
\textbf{(instructions)} & I ::= x := \mathsf{alloc}(E) \mid x := [E] \mid [E] := E \mid \mathsf{free}(E) \mid \\
 & \qquad x := E \mid \alpha := L \mid \mathsf{assert}(B) \\
\textbf{(statements)} & C ::= I \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \mid \mathsf{skip}
\end{array}
$$

Fig. 1. Syntax of Programs

The Coq formalization has been used to provide a formal proof of the equivalence of Depth First Search and the Schorr-Waite graph marking algorithm, reproducing the proof of the Schorr-Waite graph marking algorithm performed by Yang. We have extended Yang's prove to the total relational correctness case, hence ensuring that both programs terminate. The formal proof of the Schorr-Waite algorithm required a slightly stronger loop invariant, indicating perhaps a small weakness in the original specification provided in Yang's article.

We also introduce an extension of the relational calculus beyond structurally equivalent program, preserving relational reasoning over loop invariants, and thus retaining the aforementioned advantages. We illustrate the application of the calculus with the validation of a complex program optimization.

The present article does not intend to be a thorough description of the Coq framework but to present the main ideas in an amenable way. The interested reader is invited to study the Coq formalization, which has been done using the ssreflect library [13]. The sources are available online at `http://software.imdea.org/~ckunz/coqrelwp/`

Contents. The rest of the paper is structured as follows: Section 2 describes the formalization of the relational weakest precondition calculus, instantiated with a simple programming setting. In this section, we briefly review relational separation logic and present the main properties of the calculus: soundness and framing. Also, we present a variant of the calculus that ensures termination of both programs. Section 3 presents our main case study, the proof of equivalence between the Schorr-Waite graph marking algorithm w.r.t. depth-first search. Section 4 describes an extension to non-structurally equivalent code.

2 Formalization of relational separation logic

We start this section by introducing a simple program setting and then we provide an overview of relational separation logic. Afterwards, we develop our relational calculus based on weakest precondition computation.

The programming language presented in Figure 1 is a mild extension of the typical setting used in standard separation logic [24] to include list expressions. List values are rather uncommon in similar formalizations of imperative languages but are included here to ease the description of the Depth First Search (DFS) algorithm, which uses a stack as auxiliary data structure. In the figure, α stands for a list variable. We let BExp denote the set of boolean expressions and Stmt the set of statements.

State model. We let $\mathcal{S}$ denote the set of states. A state comprises two components: the store and the heap. The store itself comprises two components to accommodate two types of expressions: natural numbers and lists. Each of the store components is modeled the usual way, as a finite mapping from scalar variables in $\mathsf{Var_N}$ to natural numbers and as a finite mapping from list variables in $\mathsf{Var_L}$ to lists of natural numbers. We assume that the sets of variables $\mathsf{Var_N}$ and $\mathsf{Var_L}$ are disjoint. We let $\mathsf{upd}(x, n, s)$ stand for the result of updating the variable x to value n in the store s.

The heap is modeled as a finite mapping from locations (natural numbers) to values. The special location 0 is denoted null and cannot belong to the domain of a heap. Heaps are equipped with several operations such as look-up, free, fresh, disjoint union and interact in the expected way:

Expression	Meaning
$\mathsf{freshn}(h, n)$	base location for a sequence of n consecutive free cells in h;
$\mathsf{look}(h, n)$	value of the cell n in the heap h;
$\mathsf{mut}(n, m, h)$	result of setting the contents of cell n of heap h to m;
$\mathsf{dealloc}(h, n)$	result of freeing cell n from heap h;

Moreover, we let $\mathsf{dom}(h)$ stand for the set of allocated locations of heap h, and $h_1 \uplus h_2$ denote the disjoint union of heaps h_1 and h_2. In the actual Coq development, failure is captured in an error monad, but for simplicity we omit these details here. Much of the formalization is adapted from Nanevski *et al* [22].

Semantics of basic instructions. The semantics of an instruction i is modeled as a relation $[\![i]\!]$ on states; the rules are given in Fig. 2. The denotation of an instruction is a relation between states. States are noted as tuples (h, s_i, s_l) where h represents the heap and s_i and s_l denote the stores for integer and list variables, respectively. The instruction $x := \mathsf{alloc}(E)$ evaluates the expression E to a natural number n and then allocates n free contiguous heap cells, initializes them with value 0 and sets the value of x to the first allocated cell. The look up instruction $x := [E]$ evaluates expression E to a location n and if it is allocated it updates the value of variable x to the contents of the heap cell n. The mutation instruction $[E_1] := E_2$ evaluates E_1 to a location n and if n is a valid location in the current heap, this is modified so that it maps n to the result of evaluating E_2. A field access $x := E.f$ is used as a syntactic sugar of $x := [E+f]$, when the field identifier f represents a known offset. Similarly, we use $E_1.f := E_2$ as a syntax sugar of $[E_1+f] := E_2$. The instruction $\mathsf{free}(E)$ releases the heap cell allocated at the location represented by E. The assert instruction has blocking semantics. The remaining assignments for integer and list variables are completely standard.

