Relational Separation Logic

Hongseok Yang

Department of Computer Science, Queen Mary, University of London, Mile End Road, London, UK

To John C. Reynolds for his 70th birthday.

Abstract

In this paper, we present a Hoare-style logic for specifying and verifying how two pointer programs are related. Our logic lifts the main features of separation logic, from an assertion to a relation, and from a property about a single program to a relationship between two programs. We show the strength of the logic, by proving that the Schorr-Waite graph marking algorithm is equivalent to the depth-first traversal.

Key words: Separation Logic, Program Verification, Relational Reasoning, Schorr-Waite Graph Marking Algorithm

1 Introduction

Finding a proper verification formalism for pointers is a long-standing problem in program verification research. The main challenge is to develop an effective formalism, which does not produce an unnecessarily complicated proof. Ideally, a formal proof about pointers should be only as complicated as an informal correctness argument.

Recently, with Reynolds, O'Hearn and others, we have developed separation logic to attack this problem [1–4]. The main feature of separation logic is that

* This work was done while the author was associated with Seoul National University. It was supported by grant No. R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

Email address: hyang@dcs.qmul.ac.uk (Hongseok Yang).
URL: www.dcs.qmul.ac.uk/~hyang (Hongseok Yang).

the logic supports *local reasoning*, a style of formal reasoning where specification and verification of a program focus on what the program accesses; since the accessed portion of memory is usually much smaller than the whole storage, we argued, local reasoning can simplify formal verification. Indeed, local reasoning in separation logic led to relatively simple proofs for several programs [5–7], including the Schorr-Waite graph marking algorithm [8].

The work in this paper is motivated by Uday Reddy's remark on our proof of the Schorr-Waite algorithm in separation logic [8]. When seeing our proof, Reddy remarked that our specification did not match up with a common understanding of the algorithm. The algorithm is usually understood as an optimized version of the depth-first traversal. Thus, the equivalence between the algorithm and the depth-first traversal is what a verification should focus on. However, our specification expresses full correctness of the algorithm, and thus the specification as well as the verification became unnecessarily complicated.

In fact, it is common to specify a program by its relationship with another program. When a compiler optimizes an input program, the optimized program and the original program must be equivalent [9]. Another example is a client of an abstract data type which has two different implementations. In this case, we often want to specify that a client is insensitive to the choice of the implementation; the client with one implementation is (observationally) equivalent to the client with the other implementation [10–13].

Unfortunately, separation logic is not appropriate for reasoning about such specifications. Separation logic does not provide a language for specifying how two programs are related, not to mention proof rules for such a specification. Specifications in the logic are given by a Hoare triple $\{P\}C\{Q\}$, which is good for specifying the input and output relation of a single command, but not for the equivalence between two programs.

In this paper, we propose a Hoare-style logic for specifying and verifying how two *pointer* programs are related. We call this logic *relational separation logic*, because first, most features of our logic are the liftings of the corresponding features of separation logic, and second, our logic includes separation logic: it contains all the proof rules for total-correctness triples in separation logic, and provides a new rule that takes a pair of such Hoare triples, and concludes a relationship between programs. Our logic is based on separation logic in this way, in order to be effective for reasoning about pointers.

The central ideas of our logic are Hoare quadruples and separating conjunction $*$ for *relations*. A Hoare quadruple $\{R\}^{C_1}_{C_2}\{S\}$ consists of relations R and S between states, and commands C_1 and C_2. Intuitively, it means that when C_1 and C_2 are started at R-related states, they end in S-related states, and, moreover, during execution, both C_1 and C_2 access only those memory cells that

the "pre-relation" R guarantees to exist. The common examples of quadruple specifications include the equivalence of two programs where R and S are the equality relation, and the simulation between two programs where R and S denote the same simulation relation. However, R and S are not restricted to only these two cases. For instance, when a quadruple is derived directly from a pair of Hoare triples, the pre- and post-relations of the quadruple are usually different from the equality relation, and they are also different from each other. Note that a quadruple constrains the accessible memory cells to those "mentioned" in the pre-relation R. Because of this constraint, the logic has a proof rule, called the frame rule, that supports local reasoning for quadruples.

The separating conjunction $R * S$ of relation R and S relates two states, when each state can be split into two parts, so that the first parts of these two splittings are related by R and the second parts by S. This $*$ connective allows a "smooth" inclusion of separation logic. When a quadruple is derived from a pair of Hoare triples, the pre- and post-relations of the quadruple usually contain assertions with separating connectives. The $*$ connective for relations facilitates the manipulations of these separating connectives. The $*$ connective also plays an essential role in the formulation of the frame rule for quadruples.

We start the paper by explaining the setup; we present our storage model in Section 2, and the language for expressions, assertions, and commands in Section 3. Then, we develop relational separation logic. We first define a language for relations in Section 4. Using this relation language, we define the syntax and semantics of Hoare quadruples, illustrate their use for specifying a relationship between programs, and provide proof rules for deriving correct quadruples in Section 5. In Section 6, we test the effectiveness of our logic by specifying and proving that the Schorr-Waite graph traversal algorithm is equivalent to the depth-first traversal. In Section 7, we discuss a problem with a nondeterministic allocator; when programs use nondeterministic memory allocator, a naive interpretation of a quadruple leads to the unsoundness of the frame rule for quadruples. After discussing the problem, we present a solution. Finally, we discuss related work and explain the shortcomings of our logic with possible solutions in Section 8.

2 Storage Model

In this paper, we make two assumptions about storage: first, a store holds not only integer values, but also set and list values; second, pointers are simply positive natural numbers, so that arithmetic operations can be applied to pointers. In this section, we will make clear these assumptions by defining semantic domains for storage.

To model three different kinds of variables, we assume countably infinite, mutually disjoint sets $\mathsf{IntVars}$, $\mathsf{SetVars}$, and $\mathsf{ListVars}$ of variables. Variables x, y in $\mathsf{IntVars}$ denote integers, and X, Y in $\mathsf{SetVars}$ finite sets of integers, and α, β in $\mathsf{ListVars}$ finite sequences of integers. The value of each variable is given by a *store*:

$$\begin{aligned}\mathsf{Stores} \stackrel{\text{def}}{=} \{s : (\mathsf{IntVars} \cup \mathsf{SetVars} \cup \mathsf{ListVars}) \to (\mathsf{Ints} \cup \mathcal{P}_{\text{fin}}(\mathsf{Ints}) \cup \mathsf{Ints}^*) \\ \mid \forall x X \alpha.\ s(x) \in \mathsf{Ints} \ \wedge\ s(X) \in \mathcal{P}_{\text{fin}}(\mathsf{Ints}) \ \wedge\ s(\alpha) \in \mathsf{Ints}^*\}\end{aligned}$$

A state in our model consists of the store and heap components. While the store component gives the values of variables, the heap component determines which cells are currently allocated, and what their contents are. The formal definitions of heaps and states are given below:

$$\begin{aligned}\mathsf{PosNats} &\stackrel{\text{def}}{=} \{1, 2, \ldots\} \\ \mathsf{Heaps} &\stackrel{\text{def}}{=} \mathsf{PosNats} \rightharpoonup_{\text{fin}} \mathsf{Ints} \\ \mathsf{States} &\stackrel{\text{def}}{=} \mathsf{Stores} \times \mathsf{Heaps}\end{aligned}$$

Note that the pointer arithmetic is allowed in this model, because the addresses of heap cells are simply positive natural numbers.

In this paper, we frequently use a heap-disjointness predicate $\#$ and a heap-combining operator $*$. Let h_1 and h_2 be heaps.

$$\begin{aligned}h_1 \# h_2 &\stackrel{\text{def}}{\iff} \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset \\ h_1 * h_2 &\stackrel{\text{def}}{=} \mathsf{if}\ (h_1 \# h_2)\ \mathsf{then}\ (h_1 \cup h_2)\ \mathsf{else}\ \text{undefined}\end{aligned}$$

The predicate $h_1 \# h_2$ means that h_1 and h_2 do not share any active cells, so they describe the disjoint areas of heap storage. When heaps h_1 and h_2 are such disjoint heaps, $h_1 * h_2$ denotes the combined heap of h_1 and h_2; in the other case, $h_1 * h_2$ is not defined.

3 Language for Assertions and Programs

We use a version of the language for assertions and programs in separation logic [7,3], where variables can contain not just integers, but also finite sets and finite lists. The syntax of our language is shown in Figure 1.

The language has four different types of expressions: integers, booleans, finite sets, and finite lists. All these expressions denote heap-independent state

$$
\begin{array}{lll}
\text{Integer Expressions} & E & ::= x \mid 0 \mid 1 \mid E + E \mid E \times E \mid E - E \mid \mathtt{hd}\ L \\
\text{Boolean Expressions} & B & ::= \mathsf{false} \mid B \Rightarrow B \mid E = E \mid E < E \mid E \in G \mid G \subseteq G \\
\text{Set Expressions} & G & ::= X \mid G \cup G \mid \{E, \ldots, E\} \mid \mathsf{toSet}(L) \\
\text{List Expressions} & L & ::= \alpha \mid \epsilon \mid E{::}L \mid \mathtt{tl}\ L \\
\text{Commands} & C & ::= x := \mathtt{cons}(E, \ldots, E) \mid x := [E] \mid [E] := E \\
 & & \quad \mid \ \mathtt{free}(E) \mid x := E \mid X := G \mid \alpha := L \\
 & & \quad \mid \ C; C \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od} \\
\text{Assertions} & P & ::= B \mid \mathsf{false} \mid P \Rightarrow Q \mid \forall x.P \mid \forall X.P \mid \forall \alpha.P \\
 & & \quad \mid \ E \mapsto E \mid \mathsf{emp} \mid P * Q \mid \forall_* x \in G.P
\end{array}
$$

Fig. 1. Syntax of Expressions, Commands and Assertions

readers. That is, they mean mappings from stores, not states, to values of appropriate types. The reason for considering only heap-independent expressions is that such expressions allow simpler proof rules. For instance, if an expression E reads the heap, then assertion $P \wedge x = E$ does not imply $P[E/x]$ in general; on the other hand, the implication always holds if E does not access the heap. [1] The semantics of most of the expressions is standard. The only exceptions are the meaning of `hd` and `tl` for the "error" cases.

$$
[\![\mathtt{hd}\ L]\!]s = \begin{cases} i \text{ if } [\![L]\!]s = i{:}a \\ 0 \text{ otherwise} \end{cases} \qquad [\![\mathtt{tl}\ L]\!]s = \begin{cases} a \text{ if } [\![L]\!]s = i{:}a \\ \epsilon \text{ otherwise} \end{cases}
$$

Here we choose 0 and the empty list ϵ for the error cases of `hd` and `tl`, respectively. Choosing different values for the error cases will not affect any of our examples in this paper, because we will always make sure that these error cases never arise in the examples.

The assertions include all the boolean expressions, the usual connectives from classical logic, the points-to relation $E \mapsto E'$, and separating connectives emp, $*$ and $\forall_*$. [2] The meaning of these assertions is given by a forcing relation $\models$, and appears in Figure 2. We note the clauses for $E \mapsto E'$, separating conjunction $*$ and its iterated version $\forall_*$. The points-to predicate $E \mapsto E'$ means that the heap has a cell E, the content of this cell is E', and the cell E

[1] Reynolds gave a counterexample, which is similar to $((x = 1) * (x = 1) \wedge x = [y])$. This assertion is satisfiable, but $([y] = 1) * ([y] = 1)$ is not satisfiable.

[2] The assertion language in separation logic also includes separating implication $-\!\!*$. We will not consider $-\!\!*$ in this paper because it does not raise any new issues.