Semantics of commands. The semantics $[\![c]\!]$ of a command is defined as a relation on states (big step style), using as auxiliary definition the semantics of boolean expressions, modeled as a function from states to booleans. The definitions are standard and omitted. Also, we denote $\langle c, \mu\rangle \rightsquigarrow \langle c', \mu'\rangle$ the small-step command semantics and we use $\rightsquigarrow^\star$ for its reflexive transitive closure. Obviously these two semantic styles are sound and complete w.r.t. each other, i.e. $[\![c]\!]\,\mu\,\mu'$ if and only if $\langle c, \mu\rangle \rightsquigarrow^\star \langle \mathsf{skip}, \mu'\rangle$. Also, we say that a command c is φ-safe if for any μ such that $\varphi\,\mu$ there exists μ' and c' such that $\langle c, \mu\rangle \rightsquigarrow \langle c', \mu'\rangle$, i.e., c is not stuck in φ-states.

$$
\begin{array}{ll}
((h,s_i,s_l),(h',s_i',s_l')) \in [\![x := \mathsf{alloc}(E)]\!] & \doteq s_i' = \mathsf{upd}(x,m,s_i) \\
& \quad \wedge\ s_l' = s_l \wedge h' = h \uplus (\biguplus_{i=0}^{n-1} (m+i) \mapsto 0) \\
& \quad \wedge\ m = \mathsf{freshn}(h,n) \wedge n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![x := [E]]\!] & \doteq s_l' = s_l \wedge h' = h \wedge s_i' = \mathsf{upd}(x,\mathsf{look}(h,n),s_i) \\
& \quad \wedge\ n \in \mathsf{dom}(h) \wedge n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![[E_1] := E_2]\!] & \doteq s_i' = s_i \wedge s_l' = s_l \wedge h' = \mathsf{mut}(n,m,h) \\
& \quad \wedge\ n \in \mathsf{dom}(h) \wedge n \in ([\![E_1]\!]\ (h,s_i,s_l)) \\
& \quad \wedge\ m \in ([\![E_2]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![\mathsf{free}(E)]\!] & \doteq s_i' = s_i \wedge s_l' = s_l \wedge h' = \mathsf{dealloc}(h,n) \\
& \quad \wedge\ n \in \mathsf{dom}(h) \wedge n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![x := E]\!] & \doteq h' = h \wedge s_l' = s_l \wedge s_i' = \mathsf{upd}(x,n,s_i) \\
& \quad \wedge\ n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![\alpha := L]\!] & \doteq h' = h \wedge s_i' = s_i \wedge s_l' = \mathsf{upd}(\alpha,xs,s_l) \\
& \quad \wedge\ xs \in ([\![L]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![\mathsf{assert}(B)]\!] & \doteq [\![b]\!](h,s_i,s_l) \wedge h = h' \wedge s_i = s_i' \wedge s_l = s_l'
\end{array}
$$

Fig. 2. Semantics of basic instructions

2.1 Relational calculus

We introduce in this section the relational calculus establishing the validity of relational specifications. Relational judgments are formalized as quadruples of the form $\{\varphi\}\, c_1 \sim c_2\, \{\psi\}$, where φ and ψ are relations on states and c_1 and c_2 are programs, establishing a relation over every pair of executions of c_1 and c_2, as formalized in the following definition:

Definition 1 (valid relational judgment). *Two commands c_1 and c_2 satisfy the pre and post-relation φ and ψ, denoted by the judgment $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ if for all states μ_1, μ_2 s.t. $[\![\varphi]\!]\ \mu_1\ \mu_2$ one of the following holds:*

- *c_1 diverges with initial state μ_1 iff c_2 diverges with initial state μ_2; or*
- *for all states μ_1' and μ_2' s.t. $[\![c_1]\!]\ \mu_1\ \mu_1'$ and $[\![c_2]\!]\ \mu_2\ \mu_2'$ we have $[\![\psi]\!]\ \mu_1'\ \mu_2'$.*

Assertions. Rather than representing assertions as syntactic objects, we have modeled them as relations between states. All of the assertions presented in Yang's work have a straightforward interpretation as state relations. The definition of some of them is shown in Figure 3. We let P, Q stand for relational assertions and p, q for standard separation logic assertions.

Adopting a shallow embedding of assertions provides extra flexibility by not committing beforehand to a particular logical language, and allows inheriting all the features of Coq's rich higher-order language. This proved to be convenient when defining a weakest precondition calculus ensuring termination, in which a well-founded relation must be provably decreasing throughout loop iterations—see Subsection 2.2.

$$
\begin{array}{ll}
\mathsf{Same}\ st_1\ st_2 & \doteq st_1.h = st_2.h \\
\mathsf{emp2}\ st_1\ st_2 & \doteq st_1.h = empty \wedge st_2.h = empty \\
(P \star Q)\ st_1\ st_2 & \doteq \exists h_{11}\ h_{12}\ h_{21}\ h_{22}. \\
 & \quad st_1.h = h_{11} \uplus h_{12} \wedge st_2.h = h_{21} \uplus h_{22} \\
 & \wedge\ P(h_{11}, st_1.s_i, st_1.s_l)(h_{21}, st_2.s_i, st_2.s_l) \\
 & \wedge\ Q(h_{12}, st_1.s_i, st_1.s_l)(h_{22}, st_2.s_i, st_2.s_l) \\
\binom{p}{q}\ st_1\ st_2 & \doteq p\ st_1 \wedge q\ st_2
\end{array}
$$

Fig. 3. Definition of some Relational Assertions

$$
\begin{array}{ll}
\mathsf{wp}(x := \mathsf{alloc}(E))\ \varphi\ (h, s_i, s_l) & \doteq \forall n\ m.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \wedge m = \mathsf{freshn}(h, n) \Rightarrow \\
 & \quad \varphi(h \uplus \biguplus_{i=0}^{n-1} (m+i) \mapsto 0, s_i, s_l) \\
\mathsf{wp}(x := [E])\ \varphi\ (h, s_i, s_l) & \doteq \forall n.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \Rightarrow \\
 & \quad \varphi(h, \mathsf{upd}(x, \mathsf{look}(h, n), s_i), s_l) \\
\mathsf{wp}([E_1] := E_2)\ \varphi\ (h, s_i, s_l) & \doteq \forall n\ m.\ n \in (\llbracket E_1 \rrbracket\ (h, s_i, s_l)) \wedge m \in (\llbracket E_2 \rrbracket\ (h, s_i, s_l)) \Rightarrow \\
 & \quad n \in \mathsf{dom}(h) \wedge \varphi(\mathsf{mut}(n, m, h), s_i, s_l) \\
\mathsf{wp}(\mathsf{free}(E))\ \varphi\ (h, s_i, s_l) & \doteq \forall n.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \Rightarrow \\
 & \quad n \in \mathsf{dom}(h) \wedge \varphi(\mathsf{dealloc}(h, n), s_i, s_l) \\
\mathsf{wp}(x := E)\ \varphi\ (h, s_i, s_l) & \doteq \forall n.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \Rightarrow \varphi(h, \mathsf{upd}(x, n, s_i), s_l) \\
\mathsf{wp}(\alpha := L)\ \varphi\ (h, s_i, s_l) & \doteq \forall xs.\ xs \in (\llbracket L \rrbracket\ (h, s_i, s_l)) \Rightarrow \varphi(h, s_i, \mathsf{upd}(\alpha, xs, s_l)) \\
\mathsf{wp}(\mathsf{assert}(B))\ \varphi\ (h, s_i.s_l)) & \doteq \llbracket B \rrbracket(h, s_i.s_l) \wedge \varphi(h, s_i.s_l)
\end{array}
$$

Fig. 4. Weakest Precondition of basic instructions

Weakest precondition calculus for basic instructions. Most program verification tools rely on weakest precondition calculi rather than program logics: concretely, the prevailing means to verify programs against a pre-condition and a post-condition is to generate a set of proof obligations using a weakest precondition calculus, and finally to discharge the proof obligations using automatic or interactive provers. Our formalization supports a similar methodology for relational judgments, and provides a weakest precondition calculus that computes a set of proof obligations from relational judgments. The weakest precondition of a basic instruction i w.r.t. to a state predicate ϕ is again, a state predicate (a function taking states and returning propositions). Here instead of using λ-abstractions we write the state on the left side as arguments to the wp function. Moreover, by abuse of notation we use pattern matching, i.e. a state is noted as a tuple. The definition of the weakest precondition of the basic instructions is provided in figure 4. Its definition is straightforward and obviously sound w.r.t. the semantics.

Weakest precondition calculus for 2-statements. Our weakest precondition calculus wp_2 operates on 2-statements, which combine two structurally equivalent statements into a single construction. Formally, the set Stmt_2 of 2-statements is defined inductively by the clauses: i) if i_1 and i_2 are instructions, then $[i_1, i_2]$ is a 2-statement; ii) if c, c_1, c_2 are 2-statements and b, b' are boolean expressions, then $c_1; c_2$, and

$$\mathsf{wp}_2\ \langle i_1, i_2\rangle\ \phi = \mathsf{wp}\ i_1\ (\lambda m_1.\ \mathsf{wp}\ i_2\ (\lambda m_2.\ \phi\ m_1\ m_2))$$

$$\mathsf{wp}_2\ (c_1; c_2)\ \phi = \mathsf{wp}_2\ c_1\ (\mathsf{wp}_2\ c_2\ \phi)$$

$$\mathsf{wp}_2\ (\mathsf{if}\ \langle b, b'\rangle\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \phi = \Psi_{b,b'} \wedge (b_{\langle 1\rangle} \Rightarrow \mathsf{wp}_2\ c_1\ \phi) \wedge (\neg b_{\langle 1\rangle} \Rightarrow \mathsf{wp}_2\ c_2\ \phi)$$

$$\mathsf{wp}_2\ (\mathsf{while}\ \langle b, b'\rangle\ \mathsf{do}\ c)\ \phi = \exists \varphi.\varphi \wedge \forall m_1, m_2.\ \Psi_{b,b'}\ m_1\ m_2 \wedge \Psi_\varphi\ m_1\ m_2 \wedge \Psi_\phi\ m_1\ m_2$$

where

$$\Psi_{b,b'} \doteq b_{\langle 1\rangle} \Leftrightarrow {b'}_{\langle 2\rangle} \qquad \textit{guard equivalence}$$
$$\Psi_\varphi \doteq \varphi \wedge b_{\langle 1\rangle} \Rightarrow \mathsf{wp}_2\ c\ \varphi \qquad \textit{invariant preservation}$$
$$\Psi_\phi \doteq \varphi \wedge \neg b_{\langle 1\rangle} \Rightarrow \phi \qquad \textit{valid postcondition}$$

Fig. 5. Relational weakest precondition calculus

if $\langle b, b'\rangle$ then c_1 else c_2, and while $\langle b, b'\rangle$ do c are 2-statements. Each 2-statement yields two structurally equivalent statements; we write $c \triangleright (c_1, c_2)$ to denote that c is a 2-statement whose left and right components are the statements c_1 and c_2 respectively. Conversely, any two structurally equivalent statements yield a 2-statement. Intuitively, a 2-statement encodes the simultaneous execution of its components, restricting the calculus to structurally similar programs. In Section 4 we explain how to remove this restriction by the application of a preliminary program transformation.