For a state (s,h) in States,

$$
\begin{array}{lll}
(s,h) \models B & \text{iff} & [\![B]\!]s = \mathit{true} \\
(s,h) \models \mathsf{false} & & \text{never} \\
(s,h) \models P \Rightarrow Q & \text{iff} & \text{if } (s,h) \models P, \text{ then } (s,h) \models Q \\
(s,h) \models \forall x.\, P & \text{iff} & \text{for all } i \in \mathsf{Ints},\ (s[x \mapsto i], h) \models P \\
(s,h) \models \forall X.\, P & \text{iff} & \text{for all } V \in \mathcal{P}_{\mathrm{fin}}(\mathsf{Ints}),\ (s[X \mapsto V], h) \models P \\
(s,h) \models \forall \alpha.\, P & \text{iff} & \text{for all } a \in \mathsf{Ints}^*,\ (s[\alpha \mapsto a], h) \models P \\
(s,h) \models E \mapsto E' & \text{iff} & \mathsf{dom}(h) = \{[\![E]\!]s\},\ \text{and } h([\![E]\!]s) = ([\![E']\!]s) \\
(s,h) \models \mathsf{emp} & \text{iff} & \mathsf{dom}(h) = \emptyset \\
(s,h) \models P * Q & \text{iff} & \text{there exist } h_0, h_1 \text{ s.t.} \\
 & & \qquad h_0 * h_1 = h,\ (s,h_0) \models P,\ \text{and } (s,h_1) \models Q \\
(s,h) \models \forall_* x \in G.\, P & \text{iff} & \text{if } [\![G]\!]s \neq \emptyset \wedge [\![G]\!]s = \{i_1, .., i_n\} \text{ for some } i_1, .., i_n \in \mathsf{Ints} \\
 & & \text{then } (s,h) \models P[i_1/x] * \ldots * P[i_n/x] \\
 & & \text{else } (s,h) \models \mathsf{emp}
\end{array}
$$

Fig. 2. Semantics of Assertions

is the only active cell in the heap. The other cases, $*$ and $\forall_*$, express properties about heap splitting. The separating conjunction $P * Q$ says that the current heap can be split into two disjoint subheaps such that P holds for the one subheap, and Q holds for the other. This binary splitting is generalized in the iterated separating conjunction $\forall_*$. For a state (s,h), the iterated separating conjunction $\forall_* x \in G.\, P$ means that the current heap h is split into $|[\![G]\!]s|$-many subheaps, so that $P[i/x]$ holds for the i-th subheap. Note that we can express all the other classical connectives using the connectives in Figure 2: $\neg P \stackrel{\text{def}}{=} P \Rightarrow \mathsf{false}$, $\mathsf{true} \stackrel{\text{def}}{=} \neg\mathsf{false}$, $P \vee Q \stackrel{\text{def}}{=} \neg P \Rightarrow Q$, $P \wedge Q \stackrel{\text{def}}{=} \neg(\neg P \vee \neg Q)$, $\exists x.\, P \stackrel{\text{def}}{=} \neg\forall x.\, \neg P$, $\exists X.\, P \stackrel{\text{def}}{=} \neg\forall X.\, \neg P$, and $\exists \alpha.\, P \stackrel{\text{def}}{=} \neg\forall \alpha.\, \neg P$.

We will often use the following abbreviations for assertions:

$$
\begin{array}{rl}
E \mapsto \text{-} \stackrel{\text{def}}{=} \exists y.\, E \mapsto y & \qquad (\text{where } y \notin \mathsf{FV}(E)) \\
E \mapsto E_1, \ldots, E_n \stackrel{\text{def}}{=} (E \mapsto E_1) * \ldots * (E + n - 1 \mapsto E_n) & \\
E \doteq E' \stackrel{\text{def}}{=} E = E' \ \wedge\ \mathsf{emp} &
\end{array}
$$

The first abbreviation means that cell E is the only active cell in heap storage, and the second means that the n consecutive cells $E, \ldots, E + n - 1$ are the

only active cells in the heap, and the contents of these cells are $E_1, \ldots, E_n$. The last abbreviation $E \doteq E'$ expresses the equality of E and E' without using any heap cells: E must be equal to E', and the heap must be empty. This last abbreviation $E \doteq E'$ will frequently be used in our proof, especially when we need to transform normal conjunction to separating conjunction. We does such a transformation using the following equivalence:

$$P \ \wedge \ (E = E') \iff P * (E \doteq E').$$

The language for commands is a simple imperative language extended with heap operations, and set and list variables. It has four heap operations $x := \texttt{cons}(E_1, \ldots, E_n)$, $x := [E]$, $[E] := E'$, and $\texttt{free}(E)$. The command $x := \texttt{cons}(E_1, \ldots, E_n)$ allocates n consecutive cells, initializes them by $E_1 \ldots E_n$, and stores the address of the first cell in variable x. The command is nondeterministic, because it chooses *any* n consecutive inactive cells. The content of a heap cell E is read by $x := [E]$, and updated to E' by $[E] := E'$. A cell E is freed when $\texttt{free}(E)$ is called.

The commands can have set or list variables. They can contain assignments to such variables, and include a conditional statement whose boolean condition has set or list expressions. These set or list variables are mainly used for verification purposes, especially for writing an "abstract" program that serves as a specification for a "concrete" program.

We interpret commands using a small-step operational semantics. A *configuration* is a triple consisting of a store, a heap, and a command. The semantics defines a relation $\rightsquigarrow$ from configurations to configurations, states or `fault`. When $(s, h, C) \rightsquigarrow (s_1, h_1, C_1)$, it means that the state is transformed from (s, h) to (s_1, h_1) by C, and the remaining computation is C_1. When C_1 is omitted, the transition means that this computation of C has been finished. The faulting result, `fault`, indicates that C tries to dereference a dangling pointer. The full operational semantics is in Appendix A.

An important property of this semantics is that all commands satisfy locality properties. Let's say that configuration (s, h, C) is *safe* when $(s, h, C) \not\rightsquigarrow^* \texttt{fault}$, and that (s, h, C) *can diverge* when there is an infinite path from (s, h, C) following the relation $\rightsquigarrow$. Intuitively, the safety of (s, h, C) means that h contains all the cells that C will only access, except newly allocated cells. The locality properties of a command are given below [14]:

(1) Safety monotonicity: for all states (s, h) and heaps h_0, if $h \# h_0$ and (s, h, C) is safe, then $(s, h * h_0, C)$ is safe.
(2) Termination monotonicity: for all states (s, h) and heaps h_0, if $h \# h_0$, (s, h, C) is safe, and (s, h, C) cannot diverge, then $(s, h * h_0, C)$ cannot diverge.

(3) Frame property: for all states (s, h) and heaps h_0, if $h \# h_0$, (s, h, C) is safe, and $(s, h * h_0, C) \rightsquigarrow^* (s_1, h_1)$, then there exists a heap h_2 such that

$$h_2 \# h_0, \quad h_1 = h_2 * h_0, \quad \text{and} \quad (s, h, C) \rightsquigarrow^* (s_1, h_2).$$

Intuitively, these properties say that when C only accesses cells in h, every computation of C from a bigger heap $h * h_0$ can be tracked to some computation from h.

In this paper, we use the total-correctness Hoare triples in separation logic. In separation logic, the interpretation of a total-correctness triple $[P]C[Q]$ has an additional requirement about memory access; in addition to the usual total-correctness requirement, the triple asks that C should access only those cells "mentioned" in P, except newly allocated ones. More precisely, $[P]C[Q]$ holds if and only if for all states (s, h) satisfying the precondition P (i.e., $(s, h) \models P$),

(1) (s, h, C) is safe,
(2) (s, h, C) cannot diverge, and
(3) if $(s, h, C) \rightsquigarrow^* (s_1, h_1)$, then the final state (s_1, h_1) satisfies Q.

In the definition, the first additional condition on safety prevents C from accessing cells not mentioned in P.

4 Relation Language

The goal of this work is to obtain a good language for specifying how two pointer programs are related, and effective rules for proving such specifications. In this section, we present a first step for reaching this goal; we design a language for relations between states, by extending the main features of the assertion language.

Intuitively, each term R in the relation language considers two computers that use disjoint sets of variables, and expresses how the states of the computers are related. Formally, we interpret R as a set of triples consisting of a store and two heaps, and write $(s, h, h') \models R$ to mean that (s, h, h') belongs to this set. Note that this interpretation exploits the assumption that the two computers use disjoint sets of variables. It represents the stores of both computers by a single s, so that each triple (s, h, h') in R denotes related states of the computers. The semantics of R can alternatively be read as a relation between heaps parameterized by a store: given a store for both computers, R relates the heaps of the computers. We will use this alternative reading of R when we focus on heap storage.

Figure 3 shows the syntax and semantics of the relation language. The lan-

$$
\begin{array}{ll}
(s,h,h') \models \mathsf{Same} & \text{iff } h = h' \\
(s,h,h') \models \mathsf{Emp} & \text{iff } h = h' = [] \\
(s,h,h') \models \begin{pmatrix} P \\ P' \end{pmatrix} & \text{iff } (s,h) \models P \text{ and } (s,h') \models P' \\
(s,h,h') \models R_1 * R_2 & \text{iff there exist } h_1, h_2, h_1', h_2' \text{ s.t. } h = h_1 * h_2,\ h' = h_1' * h_2', \\
 & \qquad (s,h_1,h_1') \models R_1, \ \text{ and } \ (s,h_2,h_2') \models R_2 \\
(s,h,h') \models B & \text{iff } [\![B]\!]s = \mathit{true} \\
(s,h,h') \models \mathsf{False} & \text{Never} \\
(s,h,h') \models R_1 \Rightarrow R_2 & \text{iff if } (s,h,h') \models R_1, \ \text{ then } (s,h,h') \models R_2 \\
(s,h,h') \models \forall x.\ R & \text{iff for all } v \in \mathsf{Ints}, \ (s[x \mapsto v], h, h') \models R \\
(s,h,h') \models \forall X.\ R & \text{iff for all } V \in \mathcal{P}_{\mathrm{fin}}(\mathsf{Ints}), \ (s[x \mapsto V], h, h') \models R \\
(s,h,h') \models \forall \alpha.\ R & \text{iff for all } F \in \mathsf{Ints}^*, \ (s[x \mapsto F], h, h') \models R
\end{array}
$$

Fig. 3. Relation Language

guage has three new atomic formulas Same, Emp and $\binom{P}{P'}$. Given a store, the first two atomic formulas ignore this store; the diagonal relation Same, then, relates identical heaps, and Emp relates empty heaps. The last atomic formula $\binom{P}{P'}$, called *assertion pair*, constructs a relation from assertions P and P'. The constructed relation means the cartesian product of P and P': for each store s, it relates all the heaps h and h' if (s,h) and (s,h') satisfy P and P', respectively.

The main feature of the relation language is the separating conjunction $R{*}S$ of relations. Given the values s of variables, separating conjunction $R * S$ relates heaps h and h' if and only if each heap can be split into two subheaps, i.e., $h = h_1 * h_2$ and $h' = h_1' * h_2'$, so that the first parts h_1 and h_1' of these splittings are related by R, and the second parts h_2 and h_2' by S. For instance,

$$
\mathsf{Same} * \begin{pmatrix} 1 \mapsto 2 \\ 1 \mapsto -2 \end{pmatrix}
$$

relates heaps h and h' if and only if the heaps are the same except the content of cell 1: in h, the cell contains 2, and in h', it contains -2.