The weakest precondition calculus wp_2 is defined inductively on the structure of 2-statements; the rules are given in Figure 5, where $b_{\langle 1\rangle}$ and $b_{\langle 2\rangle}$ respectively denote the interpretation of the expression b in the first and second memories, and the extension of connectives to relations is defined in the usual way.

Frame rule. The frame rule lies at the very heart of any separation logic based verification framework, being the cornerstone of so called "local reasoning". In order to support modular verification, we have shown that it holds on the framework presented in this paper. Let P, Q, and R be relational assertions and c a 2-statement. Then, if

1. the proposition $\forall st_1, st_2.\ P\ st_1\ st_2 \Rightarrow \mathsf{wp}_2\ c\ Q\ st_1\ st_2$ holds; and
2. R is independent of the variables modified by c

then the following proposition holds:

$$\forall st_1, st_2.\ (P \star R)\ st_1\ st_2 \Rightarrow \mathsf{wp}_2\ c\ (Q \star R)\ st_1\ st_2$$

Intuitively, the hypothesis 1 and 2 implies that the only part of state that program c is allowed to inspect or operate on is described by P and any other part R will remain unchanged after its execution. This simplifying result has been systematically used to ease the verification of the Schorr-Waite algorithm.

One of the most challenging aspects of characterizing the frame rule in our setting is the fact that there is no syntax for assertions, so the customary side condition on R is formulated semantically by defining the modset of c, i.e. the set of modified variables, and requiring the validity of R to be independent of it.

Soundness. The calculus is sound, i.e. for all statements c_1 and c_2, and 2-statement c s.t. $c \triangleright (c_1, c_2)$, and assertions φ and ψ,

$$[\![\varphi]\!] \subseteq [\![\mathsf{wp}_2\ c\ \psi]\!] \quad \Longrightarrow \quad \vDash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$$

Moreover, the weakest precondition calculus is sound and complete w.r.t. relational separation logic, i.e. for all statements c_1 and c_2, and 2-statement c s.t. $c \triangleright (c_1, c_2)$, and assertions φ and ψ,

$$[\![\varphi]\!] \subseteq [\![\mathsf{wp}_2\ c\ \psi]\!] \quad \Longleftrightarrow \quad \vdash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$$

where $\vdash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$ is used to denote that the judgment is derivable in relational separation logic.

2.2 Total correctness

One can modify the weakest precondition calculus wp_2 to enforce total correctness. To this end, one must provide for each while statement a variant relation between pairs of initial and final states, and prove that it is a well-founded order (i.e. no infinite descending chains) and that it decreases with each iteration. The clause for the while statement is modified accordingly:

$$\begin{aligned}&\mathsf{wp}_2^{\mathrm{tc}}\ (\mathsf{while}\ \langle b, b'\rangle\ \mathsf{do}\ c)\ \phi\ \mu_1\ \mu_2 \doteq \\ &\qquad \exists \varphi, \exists \mu, \varphi \wedge \forall m_1, m_2.\ (\Psi_{b,b'}\ m_1\ m_2 \wedge \Psi_{\varphi}\ m_1\ m_2 \wedge \Psi_{\phi}\ m_1\ m_2) \wedge \\ &\qquad \mathsf{wellfounded}(\mu) \wedge \forall m_1, m_2.\ (\varphi\ m_1\ m_2 \wedge [\![b]\!]\ m_1\ m_2 \Rightarrow \\ &\qquad \mathsf{wp}_2\ c\ (\lambda s_1, s_2.\ \mu\ (s_1, s_2)\ (m_1, m_2))\ m_1\ m_2)\end{aligned}$$

where $\Psi_{b,b'}$, Ψ_{φ}, and Ψ_{ϕ} are defined as in Figure 5 (replacing wp_2 by $\mathsf{wp}_2^{\mathrm{tc}}$), and μ stands for the variant relation. The predicate $\mathsf{wellfounded}(\mu)$ requires μ to be well-founded to establish the termination of the loop. Notice that we use wp_2 instead $\mathsf{wp}_2^{\mathrm{tc}}$ in the last line of the formulae above, to avoid redundancy on the verification of termination of c, which is already established by Ψ_{φ}. Then, assuming termination of instructions, we can prove total correctness, i.e. for all statements c_1 and c_2, and 2-statement c s.t. $c \triangleright (c_1, c_2)$, and assertions φ and ψ, and memories μ_1 and μ_2 s.t. $\varphi\ \mu_1\ \mu_2$,

$$[\![\varphi]\!] \subseteq [\![\mathsf{wp}_2^{\mathsf{tc}}\ c\ \psi]\!] \quad \Longrightarrow \quad \exists \mu_1', \mu_2'.\ [\![c_1]\!]\ \mu_1\ \mu_1' \wedge [\![c_2]\!]\ \mu_2\ \mu_2' \wedge \psi\ \mu_1'\ \mu_2'$$

Note that the shallow embedding of assertions plays a crucial role here, a partial application of the variant is used as argument for the wp. This would not be possible if we had established a syntax for the formulae through a deep embedding.

3 Verification of the Schorr-Waite algorithm

The Schorr-Waite graph marking algorithm is a widely used case study, see Section 5. Yang [30] uses relational separation logic to prove the equivalence between the Schorr-Waite algorithm and depth-first search, and convincingly argues that the proof in relational separation logic is more elegant and more concise than an earlier functional

verification [29] of the SW algorithm in separation logic. In this section we report on a machine-checked proof of the Schorr-Waite algorithm using the weakest precondition calculus described in the previous section. The structure of the proof is similar to Yang's pen-and-paper proof [30]; one difference is that we prove total correctness rather than co-termination.

Algorithm and relational specification. DFS traverses a binary tree marking every node in a depth-first basis. In order to backtrack the tree traversal, it uses a stack as an auxiliary storage to keep track of the parent nodes that need to be revisited. The Schorr-Waite algorithm optimizes the space needed by DFS by removing the stack. The set of nodes to be revisited are encoded as a transformation on the heap structure: pushing a node in the stack is implemented as an inversion of the left edge that is traversed, removing a node from the stack is defined as restoring the original edge. Figure 6 shows a 2-statement merging the Schorr-Waite algorithm (marked with a gray shadow) with DFS.

Verification. We have used the Coq framework to verify the 2-statement in Figure 6 against the specification:

$$\begin{array}{l} Pre \doteq \mathsf{Same} \wedge c{=}c' \wedge \begin{pmatrix} \mathsf{noDang}\, G \wedge c \in G \cup \{\mathsf{nil}\} \\ \mathsf{noDang}\, G \wedge c' \in G \cup \{\mathsf{nil}\} \end{pmatrix} \\ Post \doteq \mathsf{Same} \end{array}$$

where c and c' represent the corresponding tree roots and G denotes the set of tree nodes. The predicate $\mathsf{noDang}\, G$ states that G is a set of non-dangling pointers closed under heap reachability:

$$\mathsf{noDang}\, G \doteq \forall_\star x \in G.\ \exists lr.\ (x \mapsto l, r, -, -) \wedge l \in G \cup \{\mathsf{nil}\} \wedge r \in G \cup \{\mathsf{nil}\}$$

The additional condition $c \in G$ implies that the set of tree nodes reachable from the root c is a subset of G. The specification states that under initial heaps with the same tree structure with root c, SW and DFS terminate with the same final states.

The application of the wp_2 function to the 2-statement and the postcondition above returns a verification condition that contains an existential quantification for the loop invariant. We have used a slightly modified version of the invariant proposed by Yang [30]:

$$\mathsf{Same} \star \mathsf{uniq}\, \alpha \wedge \mathsf{Stack}\, p\, c\, \alpha \wedge p{=}p' \wedge \begin{pmatrix} \mathsf{noDang}\, G \wedge p{\in}G \wedge c \in G \\ \mathsf{noDang}\, G \wedge p' {\in} G \wedge \alpha \subseteq G \cup \{\mathsf{nil}\} \end{pmatrix}$$

Basically, the invariant establishes that no dangling pointers can be introduced during the algorithms execution, and provides a relation between the auxiliary stack storage used by DFS and its representation in the Schorr-Waite algorithm. This relation is formalized by the predicate Stack:

$$\begin{array}{l} \mathsf{Stack}\, p\, c\, \epsilon \doteq c{=}\mathsf{nil} \\ \mathsf{Stack}\, p\, c\, a{::}\alpha \doteq \exists n_0, x.\ \mathsf{Stack}\, c\, n_0\, \alpha \star c{=}a \wedge \\ \qquad \left[\begin{pmatrix} c \mapsto n_0, x, Marked, Left \\ c \mapsto p, x, Marked, Left \end{pmatrix} \vee \begin{pmatrix} c \mapsto x, n_0, Marked, Right \\ c \mapsto x, p, Marked, Right \end{pmatrix} \right] \end{array}$$