The remaining constructs in the relation language are the usual connectives

from classical logic.[3] These classical connectives have the standard meaning, and as in the assertion language, they can express all the missing connectives of classical logic: $\neg R \stackrel{\text{def}}{=} R \Rightarrow \mathsf{False}$, $\mathsf{True} \stackrel{\text{def}}{=} \neg\mathsf{False}$, $R \vee S \stackrel{\text{def}}{=} \neg R \Rightarrow S$, $R \wedge S \stackrel{\text{def}}{=} \neg(\neg R \vee \neg S)$, $\exists x.\, R \stackrel{\text{def}}{=} \neg\forall x.\, \neg R$, $\exists X.\, R \stackrel{\text{def}}{=} \neg\forall X.\, \neg R$, and $\exists \alpha.\, R \stackrel{\text{def}}{=} \neg\forall \alpha.\, \neg R$.

We use an abbreviation $E \doteq E' \;\stackrel{\text{def}}{=}\; E = E' \;\wedge\; \mathsf{Emp}$. Here we put two dots above the equality symbol in order to distinguish this abbreviation from the assertion $E \doteq E'$.

Example 1 The first example is an assertion pair, and shows that variables can be used to link locations in two heaps:

$$\begin{pmatrix} x \mapsto \text{-} \\ x+1 \mapsto \text{-} \end{pmatrix}$$

This relation relates h and h', if heap h is a singleton heap with cell x, and similarly, h' a singleton heap with cell $x+1$. Note that variable x is used here to say that the address of the cell in h is one smaller than that of the cell in h'. $\square$

Example 2 The second example is an instance of a common use of the $*$ connective. In this usage, the $*$ connective splits out the common parts of heaps h and h', and allows us to focus on the difference between h and h'.

$$\mathsf{Same} * \begin{pmatrix} x_0 \mapsto x_1 * x_1 \mapsto x_2 * x_2 \mapsto \mathtt{nil} \\ x_0 \mapsto \mathtt{nil} * x_1 \mapsto x_0 * x_2 \mapsto x_1 \end{pmatrix}$$

This relation relates h and h' if h and h' are identical except the three cells x_0, x_1 and x_2: in h, these three cells form a linked list starting from x_0 and ending with x_2; and in h', the list is reversed, so that the three cells form a linked list starting from x_2 and ending with x_0. $\square$

The proof system for relations consists of the four groups of proof rules: the usual proof rules from classical logic, the rules from the logic of Bunched Implications [15], the rules about assertion pairs, and the rules specific to relational separation logic. Figure 4 shows the most-frequently used proof rules for relations, except the ones from classical logic. The first seven rules in the figure come from the logic of Bunched Implications. They express that the $*$ connective is a commutative, associative and monotone operator having Emp as a unit, and that it is distributive over finite disjunction or existential quantification. The next eight rules are about the assertion pair. They say

[3] The relation language can include separating implication and iterated separating conjunction for relations. However, we do not consider them to simplify presentation.

that the assertion pair is a monotone operator that preserves the separating connectives, conjunction, disjunction and the falsity. The remaining rules are specific to relational separation logic in this paper. They express that the diagonal relation Same denotes the equality of heaps; the boolean expression B is independent of heaps, so it can move into and out of spatial conjunction and assertion pair; the abbreviation $E \doteqdot E'$ is convertible into $E \doteq E'$. All these rules are sound, but they are not complete; this incompleteness cannot be avoided, because the set of all true implications between relations is not recursively enumerable, so there are no complete proof systems for such implications [16].

5 Relational Separation Logic

Relational separation logic consists of *Hoare quadruples*, by which we specify how two programs are related, and inference rules for deriving valid quadruples. In this section, we present these two components of the logic. We only consider deterministic commands in this presentation, and thus disregard commands that call `cons`. This is because when a program can call the nondeterministic allocator `cons`, a simple interpretation of quadruples does not validate the frame rule for quadruples. This problem about `cons` and our solution will be discussed in Section 7.

A Hoare quadruple

$$\{R\} \begin{matrix} C \\ C' \end{matrix} \{S\}$$

consists of pre-relation R and post-relation S, and commands C and C' that access disjoint sets of variables: $\mathsf{FV}(C) \cap \mathsf{FV}(C') = \emptyset$. Intuitively, this quadruple says that when C and C' are started from R-related states, either they both diverge or they terminate in S-related states. For s, t in Stores and a set X of variables, let $s[t|_X]$ be the result of updating s by the value in t for every x in X:

$$s[t|_X](x) \stackrel{\text{def}}{=} \begin{cases} t(x) \text{ if } x \in X \\ s(x) \text{ otherwise} \end{cases}$$

The precise meaning of a quadruple is given below:

Definition 3 *For deterministic commands C and C', a Hoare quadruple*

$$\{R\} \begin{matrix} C \\ C' \end{matrix} \{S\}$$

$$\frac{}{R * S \Rightarrow S * R} \qquad \frac{}{R * (S * T) \Rightarrow (R * S) * T} \qquad \frac{R \Rightarrow R' \quad S \Rightarrow S'}{R * S \Rightarrow R' * S'}$$

$$\frac{}{R * \mathsf{Emp} \Leftrightarrow R} \qquad \frac{}{R * \mathsf{False} \Leftrightarrow \mathsf{False}}$$

$$\frac{}{R * (S \vee T) \Leftrightarrow (R * S) \vee (R * T)} \qquad \frac{}{R * (\exists x.S) \Leftrightarrow \exists x.R * S}\ x \notin \mathsf{FV}(R)$$

$$\frac{}{\begin{pmatrix}\mathsf{emp}\\ \mathsf{emp}\end{pmatrix} \Leftrightarrow \mathsf{Emp}} \qquad \frac{}{\begin{pmatrix}P_1 * P_2\\ Q_1 * Q_2\end{pmatrix} \Leftrightarrow \begin{pmatrix}P_1\\ Q_1\end{pmatrix} * \begin{pmatrix}P_2\\ Q_2\end{pmatrix}} \qquad \frac{P_1 \Rightarrow P_2 \quad Q_1 \Rightarrow Q_2}{\begin{pmatrix}P_1\\ Q_1\end{pmatrix} \Rightarrow \begin{pmatrix}P_2\\ Q_2\end{pmatrix}}$$

$$\frac{}{\begin{pmatrix}P_1 \wedge P_2\\ Q\end{pmatrix} \Leftrightarrow \begin{pmatrix}P_1\\ Q\end{pmatrix} \wedge \begin{pmatrix}P_2\\ Q\end{pmatrix}} \qquad \frac{}{\begin{pmatrix}P\\ Q_1 \wedge Q_2\end{pmatrix} \Leftrightarrow \begin{pmatrix}P\\ Q_1\end{pmatrix} \wedge \begin{pmatrix}P\\ Q_2\end{pmatrix}}$$

$$\frac{}{\begin{pmatrix}P_1 \vee P_2\\ Q\end{pmatrix} \Leftrightarrow \begin{pmatrix}P_1\\ Q\end{pmatrix} \vee \begin{pmatrix}P_2\\ Q\end{pmatrix}} \qquad \frac{}{\begin{pmatrix}P\\ Q_1 \vee Q_2\end{pmatrix} \Leftrightarrow \begin{pmatrix}P\\ Q_1\end{pmatrix} \vee \begin{pmatrix}P\\ Q_2\end{pmatrix}}$$

$$\frac{}{\begin{pmatrix}P\\ \mathsf{false}\end{pmatrix} \Leftrightarrow \begin{pmatrix}\mathsf{false}\\ Q\end{pmatrix} \Leftrightarrow \mathsf{False}}$$

$$\frac{}{\mathsf{Emp} \Rightarrow \mathsf{Same}} \qquad \frac{}{\mathsf{Same} * \mathsf{Same} \Rightarrow \mathsf{Same}}$$

$$\frac{}{\begin{pmatrix}E \mapsto E_1, \ldots, E_n\\ E \mapsto E_1, \ldots, E_n\end{pmatrix} \Rightarrow \mathsf{Same}} \qquad \frac{}{\mathsf{Same} \wedge \left(\begin{pmatrix}E \mapsto E_1\\ E \mapsto E_1'\end{pmatrix} * \mathsf{True}\right) \Rightarrow E_1 = E_1'}$$

$$\frac{}{(R * Q) \wedge B \Leftrightarrow R * (Q \wedge B)} \qquad \frac{}{\begin{pmatrix}P \wedge B\\ Q\end{pmatrix} \Leftrightarrow \begin{pmatrix}P\\ Q \wedge B\end{pmatrix} \Leftrightarrow \begin{pmatrix}P\\ Q\end{pmatrix} \wedge B}$$

$$\frac{}{R * E \doteq\!\!\doteq E' \Leftrightarrow R \wedge E{=}E'} \qquad \frac{}{\begin{pmatrix}P\\ Q\end{pmatrix} * E \doteq\!\!\doteq E' \Leftrightarrow \begin{pmatrix}P * E \doteq E'\\ Q\end{pmatrix} \Leftrightarrow \begin{pmatrix}P\\ Q * E \doteq E'\end{pmatrix}}$$

Fig. 4. Sample Proof Rules for Relations

holds if and only if for all stores s and heaps h and h' that satisfy pre-relation R (i.e., $(s, h, h') \models R$),

(1) (s, h, C) is safe and (s, h', C') is safe;
(2) (s, h, C) can diverge if and only if (s, h', C) can diverge; and
(3) if $(s, h, C) \rightsquigarrow^* (t, h_1)$ and $(s, h', C') \rightsquigarrow^* (t', h'_1)$, then

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_1,\ h'_1) \models S.$$

Here $s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}]$ denotes the final store after running both C and C'. Since $\mathsf{FV}(C) \cap \mathsf{FV}(C') = \emptyset$ by assumption, commands C and C' modify disjoint sets of variables, so the store $s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}]$ records, without ambiguity, all the changes of the store s by C and C'.

We note two features of this definition of quadruples. First, the definition has the safety requirement for related states. Intuitively, this condition means that C and C' access only those heap cells which are guaranteed to exist by pre-relation R, or newly allocated. As in separation logic, this safety requirement is used in a proof rule for local reasoning, which adds an invariant relation to the pre- and post-relations of a quadruple; while adding a relation, the rule uses the safety requirement and the $*$ connective to ensure that the relation does not depend on cells modified by the commands in the quadruple, so the relation really becomes an invariant. Second, the definition of quadruples requires that C and C' should behave the same with respect to divergence. This requirement about divergence reflects the basic design decision of a quadruple specification: a quadruple is really about the equivalence of two programs modulo the different data structures used by the programs. Thus, if one program can diverge in a state, then the other program must be able to diverge in the corresponding state.

A Hoare quadruple is normally used for two kinds of specifications. The first kind of specifications require that programs C and C' be equivalent, but under some condition about initial states. Consider a specification which says that commands C and C' are equivalent if variables x of C and x' of C' point to cells initialized to 0.[4] This specification is expressed by the following quadruple:

$$\left\{ \left(\mathsf{Same} * \begin{pmatrix} x \mapsto 0 \\ x' \mapsto 0 \end{pmatrix} \right) \wedge x = x' \right\} \begin{matrix} C \\ C' \end{matrix} \{\mathsf{Same} \wedge x = x'\}$$

The second kind of specifications express that C' replaces a high-level "abstract" data structure in C by a low-level "concrete" data structure, but it preserves the meaning of C modulo the changes in this data structure. Suppose that both C and C' are programs that add an element to a finite set;

[4] $[x] := 0$ and `skip` are such commands.