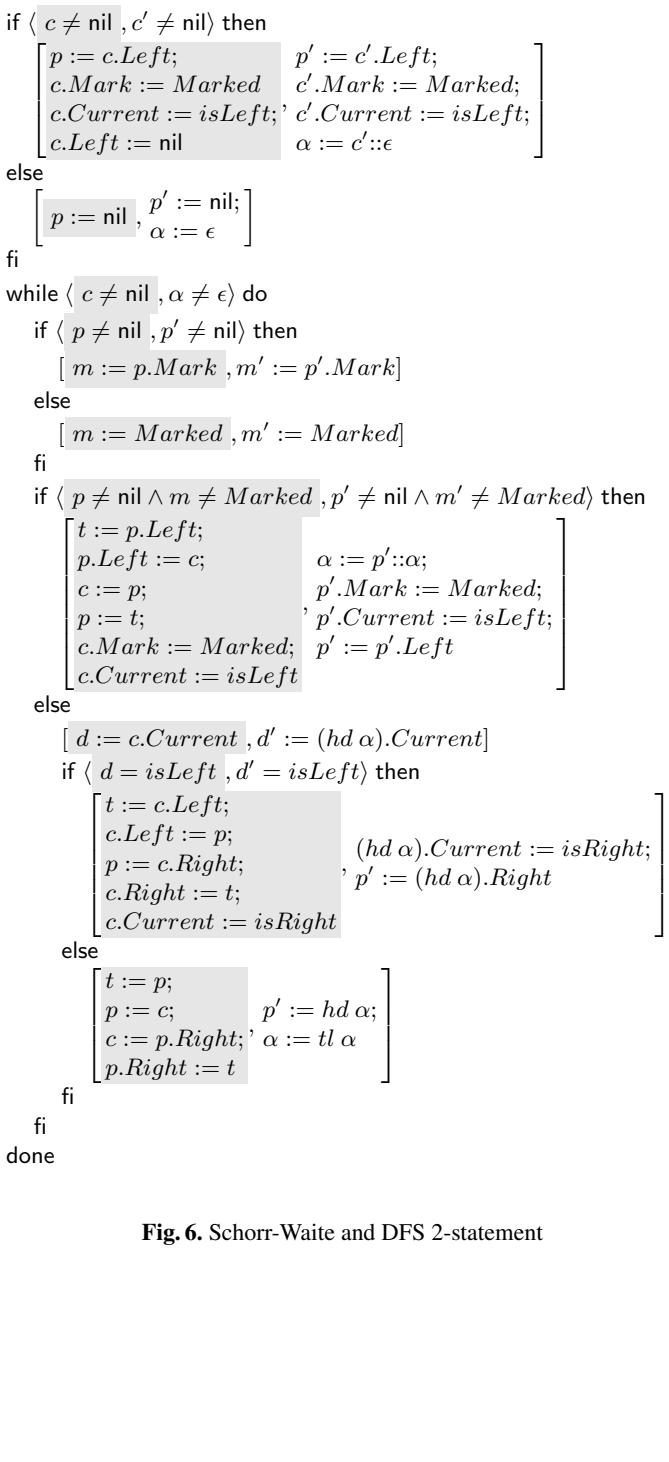

Fig. 6. Schorr-Waite and DFS 2-statement

In particular, when $c \neq \mathsf{nil}$, c is the top element in the stack α and p its left or right child. The remaining stack elements are related inductively. The difference with respect to Yang's invariant consists on the predicate $\mathsf{uniq}\ \alpha$, that states that the list α does not contain repeated elements. The need for this extra condition became evident when discharging the verification conditions in the Coq proof assistant.

Total correctness. We have also developed a total correctness argument for the Schorr-Waite algorithm using the total correctness version of the weakest precondition calculus presented earlier. Then, we extended the proof with the addition of a variant relation, a lexicographic order similar to the one used by Giorgino et al [12]: let (st_1, st_2) and (st'_1, st'_2) be pairs of states, then $\mathsf{var}(st_1, st_2)(st'_1, st'_2)$ iff one of the following holds:

- the number of unmarked nodes in (st_1, st_2) is smaller than the number of unmarked nodes in (st'_1, st'_2),
- the number of unmarked nodes in (st_1, st_2) and (st'_1, st'_2) is the same but the number of nodes in (st_1, st_2) with $Current$ field set to $isLeft$ is smaller than in (st'_1, st'_2), or
- the number of unmarked nodes and the number of nodes of (st_1, st_2) and (st'_1, st'_2) is the same but the size of the stack α in (st_1, st_2) is smaller than in (st'_1, st'_2).

We showed that this is a well-founded order and proved that it holds for the pre and post states of the loop body using the wp_2 calculus. In particular note that of the three ways to construct the order, the first one corresponds to a push, the second one to a swing and the third one to a pop operation.

4 Beyond structurally equivalent programs

A common caveat of syntactic relational methods is the limited support for non structurally equivalent programs. Although this restriction can be circumvented in the setting of relational separation logic by using the embedding rule, the ability to reason in terms of relational loop invariants is still not supported.

In this section, we present a different strategy that cleanly extends the weakest precondition based calculus presented in Section 2 to cope with non structurally equivalent code. We enhance the previous formalism through a preliminary transformation that is performed on the programs to be verified. This syntactic transformations can yield structurally equivalent programs while retaining some semantic properties that ensure that the relational validity on the transformed programs also holds on the original programs.

Let us first make precise the notion of refinement we adopt. We say that c is a refinement of c', noted $c \succcurlyeq c'$ if the following conditions hold for all μ, μ', μ'' and σ:

- if $[\![c']\!]\ \mu\ \mu'$ then $[\![c]\!]\ \mu\ \mu'$;
- if $[\![c']\!]\ \mu\ \mu'$ and $[\![c]\!]\ \mu\ \mu''$ then $\mu' = \mu''$ and
- if c is σ-safe then c' is σ-safe.

We know, under this rather weak definition of refinement, that in order to establish a relational property on two programs c_1 and c_2, it is sufficient to establish such property for any two programs c'_1 and c'_2 s.t. $c_1 \succcurlyeq c'_1$ and $c_2 \succcurlyeq c'_2$:

$$\text{(RO)}\frac{}{\vdash c; d \succcurlyeq d; c}\ \text{if } \mathsf{fv}(c) \cap \mathsf{fv}(d) = \emptyset \qquad \text{(SK)}\frac{}{\vdash c; \mathsf{skip} \succcurlyeq c}$$

$$\text{(IF1)}\frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \succcurlyeq \mathsf{assert}(b); c_1} \qquad \text{(IF2)}\frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \succcurlyeq \mathsf{assert}(\neg b); c_2}$$

$$\text{(WHU)}\frac{}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(b); c; \mathsf{while}\ b\ \mathsf{do}\ c}$$

$$\text{(WHS)}\frac{}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(b); c; \mathsf{assert}(\neg b)}$$

$$\text{(IFM)}\frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c\ ;\ \mathsf{if}\ b'\ \mathsf{then}\ c'\ \succcurlyeq \mathsf{assert}(b \Leftrightarrow b'); \mathsf{if}\ b\ \mathsf{then}\ c; \mathsf{assert}(b'); c}$$

$$\text{(LRS)}\frac{}{\vdash \mathsf{for}\ i{=}m\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(m \le n' \le n); \mathsf{for}\ i{=}m\ \mathsf{to}\ n'\ \mathsf{by}\ k\ \mathsf{do}\ c; \mathsf{for}\ j{=}i\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ c[j/i]}$$