C implements a finite set by a list variable α, and C' by a linked list x'. To relate the different implementations of a finite set by C and C', we use an inductively defined relation $\mathsf{Set}\ x'\ \alpha$:

$$\mathsf{Set}\ x'\ \epsilon \stackrel{\text{def}}{=} x' \doteq \mathtt{nil}$$

$$\mathsf{Set}\ x'\ E{::}\alpha \stackrel{\text{def}}{=} \exists x'_1.\ \begin{pmatrix}\mathsf{emp}\\ x' \mapsto E, x'_1\end{pmatrix} * \mathsf{Set}\ x'_1\ \alpha$$

Intuitively, this relation says that the linked list from x' implements list α. Using relation $\mathsf{Set}\ x'\ \alpha$, we now express that C and C' are equivalent modulo the different implementations of a finite set, by the following quadruple:

$$\{\mathsf{Same} * \mathsf{Set}\ x'\ \alpha\} \begin{matrix} C \\ C' \end{matrix} \{\mathsf{Same} * \mathsf{Set}\ x'\ \alpha\}$$

Relational separation logic has three proof rules specific to particular language constructs, and five "structural" rules that are not restricted to any language constructs. These rules appear in Figure 5.

The construct-specific rules deal with loop, conditional statement, and sequential composition. The rule for loop says that if the bodies of two `while` loops preserve a relation R, then the loops themselves preserve R. Note that this rule is similar to the `while` rule for partial correctness in Hoare logic. However, the additional premise $R \Rightarrow (B \Leftrightarrow B')$ lets the rule for quadruples to conclude a stronger statement than the rule for triples: while the rule for triples does not ensure the termination, the rule for quadruples guarantees that either both loops diverge, or both terminate. The other two rules for conditional statement and sequential composition are also similar to the corresponding rules in Hoare logic. The only differences are that the rule for conditional statement has an additional requirement $R \Rightarrow (B \Leftrightarrow B')$, which ensures that C and C' take the same true or false branch; and that the rule for sequential composition can be applied when one of C and C' is a single atomic command, because `skip` can be inserted using the equivalence $\mathtt{skip}; C = C; \mathtt{skip} = C$.

The structural rules include embedding rule, frame rule, Consequence, Conjunction, and Auxiliary Variable Elimination. Among these five rules, the frame rule and embedding rule play the most prominent roles to simplify reasoning about quadruples.

The frame rule adds an invariant relation to the pre- and post- relations, if the invariant does not read cells or variables that programs C and C' access. The merit of the rule is that checking this condition about memory access is cheap: the rule ensures that C and C' do not modify variables in the invariant

Loop

$$\frac{R \Rightarrow (B \Leftrightarrow B') \quad \{R \wedge B\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{R\}}{\{R\}\begin{smallmatrix} \texttt{while } B \texttt{ do } C \texttt{ od} \\ \texttt{while } B' \texttt{ do } C' \texttt{ od} \end{smallmatrix}\{R \wedge \neg B\}}$$

Conditional Statement

$$\frac{R \Rightarrow (B \Leftrightarrow B') \quad \{R \wedge B\}\begin{smallmatrix} C_1 \\ C_1' \end{smallmatrix}\{S\} \quad \{R \wedge \neg B\}\begin{smallmatrix} C_2 \\ C_2' \end{smallmatrix}\{S\}}{\{R\}\begin{smallmatrix} \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \\ \texttt{if } B' \texttt{ then } C_1' \texttt{ else } C_2' \end{smallmatrix}\{S\}}$$

Sequencing

$$\frac{\{R\}\begin{smallmatrix} C_1 \\ C_1' \end{smallmatrix}\{S\} \quad \{S\}\begin{smallmatrix} C_2 \\ C_2' \end{smallmatrix}\{T\}}{\{R\}\begin{smallmatrix} C_1; C_2 \\ C_1'; C_2' \end{smallmatrix}\{T\}}$$

Frame Rule

$$\frac{\{R\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S\}}{\{R * T\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S * T\}} \; (\mathsf{Mod}(C, C') \cap \mathsf{FV}(T) = \emptyset)$$

Embedding Rule

$$\frac{[P]\, C\, [Q] \quad [P']\, C'\, [Q']}{\left\{ \begin{pmatrix} P \\ P' \end{pmatrix} \right\} \begin{smallmatrix} C \\ C' \end{smallmatrix} \left\{ \begin{pmatrix} Q \\ Q' \end{pmatrix} \right\}} \begin{pmatrix} \mathsf{Mod}(C) \cap \mathsf{FV}(P', Q') = \emptyset \\ \mathsf{Mod}(C') \cap \mathsf{FV}(P, Q) = \emptyset \end{pmatrix}$$

Consequence

$$\frac{R \Rightarrow R_1 \quad \{R_1\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S_1\} \quad S_1 \Rightarrow S}{\{R\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S\}}$$

Conjunction

$$\frac{\{R_1\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S_1\} \quad \{R_2\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S_2\}}{\{R_1 \wedge R_2\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S_1 \wedge S_2\}}$$

Auxiliary Variable Elimination

$$\frac{\{R\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S\}}{\{\exists x.\, R\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{\exists x.\, S\}} \; (x \notin \mathsf{FV}(C, C'))$$

Fig. 5. Proof Rules for Hoare Quadruples

T, just by checking $\mathsf{Mod}(C, C') \cap \mathsf{FV}(T) = \emptyset$; and it guarantees that C and C' do not access heap cells that T depends on, simply by using the $*$ connective in the conclusion. This rule is an extension of the frame rule in separation logic, and just as the frame rule in separation logic, it simplifies specification and verification of a quadruple, by making them focus on what the compared programs access.

The embedding rule concludes a quadruple $\left\{ \begin{pmatrix} P \\ P' \end{pmatrix} \right\}_{C'}^{C} \left\{ \begin{pmatrix} Q \\ Q' \end{pmatrix} \right\}$ from two triples $[P]C[Q]$ and $[P']C[Q']$. Note that the rule demands total correctness triples, because a quadruple requires that either both of the compared commands diverge, or they terminate. This rule often contributes to reduce the complexity of verification, because it allows the use of existing verification techniques in separation logic.

Besides being used in a verification of a specific quadruple, the structural rules are also used to derive useful proof rules for quadruples, especially those that compare atomic commands. The derivation usually consists of three steps: first, we embed derivable Hoare triples to get a quadruple; second, we attach an invariant using the frame rule; finally, we adjust the pre- and post- relations using Consequence. For example, the below rules for quadruples can be obtained following this pattern:

Lookup

$$\frac{}{\left\{ \begin{pmatrix} y \mapsto x_0 \\ P \end{pmatrix} * R \right\} \; x := [y] \; \left\{ \begin{pmatrix} y \mapsto x \\ P \end{pmatrix} * R * x \doteqdot x_0 \right\}} \; (x \notin \mathsf{FV}(P, R))$$

Update

$$\frac{}{\left\{ \begin{pmatrix} y \mapsto \text{-} \\ P \end{pmatrix} * R \right\} \; [y] := E \; \left\{ \begin{pmatrix} y \mapsto E \\ P \end{pmatrix} * R \right\}}$$

We show the derivation of the first rule; the second rule can be derived simi-

larly.

$$\dfrac{\dfrac{\dfrac{\overline{[y \mapsto x_0]\, x := [y]\, [y \mapsto x * x \doteq x_0]} \quad \overline{[P]\, \mathtt{skip}\, [P]}}{\left\{ \begin{pmatrix} y \mapsto x_0 \\ P \end{pmatrix} \right\} x := [y] \left\{ \begin{pmatrix} y \mapsto x * x \doteq x_0 \\ P \end{pmatrix} \right\}}\ \text{Embedding}}{\left\{ \begin{pmatrix} y \mapsto x_0 \\ P \end{pmatrix} * R \right\} x := [y] \left\{ \begin{pmatrix} y \mapsto x * x \doteq x_0 \\ P \end{pmatrix} * R \right\}}\ \text{Frame Rule}}{\left\{ \begin{pmatrix} y \mapsto x_0 \\ P \end{pmatrix} * R \right\} x := [y] \left\{ \begin{pmatrix} y \mapsto x \\ P \end{pmatrix} * R * x \doteqdot x_0 \right\}}\ \text{Consequence}$$

In this paper, we present a proof of a quadruple in a linear form, using Bornat's technique [6]. For instance, a derivation

$$\dfrac{\dfrac{\dfrac{\{R_0\} \begin{matrix} C \\ C' \end{matrix} \{S_0\}}{\{R_0 * T\} \begin{matrix} C \\ C' \end{matrix} \{S_0 * T\}}\ \text{Frame} \qquad R \Rightarrow (R_0 * T) \qquad (S_0 * T) \Rightarrow S}{\{R\} \begin{matrix} C \\ C' \end{matrix} \{S\}}\ \text{Consequence}}{\{\exists a_0.\ R\} \begin{matrix} C \\ C' \end{matrix} \{\exists a_0.\ S\}}\ \text{Auxiliary Variable Elimination}$$

is shown as follows:

$$\begin{array}{l} \{\exists a_0.\ R\} \\ \quad a_0 : \left[\begin{array}{l} \{R\} \\ \{R_0 * T\} \\ \quad \text{Framed:} \left[\begin{array}{ccc} & \{R_0\} & \\ C & & C' \\ & \{S_0\} & \end{array} \right] \\ \{S_0 * T\} \\ \{S\} \end{array} \right] \\ \{\exists a_0.\ S\} \end{array}$$

Example 4 Consider commands C and C' that traverse a linked list and set the value of each node using the value of cell y:

$$
\begin{array}{lll}
C \stackrel{\text{def}}{=} & \texttt{while}\ (c \neq \texttt{nil}) & \\
& \quad \texttt{do}\ \ x := [y]; & \\
& \qquad [c] := -x; & \\
& \qquad c := [c+1] & \\
& \quad \texttt{od} &
\end{array}
\qquad
\begin{array}{lll}
C' \stackrel{\text{def}}{=} & x' := [y'] & \\
& \texttt{while}\ (c' \neq \texttt{nil}) & \\
& \quad \texttt{do}\ \ [c'] := -x'; & \\
& \qquad c' := [c'+1] & \\
& \quad \texttt{od} &
\end{array}
$$

The second program optimizes the first, by moving $x := [y]$ outside of the loop. The rationale for this optimization is that $x := [y]$ assigns the same value for all iterations of the loop, so it can be executed just once. Such a correctness argument, however, breaks if cell y is modified by the update statement $[c] := -x$ in the loop. Thus, our quadruple specification for this optimization puts a precondition that cell x is not part of a linked list, and expresses the equivalence between C and C' under this condition.

$$\mathsf{List}\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \ \vee\ \exists na.\ \begin{pmatrix} x \mapsto a, n \\ x \mapsto a, n \end{pmatrix} * \mathsf{List}\ n$$

$$\left\{ \left(\mathsf{Same} * \mathsf{List}\ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} \right) \wedge y = y' \wedge c = c' \right\} \begin{matrix} C \\ C' \end{matrix} \{\mathsf{Same} \wedge y = y' \wedge c = c'\}$$

Note that the $*$ connective in the pre-relation ensures that cells y and y' do not belong to linked lists, so their contents are not modified by C and C'.

The main part of the proof for this quadruple specification is to find a loop-invariant relation, and to show that the relation is preserved. We pick the following relation as a candidate for a loop invariant:

$$\mathsf{Inv} \stackrel{\text{def}}{=} \mathsf{Same} * \mathsf{List}\ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \doteq y' * c' \doteq c * x' \doteq x_0$$

This relation is almost identical to the pre-relation; it just adds a fact that the variable x' currently holds the value of cell y'. We will show that Inv is preserved, by proving the following quadruple:

$$\{\mathsf{Inv} \wedge c \neq \texttt{nil}\} \begin{matrix} x := [y]; [c] := -x; c := [c+1] \\ [c'] := -x'; c' := [c'+1] \end{matrix} \{\mathsf{Inv}\}$$

The first step of the Inv-preservation proof is to summarize the local effects of the loop bodies of C and C' by Hoare triples. In this step, we prove the following Hoare triples for the bodies of C and C' in separation logic:

$$
\begin{array}{ll}
[c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0] & [c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0] \\
\quad x := [y]; & \quad [c'] := -x'; \\
[c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 * x \doteq x_0] & [c_0 \mapsto -x_0, c_1 * c' \doteq c_0 * x' \doteq x_0] \\
\quad [c] := -x; & \quad c' := [c' + 1] \\
[c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_0] & [c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0] \\
\quad c := [c + 1] & \\
[c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1] &
\end{array}
$$

The proved triples describe all the changes of memory by the two commands. Note that the triples and their proofs are local; they mention only those accessed heap cells and variables by the commands.