$$\text{(LT)}\frac{}{\vdash \mathsf{for}\ i{=}0\ \mathsf{to}\ n\ \mathsf{by}\ 1\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(n \bmod k = 0); \mathsf{for}\ i{=}0\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ (\mathsf{for}\ j{=}0\ \mathsf{to}\ k\ \mathsf{by}\ 1\ \mathsf{do}\ c[i{+}j/i])}$$

$$\text{(R-RI)}\frac{}{\vdash \mathsf{for}\ i{=}m\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(\left\lceil \frac{n-m}{k} \right\rceil = \left\lceil \frac{n'-m'}{k} \right\rceil); \mathsf{for}\ i{=}m'\ \mathsf{to}\ n'\ \mathsf{by}\ k\ \mathsf{do}\ c[(i{-}m'{+}m)/i]}$$

Fig. 7. Syntactic refinement rules (excerpt).

Lemma 1. *For all programs c_1 and c_2, and c_1', c_2' such that $c_1 \succcurlyeq c_1'$ and $c_2 \succcurlyeq c_2'$, if $\vDash \{\varphi\}\ c_1' \sim c_2'\ \{\psi\}$ then $\vDash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$, provided c_1' and c_2' are φ-safe.*

Figure 7 provides a set of syntactic rules deriving a refinement relation. For clarity, we introduce the statement $\mathsf{for}\ i = m\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ c$ as a syntax sugar for statement $i := m; \mathsf{while}\ i < n\ \mathsf{do}\ c; i := i + k$. As can be seen in the figure, the rules consist of basic structure transformations. The most complex rules are perhaps (LRS) and (LT), which perform loop range splitting and loop tiling, respectively. The set of refinement rules in Figure 7 is sound, i.e., it induces a refinement relation:

Lemma 2. *For all statements c and c', if $\vdash c \succcurlyeq c'$ then $c \succcurlyeq c'$.*

Example: vectorization of sum. Figure 8 presents a simple algorithm that computes the sum of the values of the node elements in a singly linked list. A program vectorization consists on relying on special purpose *SIMD* (single instruction, multiple data) instructions, taking advantage of the associativity and commutativity of the arithmetic computation performed in a program loop. Intuitively, for this particular example the vectorization consists in grouping the loop iterations in chunks of 4 iterations, and performing 4 addition operations simultaneously with the `_mm_add_epi32` instruction. Figure 9 shows the vectorized algorithm. Let n denote the length of the linked list pointed by `head`. The first loop iterates $n \div 4$ times and computes the summation of the first $4 \times (n \div 4)$ elements of the linked list, storing it in the 128-bits vector `sum`. The second loop computes the summation of the remaining $n \bmod 4$ elements and stores it

$$\frac{sum\,(list*\ head,\ int\ size)}{\begin{array}{l} rest := 0; \\ \mathsf{for}\ i = 0\ \mathsf{to}\ size\ \mathsf{by}\ 1\ \mathsf{do} \\ \quad rest := rest + head.val; \\ \quad head := head.next; \end{array}}$$

Fig. 8. Original version of `sum` algorithm.

in variable `rest`. The final value is computed by adding to the variable `rest` the partial results stored in the bit vector `sum`.

By applying a sequence of refinement steps over the original program one can obtain a pair of structurally similar programs. Then, providing a relational invariant becomes much simpler that verifying each of the programs functionally. Indeed, assume that the predicate $\mathsf{EqList}(\mathtt{head}, \mathtt{head}', \mathtt{size})$ holds as precondition, with inductive predicate EqList is defined by the following clauses:

$$\begin{array}{ll} \mathsf{EqList}(l_1, l_2, 0) & \doteq l_1 = l_2 = \mathsf{null} \\ \mathsf{EqList}(l_1, l_2, n{+}1) & \doteq \mathsf{EqList}(l_1', l_2', n) \wedge \exists v, l_1', l_2'.\ l_1 \mapsto (v, l_1') \wedge l_2 \mapsto (v, l_2') \end{array}$$

Then, in order to verify that original and vectorized algorithms compute the same value, i.e., that $rest = rest'$ holds as a relational postcondition, it is sufficient to establish the validity of loop invariants of the form:

$$sum[0]{+}sum[1]{+}sum[2]{+}sum[3] = rest$$

and

$$rest'{+}sum[0]{+}sum[1]{+}sum[2]{+}sum[3] = rest$$

Notice that these relational loop invariants are much simpler that those required in a functional verification of the algorithm.

5 Related work

Relational methods and program verification techniques are intimately connected since their origins. In particular, methods based on program refinement, program equivalence, and logical relations have been used widely to reason about program correctness. In this respect, it is perhaps surprising that relational program logics have only been introduced recently. Benton [7] develops a relational Hoare logic for a small imperative language and shows how program optimizations can be validated using relational reasoning. Other relational logics include Yang's relational separation logic [30] and Barthe, Grégoire and Zanella's probabilistic relational Hoare logic [5]. More recently, Nanevski, Banerjee and Garg developed a relational separation logic for Hoare type theory [23]. It extends Yang's logic to a richer programming and specification language, and is tailored for reasoning information flow; the logic is formalized in the Coq proof

```
ssesum (list* head', int size)
  sum = _mm_set1_epi32(0);
  for i = 0 to size − 3 by 4 do
    curr := _mm_insert_epi32(curr, head'.val, 0);
    head' := head'.next;
    curr := _mm_insert_epi32(curr, head'.val, 1);
    head' := head'.next;
    curr := _mm_insert_epi32(curr, head'.val, 2);
    head' := head'.next;
    curr := _mm_insert_epi32(curr, head'.val, 3);
    head' := head'.next;
    sum := _mm_add_epi32(sum, curr);
  rest' := 0;
  for j = i to size by 1 do
    rest' := rest'+head'.val;
    head' := head'.next;
  rest' := rest'+_mm_extract_epi32(sum, 0)+_mm_extract_epi32(sum, 1)+
    _mm_extract_epi32(sum, 2) + _mm_extract_epi32(sum, 3);
```

Fig. 9. SSE optimized version of `sum` algorithm.

assistant; in contrast to our formalization, it uses a shallow embedding of programs. Independently, Beringer [8] provided a reconstruction of relational separation logic based on a notion of decomposition that allows reducing relational program logics to standard program logics; the soundness of the logic is formalized in the Isabelle proof assistant. In addition to these general-purpose logics, specialized relational logics have been developed to reason about specific properties, and especially information flow [1].

On the formalization side, there have been many machine-checked accounts of separation logic in proof assistants, e.g. [3, 27], including some frameworks designed to support automated reasoning in separation logic [2, 11, 19, 21]. Moreover, the Schorr-Waite algorithm is a classical example in program verification, and has been verified formally using a variety of tools and techniques. Suzuki [26] provides an early machine-supported proof of the Schorr-Waite algorithm using an automated verifier for pointer programs. More recently, Bornat [10] provides a machine-checked proof of the algorithm in the Jape proof assistant. Subsequently, Mehta and Nipkow [20], Hubert and Marché [15], Bubel [6], Jacobs and Piessens [17] formalize the algorithm in Isabelle, Caduceus, KeY and VeriFast respectively. More recently, Giorgino et al [12] prove the correctness and termination of the algorithm in Isabelle, using refinement. All these formalizations use standard program logics.

6 Conclusion

Relational separation logic is a powerful tool devised for reasoning about the relation between heap manipulating programs. To the best of our knowledge, we have formalized in the Coq proof assistant the first certified weakest precondition calculus for rela-

tional separation logic. We illustrated its usefulness and scalability by proving a challenging case study: the correctness of the Schorr-Waite graph marking algorithm.

The Coq development has been done using ssreflect library which greatly improves the conciseness of the proofs. For example, the relational weakest precondition, soundness proofs, the definition and specification and proof of the Schorr-Waite graph marking algorithm and Depth First Search take 1586 lines of specification and 3538 lines of proofs. We believe that the formalization of the verification setting and the formal proof of the algorithms poses no significant overhead over hand-written proofs as overviewed by Yang [30].

In the future, it would be interesting to formalize the modular proof of the algorithm reported in [9] and to prove the equivalence between different implementations of ADTs; for the latter, we believe that the extensions to non-structurally equivalent code will prove crucial. Another line of work is to extend our formalization to reason about concurrent separation logic [28] and verify the correctness of lock-free algorithms [14].

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In G. Morrisett and S. Peyton Jones, editors, *Principles of Programming Languages*, pages 91–102. ACM, 2006.
2. A. Appel. Tactics for separation logic. Unpublished manuscript (`http://www.cs.princeton.edu/~appel/papers/septacs.pdf`), January 2006.
3. Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher-Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.
4. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Formal Methods*, Lecture Notes in Computer Science. Springer, 2011.
5. G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages*, pages 90–101. ACM Press, 2009.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
7. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Principles of Programming Languages*, pages 14–25. ACM Press, 2004.
8. L. Beringer. Relational program logics in decomposed style, 2010. Submitted.
9. R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *Principles of Programming Languages*, pages 339–352, 2010.
10. R. Bornat. Proving pointer programs in hoare logic. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.
11. H. Gast. Lightweight separation. In O. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2008.
12. M. Giorgino, M. Strecker, R. Matthes, and M. Pantel. Verification of the Schorr-Waite algorithm - From trees to graphs, January 2010.

13. G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.
14. A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages*, pages 16–28. ACM, 2009.
15. T. Hubert and C. Marché. A case study of c source code verification: the schorr-waite algorithm. In B. Aichernig and B. Beckert, editors, *Software Engineering and Formal Methods*, pages 190–199. IEEE Computer Society, 2005.
16. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26, 2001.
17. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, 2008.
18. B. Jacobs, J. Smans, and F. Piessens. Verifying the composite pattern using separation logic. In *Workshop on Specification and Verification of Component-Based Systems, Challenge Problem Track*, November 2008.
19. A. McCreight. Practical tactics for separation logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2009.
20. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1-2):200–227, 2005.
21. M. O. Myreen. Separation logic adapted for proofs by rewriting. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 485–489. Springer, 2010.
22. A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In M. Hermenegildo and J. Palsberg, editors, *Principles of Programming Languages*, pages 261–274. ACM, 2010.
23. Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *2011 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2011. To appear.
24. P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001. Invited paper.
25. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
26. N. Suzuki. *Automatic Verification of Programs with Complex Data Structures.* PhD thesis, Stanford University, 1976.
27. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Principles of Programming Languages*, pages 97–108. ACM, 2007.
28. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *Conference on Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer-Verlag, 2007.
29. H Yang. *Local reasoning for stateful programs*. PhD thesis, University of Illinois, Urbana, IL, USA, 2001.
30. H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.
31. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.