The next step converts these triples to a "local" quadruple using the embedding rule:

$$
\frac{\begin{array}{ll}
[c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0] & [c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0] \\
x{:=}[y];\ [c]{:=}-x;\ c{:=}[c+1] & [c']{:=}-x';\ c'{:=}[c'+1] \\
[c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1] & [c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0]
\end{array}}{\left\{ \begin{pmatrix} c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix} \right\} \begin{array}{ll} x := [y]; & [c'] := -x'; \\ [c] := -x; & c' := [c' + 1] \\ c := [c + 1] & \end{array} \left\{ \begin{pmatrix} c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1 \\ c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0 \end{pmatrix} \right\}} \text{Embedding}
$$

The third step transforms this local quadruple to global one. It uses the frame rule to extend the pre- and post- relations of the local quadruple with a global

fact.

$$\frac{\left\{\begin{pmatrix} c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix}\right\}}{\left\{\begin{pmatrix} c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ c_1 * \mathsf{Same} * y \ddot{=} y' * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix}\right\}}\ \text{Frame}$$

$$\begin{array}{ll} x := [y]; & [c'] := -x'; \\ [c] := -x; & c' := [c' + 1] \\ c := [c + 1] & \end{array}$$

$$\left\{\begin{pmatrix} c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1 \\ c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0 \end{pmatrix}\right\}$$

$$\left\{\begin{pmatrix} c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ c_1 * \mathsf{Same} * y \ddot{=} y' * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix}\right\}$$

$$\begin{array}{ll} x := [y]; & [c'] := -x'; \\ [c] := -x; & c' := [c' + 1] \\ c := [c + 1] & \end{array}$$

$$\left\{\begin{pmatrix} c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1 \\ c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ c_1 * \mathsf{Same} * y \ddot{=} y' * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix}\right\}$$

Note that after this step, the pre- and post- relations of the quadruple became closer to the loop invariant; the pre- and post- relations of the global relations now describe not just the cells and variables that programs access, but also all the cells and variables that the loop invariant mentions.

The final step of the Inv-preservation proof eliminates auxiliary variables from the global quadruple, and then, it strengthen the pre-relation and weaken the post-relation of the resulting quadruple. Figures 6, 7 and 8 shows this step of the proof.

□

One evident shortcoming of our logic is that the proof rules assume that two commands have similar control structures. When this assumption breaks, our new rules for quadruples do not help, and we mostly have to reason about C and C' individually in separation logic. For example, our logic is not effective for proving the correctness of the loop unrolling, because the loop unrolling changes the control structure of an input program. In Section 8, we will discuss how to use the ideas from the work on translation validation or credible compilation [17,18] to overcome this shortcoming.

$$\left\{ \mathsf{Same} * \mathsf{List}\ \ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0\ \wedge\ c \neq \mathtt{nil} \right\}$$

$\because$ when is shown in Figure 7

$$\left\{ \exists c_0 c_1.\ \mathsf{Same} * \begin{pmatrix} c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ \ c_1 * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' \right\}$$

$$c_0 c_1 : \left[\begin{array}{l} \left\{ \mathsf{Same} * \begin{pmatrix} c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ \ c_1 * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' \right\} \\ \qquad x := [y]; \qquad\qquad [c'] := -x'; \\ \qquad [c] := -x; \qquad\quad c' := [c' + 1] \\ \qquad c := [c + 1] \\ \left\{ \mathsf{Same} * \begin{pmatrix} c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1 \\ c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ \ c_1 * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' \right\} \end{array} \right]$$

$$\left\{ \exists c_0 c_1.\ \mathsf{Same} * \begin{pmatrix} c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1 \\ c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ \ c_1 * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' \right\}$$

$\because$ what is shown in Figure 8

$$\left\{ \mathsf{Same} * \mathsf{List}\ \ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \right\}$$

Fig. 6. Proof of the Invariant Preservation (Proof Outline)

6 Schorr-Waite Graph Marking Algorithm

In this section, we apply our logic to more realistic programs; we show that the Schorr-Waite graph marking algorithm is equivalent to the depth-first traversal.

The Schorr-Waite graph marking algorithm is a depth-first traversal of a graph, except that it uses less space: the Schorr-Waite algorithm runs in constant space, while the depth-first traversal runs in linear space in the worst case. The depth-first traversal needs linear space, because it uses an additional stack data structure to record which nodes are currently being traversed and thus need to be considered again. However, the Schorr-Waite algorithm does not need this space, because it implements the stack inside the input graph: the Schorr-Waite algorithm temporarily changes the contents of some nodes in the input graph, but restores the contents back to their original value when it terminates.

$$\left\{ \mathsf{Same} * \mathsf{List}\ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \ \wedge\ c \neq \mathtt{nil} \right\}$$

$\because$ the definition of $\mathsf{List}\ c$

$$\left\{ \mathsf{Same} * \left(\exists c_1. \begin{pmatrix} c \mapsto \text{-}, c_1 \\ c \mapsto \text{-}, c_1 \end{pmatrix} * \mathsf{List}\ c_1 \right) * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \right\}$$

$\because (R * \exists x.\ S) \Rightarrow (\exists x.\ R * S)$

$$\left\{ \exists c_1. \mathsf{Same} * \begin{pmatrix} c \mapsto \text{-}, c_1 \\ c \mapsto \text{-}, c_1 \end{pmatrix} * \mathsf{List}\ c_1 * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \right\}$$

$\because c_0$ is fresh

$$\left\{ \exists c_0 c_1. \mathsf{Same} * \begin{pmatrix} c_0 \mapsto \text{-}, c_1 \\ c_0 \mapsto \text{-}, c_1 \end{pmatrix} * \mathsf{List}\ c_1 * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c_0 * x' \ddot{=} x_0 * c \ddot{=} c_0 \right\}$$

$$\because \begin{pmatrix} P \\ Q \end{pmatrix} * E \ddot{=} E' \Leftrightarrow \begin{pmatrix} P * E \doteq E' \\ Q \end{pmatrix} \Leftrightarrow \begin{pmatrix} P \\ Q * E \doteq E' \end{pmatrix}$$

$$\left\{ \exists c_0 c_1. \mathsf{Same} * \begin{pmatrix} c_0 \mapsto \text{-}, c_1 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ c_1 * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' \right\}$$

$$\because \begin{pmatrix} P_1 \\ Q_1 \end{pmatrix} * \begin{pmatrix} P_2 \\ Q_2 \end{pmatrix} \Leftrightarrow \begin{pmatrix} P_1 * P_2 \\ Q_1 * Q_2 \end{pmatrix} \quad \text{and} \quad P * \mathsf{emp} \Leftrightarrow P$$

$$\left\{ \exists c_0 c_1. \mathsf{Same} * \begin{pmatrix} c_0 \mapsto \text{-}, c_1 * y \mapsto x_0 * c \doteq c_0 \\ c_0 \mapsto \text{-}, c_1 * c' \doteq c_0 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ c_1 * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' \right\}$$

Fig. 7. Proof of the Invariant Preservation (Implication between Preconditions)

The technique for implementing this implicit stack forms the most creative part of the Schorr-Waite algorithm. For every node in the stack, the Schorr-Waite algorithm "reverses" one of the links of the node, so that the reversed link points to the previous node in the stack. For instance, when the stack contains nodes n_3, n_2 and n_1 in the first graph in Figure 9, the Schorr-Waite algorithm reverses the right field of n_1, and the left fields of n_2 and n_3, and gives the second graph in the same figure.

Figure 10 shows the implementation of the Schorr-Waite algorithm, `SW`, and Figure 11 the implementation of the depth-first traversal, `DFT`. The implementations regard four consecutive cells as a single node. The first two cells store links to other nodes, and the third and fourth cells contain information about traversing; the third cell indicates whether the node is visited, and the fourth

$$\left\{ \exists c_0 c_1.\ \mathsf{Same} * \begin{pmatrix} c_0 \mapsto -x_0, c_1 * y \mapsto x_0 * c \doteq c_1 \\ c_0 \mapsto -x_0, c_1 * c' \doteq c_1 * x' \doteq x_0 \end{pmatrix} * \mathsf{List}\ \ c_1 * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' \right\}$$

$$\because \begin{pmatrix} P \\ Q \end{pmatrix} * E \ddot{=} E' \Leftrightarrow \begin{pmatrix} P * E \doteq E' \\ Q \end{pmatrix} \Leftrightarrow \begin{pmatrix} P \\ Q * E \doteq E' \end{pmatrix}, \quad \text{and} \quad c = c_1$$

$$\left\{ \exists c_0.\ \mathsf{Same} * \begin{pmatrix} c_0 \mapsto -x_0, c * y \mapsto x_0 \\ c_0 \mapsto -x_0, c \end{pmatrix} * \mathsf{List}\ \ c * \begin{pmatrix} \mathsf{emp} \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \right\}$$

$$\because \begin{pmatrix} P_1 \\ Q_1 \end{pmatrix} * \begin{pmatrix} P_2 \\ Q_2 \end{pmatrix} \Leftrightarrow \begin{pmatrix} P_1 * P_2 \\ Q_1 * Q_2 \end{pmatrix} \ \text{and}\ \ P * \mathsf{emp} \Leftrightarrow P$$

$$\left\{ \exists c_0.\ \mathsf{Same} * \begin{pmatrix} c_0 \mapsto -x_0, c \\ c_0 \mapsto -x_0, c \end{pmatrix} * \mathsf{List}\ \ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \right\}$$

$$\because \begin{pmatrix} E \mapsto E_1, E_2 \\ E \mapsto E_1, E_2 \end{pmatrix} \Rightarrow \mathsf{Same}$$

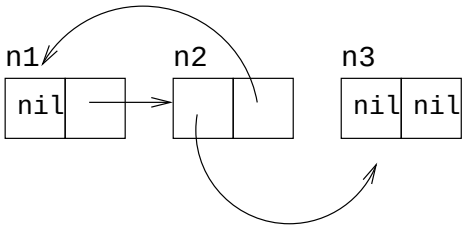
$$\left\{ \mathsf{Same} * \mathsf{Same} * \mathsf{List}\ \ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \right\}$$

$$\because \mathsf{Same} * \mathsf{Same} \Rightarrow \mathsf{Same}$$

$$\left\{ \mathsf{Same} * \mathsf{List}\ \ c * \begin{pmatrix} y \mapsto x_0 \\ y' \mapsto x_0 \end{pmatrix} * y \ddot{=} y' * c' \ddot{=} c * x' \ddot{=} x_0 \right\}$$

Fig. 8. Proof of the Invariant Preservation (Implication between Postconditions)

Initial Graph:



Graph after Pointer Reversal:

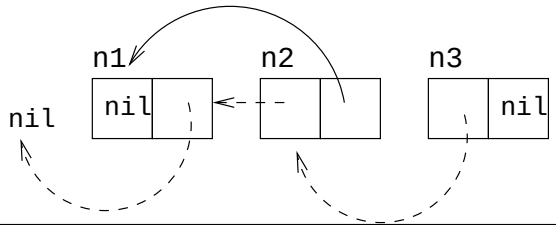


Fig. 9. Example of Pointer Reversal

cell which link of the node is currently being traversed. [5] We call the nth cell in a node "nth field", and use macros $x.\texttt{Left}$, $x.\texttt{Right}$, $x.\texttt{Mark}$, and $x.\texttt{Current}$ to denote these four fields of a node x. We also use macros for values in the last two fields; for the third field, we use `Marked` and `Unmarked`, which are expanded to 1 and 0, respectively; for the fourth field, we use `Left` and `Right`, which are expanded to 1 and 0, respectively.

The control structures of `SW` and `DFT` are the same. In both cases, the body of the loop consists of three branches. The first branch handles the "push" case, where the traversing algorithm hits an unmarked node so it pushes the node into the stack; the second branch handles the "swing" case, where the algorithm stops traversing the left link of the top node in the stack and starts to look at the right link, because it has visited all the reachable nodes from the left link; finally, the third branch deals with the "pop" case, where the algorithm pops the top node from the stack, because it has marked all the nodes reachable from the top node.

Note that in `DFT`, we use a list variable α', as opposed to a linked list, for a stack data structure. Using a list variable makes code for a depth-first traversal simpler, thus more likely to be correct.

6.1 Specification

Let's define an assertion $\mathsf{noDangling}\ G$ which means that no links in the input graph are dangling and G is the set of all the nodes in the graph.

$$\mathsf{noDangling}\, G \stackrel{\text{def}}{=} \forall_* x \in G.\, \exists lr.\, (x \mapsto l, r, \text{-}, \text{-}) \wedge l \in G \cup \{\texttt{nil}\} \wedge r \in G \cup \{\texttt{nil}\}.$$

We specify the equivalence of `SW` and `DFT` by the following quadruple:

$$\left\{ \mathsf{Same} \ \wedge \ c = c' \ \wedge \ \begin{pmatrix} \mathsf{noDangling}\ G \wedge c \in G \cup \{\texttt{nil}\} \\ \mathsf{noDangling}\ G \wedge c' \in G \cup \{\texttt{nil}\} \end{pmatrix} \right\} \begin{matrix} \texttt{SW} \\ \texttt{DFT} \end{matrix} \{\mathsf{Same}\}$$

The pre-relation in the quadruple requires that all the inputs to `SW` and `DFT` be the same; the input graphs and the given roots c and c' should be the same. Moreover, it also puts a condition to ensure that `SW` and `DFT` run without memory errors; the links in the input graph should not be dangling, and c and c' should point to the first cells of some nodes, not the middle of nodes. Then,

[5] The actual implementation of the Schorr-Waite graph marking algorithm uses two single bits, instead of two whole cells, to record these informations about traversing. We ignore this optimization here because first, this optimization does not raise any new verification issues, and second, by ignoring the optimization, we can focus on the main part of the Schorr-Waite algorithm, that is, pointer reversal.

```
if (c ≠ nil) then p := c.Left;
                  c.Mark := Marked;
                  c.Current := Left;
                  c.Left := nil
             else p := nil;
while (c ≠ nil)
      do if (p ≠ nil) then m := p.Mark else m := Marked;
         if (p ≠ nil ∧ m ≠ Marked)
         then t := p.Left;    // SWPush
              p.Left := c;
              c := p;
              p := t;
              c.Mark := Marked;
              c.Current := Left
         else d := c.Current;
              if (d = Left)
              then t := c.Left;    // SWSwing
                   c.Left := p;
                   p := c.Right;
                   c.Right := t;
                   c.Current := Right
              else t := p;    // SWPop
                   p := c;
                   c := p.Right;
                   p.Right := t
```

Fig. 10. Schorr-Waite Graph Marking Algorithm

the post-relation guarantees that the output graphs for SW and DFT are the same.

6.2 Verification

The complete verification consists of four proofs about an invariant relation. The first proof shows that the invariant is established before the loops of SW and DFT first get executed; the second shows that the invariant is preserved by the bodies of the loops; the third that the invariant relation implies the post-relation. Note that these three conditions are the usual proof steps in Hoare logic. Our logic, however, requires one more proof; we need to show that if an invariant relation holds, then the condition of the loop in SW is equivalent to that in DFT. Among these four proofs, we will focus on the preservation of the invariant and the equivalence of the loop conditions.

Let's define a relation $\mathsf{Stack}\ p\ c\ \alpha'$ which expresses that (1) c is the top node of

```
if (c' ≠ nil) then p' := c'.Left;
                   c'.Mark := Marked;
                   c'.Current := Left;
                   α' := c'::ϵ
              else α' := ϵ;
                   p' := nil;
while (α' ≠ ϵ)
      do if (p' ≠ nil) then m' := p'.Mark else m' := Marked;
         if (p' ≠ nil ∧ m' ≠ Marked)
         then α' := p'::α';    // DFTPush
              p'.Mark := Marked;
              p'.Current := Left;
              p' := p'.Left
         else d' := (hd α').Current;
              if (d' = Left)
              then (hd α').Current := Right;    // DFTSwing
                   p' := (hd α').Right
              else p' := hd α';    // DFTPop
                   α' := tl α'
```

Fig. 11. Depth-First Traversal

stack α', (2) all the nodes in the stack are marked, (3) stack α' is implemented by reversing pointers from c, and (4) the original value of the reversed link of c is p:

$$\mathsf{Stack}\, p\, c\, \epsilon \stackrel{\text{def}}{=} c \doteq \mathtt{nil}$$

$$\mathsf{Stack}\, p\, c\, (x'::\alpha') \stackrel{\text{def}}{=} \left(\exists n_0 r.\, c \ddot{=} x' * \begin{pmatrix} c \mapsto n_0, r, \mathtt{Marked}, \mathtt{Left} \\ c \mapsto p, r, \mathtt{Marked}, \mathtt{Left} \end{pmatrix} * \mathsf{Stack}\, c\, n_0\, \alpha'\right) \vee \left(\exists n_0 l.\, c \ddot{=} x' * \begin{pmatrix} c \mapsto l, n_0, \mathtt{Marked}, \mathtt{Right} \\ c \mapsto l, p, \mathtt{Marked}, \mathtt{Right} \end{pmatrix} * \mathsf{Stack}\, c\, n_0\, \alpha'\right)$$

In the second clause, c and x' denote the top node in the stack, and n_0 the second node. Thus, this clause says that the left or right link of node x' is reversed in the first heap, so that it points to the previous node n_0 in the stack. The original value of the reversed link, the clause also says, is stored in variable p.

Using $\mathsf{Stack}\, p\, c\, \alpha$, we define a candidate for a loop invariant:

$$\mathsf{Inv} \stackrel{\text{def}}{=} \mathsf{Same} * \mathsf{Stack}\, p\, c\, \alpha \wedge p{=}p' \wedge \begin{pmatrix} \mathsf{noDangling}\, G \wedge \{p, c\} \subseteq G \cup \{\mathtt{nil}\} \\ \mathsf{noDangling}\, G \wedge \{p'\} \cup \mathsf{toSet}(\alpha) \subseteq G \cup \{\mathtt{nil}\} \end{pmatrix}.$$

The first conjunct $\mathsf{Same} * \mathsf{Stack}\, p\, c\, \alpha$ explains the main difference between `SW`

and DFT. It says that the heaps h for SW and h' for DFT are identical except the nodes in stack α; in h, one of the link fields of these nodes is reversed, but in h', these link fields have their initial values. The meaning of the other two conjuncts are straightforward. The second conjunct means that SW and DFT are currently looking at the same node, and the third conjunct says that neither variables nor the link fields of the nodes in the graph contain dangling pointers.

We first prove that when Inv holds, the loop condition for SW is equivalent to the loop condition for DFT.

Lemma 5 *Relation* Inv *implies that* $c \neq \texttt{nil}$ *is equivalent to* $\alpha' \neq \epsilon$.

PROOF. We will show that when Inv holds, $\alpha' = \epsilon$ is equivalent to $c = \texttt{nil}$. We first derive $(\mathsf{Inv} \wedge \alpha' = \epsilon) \Rightarrow c = \texttt{nil}$:

$$
\begin{array}{ll}
& \mathsf{Inv} \ \wedge \ \alpha' = \epsilon \\
\Longrightarrow & \because R \wedge S \Rightarrow R \\
& (\mathsf{Same} * \mathsf{Stack}\, p\, c\, \alpha') \ \wedge \ \alpha' = \epsilon \\
\Longrightarrow & \because ((R * S) \wedge B) \Leftrightarrow (R * (S \wedge B)) \\
& \mathsf{Same} * (\mathsf{Stack}\, p\, c\, \alpha' \ \wedge \ \alpha' = \epsilon) \\
\Longrightarrow & \because \text{the definition of } \mathsf{Stack}\, p\, c\, \alpha' \\
& \mathsf{Same} * (\mathsf{Emp} \ \wedge \ c = \texttt{nil}) \\
\Longrightarrow & \because ((R * S) \wedge B) \Leftrightarrow (R * (S \wedge B)) \\
& (\mathsf{Same} * \mathsf{Emp}) \ \wedge \ c = \texttt{nil} \\
\Longrightarrow & \because R \wedge S \Rightarrow R \\
& c = \texttt{nil}
\end{array}
$$

For the other implication, we derive $(\mathsf{Inv} \wedge c = \texttt{nil} \wedge \alpha' \neq \epsilon) \Rightarrow \mathsf{False}$:

$$
\begin{array}{ll}
& \mathsf{Inv} \ \wedge \ c = \texttt{nil} \ \wedge \ \alpha' \neq \epsilon \\
\Longrightarrow & \because R \wedge S \Rightarrow R \\
& (\mathsf{Same} * \mathsf{Stack}\, p\, c\, \alpha') \ \wedge \ c = \texttt{nil} \ \wedge \ \alpha' \neq \epsilon \\
\Longrightarrow & \because ((R * S) \wedge B) \Leftrightarrow (R * (S \wedge B)) \\
& \mathsf{Same} * (\mathsf{Stack}\, p\, c\, \alpha' \ \wedge \ c = \texttt{nil} \ \wedge \ \alpha' \neq \epsilon) \\
\Longrightarrow & \because \text{the definition of } \mathsf{Stack}\, p\, c\, \alpha' \\
& \mathsf{Same} * \left(\begin{pmatrix} c \mapsto \text{-} \\ \\ c \mapsto \text{-} \end{pmatrix} * \mathsf{True} \ \wedge \ c = \texttt{nil} \right) \\
\Longrightarrow & \because c \mapsto \text{-} \Rightarrow c \neq \texttt{nil} \\
& \mathsf{Same} * \left(\begin{pmatrix} c \mapsto \text{-} \wedge c \neq \texttt{nil} \\ \\ c \mapsto \text{-} \end{pmatrix} * \mathsf{True} \ \wedge \ c = \texttt{nil} \right)
\end{array}
$$

$$
\begin{array}{ll}
\Longrightarrow & \because \left(\begin{pmatrix} P \wedge B \\ Q \end{pmatrix} \Leftrightarrow \begin{pmatrix} P \\ Q \end{pmatrix} \wedge B\right) \text{ and } (R * (S \wedge B) \Leftrightarrow R * S \wedge B) \\
& \mathsf{Same} * \left(\begin{pmatrix} c \mapsto \text{-} \\ c \mapsto \text{-} \end{pmatrix} * \mathsf{True} \ \wedge \ c \neq \texttt{nil} \ \wedge \ c = \texttt{nil}\right) \\
\Longrightarrow & \\
& \mathsf{Same} * \mathsf{False} \\
\Longrightarrow & \because R * \mathsf{False} \Rightarrow \mathsf{False} \\
& \mathsf{False}
\end{array}
$$

□

We now consider the preservation of Inv. As in Hoare logic, the proof of invariant preservation forms the most important part of the verification. We will explain the common pattern in the preservation proof for Inv, by outlining the proof of the push branch. In particular, we will show the following quadruple:

$$
\left\{\left(\mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \\ p' \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha\right) \ \wedge \ p = p'\right\}
$$

SWPush DFTPush

$$
\{(\mathsf{Same} * \mathsf{Stack}\, p\, c\, \alpha) \ \wedge \ p = p'\}
$$

where SWPush and DFTPush, respectively, denote the code in the push branches of SW and DFT. Note that this quadruple does not fully express that the push branch preserves the invariant. It is because in addition to Inv, the pre-relation requires that p and p' denote an unvisited node that is not in the stack, and the post-relation ensures only the first two conjuncts of Inv. However, proving this restricted quadruple illustrates the key idea.

The proof for the push branch consists of five steps. First, we reason about local effects of SWPush and DFTPush in separation logic, and obtain two "local" Hoare triples. Second, we embed these triples in our logic and obtain a "local quadruple." Third, we change this local quadruple to "global" one, using the frame rule. Fourth, we eliminate auxiliary variables that are used to denote the initial values of variables. Finally, we strengthen the pre-relation and weaken the post-relation, and obtain the required quadruple.

For the first step of the proof, we summarize what SWPush and DFTPush do for nodes p and p', and variables p, c, p' and α'. For this purpose, we use proof

Proof of a triple for `SWPush`:

$[(p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * c \doteq c_0 * p \doteq p_0]$
$\quad t := p.\texttt{Left};$
$[(p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * c \doteq c_0 * p \doteq p_0 * t \doteq l_0]$
$\quad p.\texttt{Left} := c;$
$[(p_0 \mapsto c, r_0, \text{-}, \text{-}) * c \doteq c_0 * p \doteq p_0 * t \doteq l_0]$
$[(p_0 \mapsto c_0, r_0, \text{-}, \text{-}) * p \doteq p_0 * t \doteq l_0]$
$\quad c := p;$
$[(p_0 \mapsto c_0, r_0, \text{-}, \text{-}) * c \doteq p_0 * t \doteq l_0]$
$\quad p := t;$
$[(p_0 \mapsto c_0, r_0, \text{-}, \text{-}) * c \doteq p_0 * p \doteq l_0]$
$\quad c.\texttt{Mark} := \texttt{Marked};$
$[(p_0 \mapsto c_0, r_0, \texttt{Marked}, \text{-}) * c \doteq p_0 * p \doteq l_0]$
$\quad c.\texttt{Current} := \texttt{Left}$
$[(p_0 \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left}) * c \doteq p_0 * p \doteq l_0]$

Proof of a triple for `DFTPush`:

$[(p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * \alpha' \doteq \alpha_0 * p' \doteq p_0]$
$[(p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * (p'::\alpha' \doteq p_0::\alpha_0) * p' \doteq p_0]$
$\quad \alpha' := p'::\alpha';$
$[(p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * (\alpha' \doteq p_0::\alpha_0) * p' \doteq p_0]$
$\quad p'.\texttt{Mark} := \texttt{Marked};$
$[(p_0 \mapsto l_0, r_0, \texttt{Marked}, \text{-}) * (\alpha' \doteq p_0::\alpha_0) * p' \doteq p_0]$
$\quad p'.\texttt{Current} := \texttt{Left};$
$[(p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left}) * (\alpha' \doteq p_0::\alpha_0) * p' \doteq p_0]$
$\quad p' := p'.\texttt{Left};$
$[(p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left}) * (\alpha' \doteq p_0::\alpha_0) * p' \doteq l_0]$

Fig. 12. Proofs of Triples for `SWPush` and `DFTPush` in Separation Logic

rules in separation logic, and derive the following triples:

$$[p_0 \mapsto l_0, r_0, \text{-}, \text{-} * c \doteq c_0 * p \doteq p_0]\ \texttt{SWPush}\ [p_0 \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left} * c \doteq p_0 * p \doteq l_0]$$

$$[p_0 \mapsto l_0, r_0, \text{-}, \text{-} * \alpha \doteq \alpha_0 * p' \doteq p_0]\ \texttt{DFTPush}\ [p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left} * \alpha \doteq p_0::\alpha_0 * p' \doteq l_0]$$

Note that both of these triples only describe the local effects of `SWPush` and `DFTPush`: they specify how a node p_0 and variables are updated. The derivations of these triples in separation logic are given in Figure 12.

Then, we use the embedding rule to derive a local quadruple from these triples:

$$\dfrac{\dfrac{[p_0 \mapsto\!\!\!\!\!\!\diagup\, l_0, r_0, \text{-}, \text{-} * c \doteq c_0 * p \doteq p_0]}{\{(p_0 \mapsto l_0, r_0, \text{-}, \text{-} * c \doteq c_0 * p \doteq p_0)\}}\ \texttt{SWPush} \qquad \dfrac{[p_0 \mapsto\!\!\!\!\!\!\diagup\, l_0, r_0, \text{-}, \text{-} * \alpha' \doteq \alpha_0 * p' \doteq p_0]}{\{(p_0 \mapsto l_0, r_0, \text{-}, \text{-} * \alpha' \doteq \alpha_0 * p' \doteq p_0)\}}\ \texttt{DFTPush}}{\{(p_0 \mapsto l_0, r_0, \texttt{-}, \texttt{-} * c \doteq c_0 * p \doteq p_0)\ (p_0 \mapsto l_0, r_0, \texttt{-}, \texttt{-} * \alpha' \doteq \alpha_0 * p' \doteq p_0)\}}\ \texttt{SWPush}\ \texttt{DFTPush}$$

$$\left\{ \begin{pmatrix} p_0 \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left} * c \doteq p_0 * p \doteq l_0 \\ p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left} * \alpha' \doteq p_0 :: \alpha_0 * p' \doteq l_0 \end{pmatrix} \right\}$$

Finally, we apply the frame rule, auxiliary variable elimination, and Consequence. Using these rules, we transform the local quadruple to global one, eliminate auxiliary variables, and obtain the required quadruple:

$$\left\{ \left(\mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \texttt{Unmarked}, \texttt{-} \\ p' \mapsto l_0, r_0, \texttt{Unmarked}, \texttt{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha' \right) \wedge p = p' \right\}$$

$$\left\{ \exists \alpha_0 p_0 c_0.\, \mathsf{Same} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * \begin{pmatrix} p_0 \mapsto l_0, r_0, \text{-}, \text{-} * c \doteq c_0 * p \doteq p_0 \\ p_0 \mapsto l_0, r_0, \text{-}, \text{-} * \alpha' \doteq \alpha_0 * p' \doteq p_0 \end{pmatrix} \right\} \because \text{Lemma 6}$$

$$\alpha_0 p_0 c_0\text{:} \left[\begin{array}{l} \left\{ \mathsf{Same} * \mathsf{Stack}\ p_0\ c_0\ \alpha_0 * \begin{pmatrix} (p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * c \doteq c_0 * p \doteq p_0 \\ (p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * \alpha' \doteq \alpha_0 * p' \doteq p_0 \end{pmatrix} \right\} \\ \text{Framed:} \left[\begin{array}{l} \left\{ \begin{pmatrix} (p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * c \doteq c_0 * p \doteq p_0 \\ (p_0 \mapsto l_0, r_0, \text{-}, \text{-}) * \alpha' \doteq \alpha_0 * p' \doteq p_0 \end{pmatrix} \right\} \\ \texttt{SWPush} \qquad\qquad \texttt{DFTPush} \\ \left\{ \begin{pmatrix} (p_0 \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left}) * c \doteq p_0 * p \doteq l_0 \\ (p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left}) * \alpha' \doteq p_0 :: \alpha_0 * p' \doteq l_0 \end{pmatrix} \right\} \end{array} \right] \\ \left\{ \mathsf{Same} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * \begin{pmatrix} p_0 \mapsto\!\!\!\!\!\!\diagup\, c_0, r_0, \texttt{Marked}, \texttt{Left} * c \doteq p_0 * p \doteq l_0 \\ p_0 \mapsto\!\!\!\!\!\!\diagup\, l_0, r_0, \texttt{Marked}, \texttt{Left} * \alpha' \doteq p_0 :: \alpha_0 * p' \doteq l_0 \end{pmatrix} \right\} \end{array} \right]$$

$$\left\{ \exists \alpha_0 p_0 c_0.\, \mathsf{Same} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * \begin{pmatrix} p_0 \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left} * c \doteq p_0 * p \doteq l_0 \\ p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left} * \alpha' \doteq p_0 :: \alpha_0 * p' \doteq l_0 \end{pmatrix} \right\}$$

$\{(\mathsf{Same} * \mathsf{Stack}\, p\, c\, \alpha') \wedge p = p'\}$ $\because$ Lemma 6

Note that the above proof has a gap; when we apply Consequence, we use two implications between relations without proofs. We fill this gap by proving these implications.

Lemma 6 *The following implications hold:*

1. $$\left(\mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \\ p' \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha' \right) \wedge p{=}p'$$
   $$\Longrightarrow$$
   $$\exists \alpha_0 p_0 c_0.\, \mathsf{Same} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * \begin{pmatrix} p_0 \mapsto l_0, r_0, \text{-}, \text{-} * c \doteq c_0 * p \doteq p_0 \\ p_0 \mapsto l_0, r_0, \text{-}, \text{-} * \alpha' \doteq \alpha_0 * p' \doteq p_0 \end{pmatrix}$$

2. $$\exists \alpha_0 p_0 c_0.\, \mathsf{Same} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * \begin{pmatrix} p_0 \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left} * c \doteq p_0 * p \doteq l_0 \\ p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left} * \alpha' \doteq p_0 :: \alpha_0 * p' \doteq l_0 \end{pmatrix}$$
   $$\Longrightarrow$$
   $$(\mathsf{Same} * \mathsf{Stack}\, p\, c\, \alpha') \wedge p{=}p'$$

PROOF. The following derivation shows the first implication:

$$\left(\mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \\ p' \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha' \right) \wedge p = p'$$
$$\Longrightarrow \quad \because p = p'$$
$$\left(\mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \\ p \mapsto l_0, r_0, \texttt{Unmarked}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha' \right) \wedge p{=}p'$$
$$\Longrightarrow \quad \because (E_0 \mapsto E_1) \Rightarrow (E_0 \mapsto \text{-})$$
$$\left(\mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \text{-}, \text{-} \\ p \mapsto l_0, r_0, \text{-}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha' \right) \wedge p{=}p'$$
$$\Longrightarrow \quad \because (R \wedge E_1 = E_2) \Leftrightarrow (R * E_1 \doteqdot E_2)$$
$$\mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \text{-}, \text{-} \\ p \mapsto l_0, r_0, \text{-}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha' * p \doteqdot p'$$

$\Longrightarrow$

$\exists \alpha_0 p_0 c_0.\ \mathsf{Same} * \begin{pmatrix} p \mapsto l_0, r_0, \text{-}, \text{-} \\ p \mapsto l_0, r_0, \text{-}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p\, c\, \alpha' * p \ddot{=} p' * c \ddot{=} c_0 * \alpha' \ddot{=} \alpha_0 * p \ddot{=} p_0$

$\Longrightarrow \quad \because c = c_0,\ \alpha' = \alpha_0,\ \text{and } p = p_0$

$\exists \alpha_0 p_0 c_0.\ \mathsf{Same} * \begin{pmatrix} p_0 \mapsto l_0, r_0, \text{-}, \text{-} \\ p_0 \mapsto l_0, r_0, \text{-}, \text{-} \end{pmatrix} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * p_0 \ddot{=} p' * c \ddot{=} c_0 * \alpha' \ddot{=} \alpha_0 * p \ddot{=} p_0$

$\Longrightarrow \quad \because \begin{pmatrix} P \\ Q * E \doteq E' \end{pmatrix} \Leftrightarrow \left(\begin{pmatrix} P \\ Q \end{pmatrix} * E \ddot{=} E' \right) \Leftrightarrow \begin{pmatrix} P * E \doteq E' \\ Q \end{pmatrix}$

$\exists \alpha_0 p_0 c_0.\ \mathsf{Same} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * \begin{pmatrix} p_0 \mapsto l_0, r_0, \text{-}, \text{-} * c \doteq c_0 * p \doteq p_0 \\ p_0 \mapsto l_0, r_0, \text{-}, \text{-} * \alpha' \doteq \alpha_0 * p' \doteq p_0 \end{pmatrix}$

The following derivation shows the second implication:

$\exists \alpha_0 p_0 c_0.\ \mathsf{Same} * \mathsf{Stack}\, p_0\, c_0\, \alpha_0 * \begin{pmatrix} p_0 \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left} * c \doteq p_0 * p \doteq l_0 \\ p_0 \mapsto l_0, r_0, \texttt{Marked}, \texttt{Left} * \alpha' \doteq p_0 :: \alpha_0 * p' \doteq l_0 \end{pmatrix}$

$\Longrightarrow \quad \because c = p_0 \text{ and } p = l_0$

$\exists \alpha_0 c_0.\ \mathsf{Same} * \mathsf{Stack}\, c\, c_0\, \alpha_0 * \begin{pmatrix} c \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left} \\ c \mapsto p, r_0, \texttt{Marked}, \texttt{Left} * \alpha' \doteq c :: \alpha_0 * p' \doteq p \end{pmatrix}$

$\Longrightarrow \quad \because \begin{pmatrix} P \\ Q * E \doteq E' \end{pmatrix} \Leftrightarrow \left(\begin{pmatrix} P \\ Q \end{pmatrix} * E \ddot{=} E' \right)$

$\exists \alpha_0 c_0.\ \mathsf{Same} * \left(\mathsf{Stack}\, c\, c_0\, \alpha_0 * \begin{pmatrix} c \mapsto c_0, r_0, \texttt{Marked}, \texttt{Left} \\ c \mapsto p, r_0, \texttt{Marked}, \texttt{Left} \end{pmatrix} \right) * \alpha' \ddot{=} c :: \alpha_0 * p' \ddot{=} p$

$\Longrightarrow \quad \because \text{the definition of } \mathsf{Stack}\ p\ c\ (c :: \alpha_0)$

$\exists \alpha_0.\ \mathsf{Same} * \mathsf{Stack}\ p\ c\ (c :: \alpha_0) * \alpha' \ddot{=} c :: \alpha_0 * p' \ddot{=} p$

$\Longrightarrow \quad \because \alpha' = c :: \alpha$

$\mathsf{Same} * \mathsf{Stack}\ p\ c\ \alpha' * p' \ddot{=} p$

$\Longrightarrow \quad \because (R \wedge E_1 = E_2) \Leftrightarrow (R * E_1 \ddot{=} E_2)$

$(\mathsf{Same} * \mathsf{Stack}\ p\ c\ \alpha') \wedge p' = p$

□

7 Nondeterministic Allocator and Hoare Quadruples

Separation logic assumes a completely nondeterministic allocator, in order to ensure that a proof in separation logic is valid no matter how the allocator

`cons` is implemented. However, this seemingly innocent assumption makes it difficult to find a proper definition of quadruples when programs can call `cons`. In particular, our current definition invalidates the frame rule. In this section, we explain this difficulty in details, and present a definition of quadruples for nondeterministic programs that conservatively extends the old definition of quadruples in Section 5 and validates all the rules in our logic.

When the allocator `cons` is completely nondeterministic, a program can use `cons` to detect that a variable y stores a dangling pointer. For instance, consider the following command:

$$x := \texttt{cons}(0); \texttt{free}(x); \texttt{if } (x = y) \texttt{ then div else skip}$$

This command behaves differently depending on whether the variable y stores a dangling pointer. If a pointer y is dangling initially, $x := \texttt{cons}(0)$ can allocate a cell at this location y. Thus, the following test $x = y$ can succeed or fail, so the command can diverge or terminate. On the other hand, if a pointer y is not dangling initially, the command cannot diverge. All the cells that $x := \texttt{cons}(0)$ can pick are different from the already allocated cell y. Thus, the following test $x = y$ fails, and the command always terminates.

Unfortunately, this ability of detecting a dangling pointer makes a naive definition of a quadruple fail to validate the frame rule. Suppose that we use Definition 3 even for programs that can call `cons`. Then, the frame rule is not sound any more. Consider a quadruple:

$$\{\mathsf{Emp} \wedge \mathsf{Pt}(y, y')\} \begin{array}{l} x{:=}\texttt{cons}(0); \texttt{free}(x); \texttt{if } x{=}y \texttt{ then div else skip} \\ x'{:=}\texttt{cons}(0); \texttt{free}(x'); \texttt{if } x'{=}y' \texttt{ then div else skip} \end{array} \{\mathsf{Emp}\}$$

where $\mathsf{Pt}(y, y')$ is an abbreviation of $y > 0 \wedge y' > 0$, which means that y and y' contain pointer values. Under the current definition, this quadruple holds, because when the first and second commands are started in the empty heap, both of them can diverge or terminate. Now, we add an invariant

$$\begin{pmatrix} y \mapsto \text{-} \\ \\ \mathsf{emp} \end{pmatrix}$$

to this quadruple, using the frame rule:

$$\left\{ (\mathsf{Emp} \wedge \mathsf{Pt}(y, y')) * \begin{pmatrix} y \mapsto \text{-} \\ \mathsf{emp} \end{pmatrix} \right\}$$

$x := \texttt{cons}(0);$	$x' := \texttt{cons}(0);$
$\texttt{free}(x);$	$\texttt{free}(x');$
$\texttt{if } (x = y) \texttt{ then div}$	$\texttt{if } (x' = y') \texttt{ then div}$
$\texttt{else skip}$	$\texttt{else skip}$

$$\left\{ \mathsf{Emp} * \begin{pmatrix} y \mapsto \text{-} \\ \mathsf{emp} \end{pmatrix} \right\}$$

The resulting quadruple is not valid. The pre-relation of this quadruple relates a heap h to the empty heap [] if h is a singleton heap containing cell y. However, for such a heap h and the empty heap [], the first and second commands behave differently. When the first command is started from such h, it always terminates, but when the second command is started from [], it can diverge or terminate. This disagreement in the behaviors of the two commands violates the second "equal divergence" condition in Definition 3, and shows that the quadruple does not hold. Thus, this example illustrates that the frame rule is not sound when we interpret a quadruple naively.

Our solution is to add to Definition 3 a requirement that the relationship between two programs should not depend on which pointers are dangling. The new definition of a quadruple is given below:

Definition 7 *For commands C and C', a quadruple*

$$\{R\} \begin{matrix} C \\ C' \end{matrix} \{S\}$$

holds if and only if for all stores s and heaps h and h' such that $(s, h, h') \models R$,

(1) (s, h, C) is safe and (s, h', C') is safe; and
(2) for all heaps h_0 and h'_0 such that $h \# h_0$ and $h' \# h'_0$, $(s, h * h_0, C)$ can diverge if and only if $(s, h' * h'_0, C')$ can diverge; and
(3) for all heaps h_0 and h'_0 such that $h \# h_0$ and $h' \# h'_0$, if $(s, h * h_0, C) \rightsquigarrow^* (t, h_1)$ and $(s, h * h'_0, C') \rightsquigarrow^* (t', h'_1)$, then there exist $h_2, h'_2 \in \mathsf{Heaps}$ such that

$$h_2 * h_0 = h_1 \ \wedge \ h'_2 * h'_0 = h'_1 \ \wedge \ (s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_2,\ h'_2) \models S.$$

In this definition of quadruples, the second and third conditions are strengthened. The conditions compare computations of C and C', not just from R-related heaps, but also from all possible extensions of them. The second condition requires that the computations of C and C' from these extended heaps have the same divergence behavior, and the third condition that such computations of C and C' produce heaps with S-related subparts. Note that extending a heap makes some dangling pointers nondangling. Thus, by considering all the extended heaps, the conditions ask that C and C' should transform R-related heaps to S-related heaps or divergence, no matter how many dangling pointers become nondangling. In this way, they ensure that the relationship of C and C' does not depend on dangling pointers.

Note that the strengthened conditions in the new definition of quadruples exclude the previous counterexample for the frame rule. The reason is that the problematic quadruple

$$\{\mathsf{Emp}\wedge\mathsf{Pt}(y,y')\}\ \begin{matrix} x{:=}\texttt{cons}(0);\texttt{free}(x);\texttt{if } x{=}y \texttt{ then div else skip} \\ x'{:=}\texttt{cons}(0);\texttt{free}(x');\texttt{if } x'{=}y' \texttt{ then div else skip} \end{matrix}\ \{\mathsf{Emp}\}$$

does not hold according to the new definition. When s maps y and y' to 1 and both h and h' are the empty heap, (s,h,h') satisfies the pre-relation Emp. However, the second "equal divergence" condition in Definition 7 does not hold for the extensions $[1\mapsto 0]$ and $[]$ of h and h': when the two commands are executed, respectively, from $(s,[1\mapsto 0])$ and $(s,[])$, only the second command can diverge.

The new definition of quadruples, indeed, solves the problem about `cons` and the frame rule, which is discussed in the beginning of this section. That is, as long as quadruples are interpreted by this new definition, the frame rule and all the other proof rules for the quadruples in our logic are sound, even when commands call `cons`. Before proving this main property of the definition, we simplify the new definition of quadruples slightly using the locality properties of commands.

Lemma 8 *For all commands C and C', a quadruple*

$$\{R\}\begin{matrix} C \\ C' \end{matrix}\{S\}$$

holds according to Definition 7 if and only if for all stores s and heaps h and h' such that $(s,h,h')\models R$,

(1) (s,h,C) is safe and (s,h',C') is safe; and
(2) for all heaps h_0 and h'_0 such that $h\#h_0$ and $h'\#h'_0$, $(s,h*h_0,C)$ can diverge if and only if $(s,h'*h'_0,C')$ can diverge; and

(3) if $(s,h,C) \rightsquigarrow^* (t,h_1)$ *and* $(s,h',C') \rightsquigarrow^* (t',h'_1)$*, then*

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_1,\ h'_1) \ \models\ S.$$

Note that the lemma simplifies the third condition in Definition 7; the new third condition in the lemma does not quantify over all heap extensions, and just considers the R-related heaps h and h'.

PROOF. Since the third condition in this lemma is weaker than the third condition in Definition 7, it suffices to show that the characterization of quadruples in the lemma implies Definition 7. We will show that if (s,h,C) and (s,h',C') are safe, then the third condition in the lemma implies the third condition in Definition 7. Consider safe configurations (s,h,C) and (s,h',C'), and heaps h_0 and h'_0 such that $h\#h_0$ and $h'\#h'_0$. Suppose that $(s,h*h_0,C) \rightsquigarrow^* (t,h_1)$ and $(s,h'*h'_0,C') \rightsquigarrow^* (t',h'_1)$. Then, by the frame property of commands, there exist h_2 and h'_2 such that

$$h_1 = h_2 * h_0,\ \ h'_1 = h'_2 * h'_0,\ \ (s,h,C) \rightsquigarrow^* (t,h_2), \quad \text{and} \quad (s,h',C') \rightsquigarrow^* (t',h'_2).$$

Now, the condition in the lemma implies that

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_2,\ h'_2) \models S.$$

Thus, the heaps h_2 and h'_2 are what the third condition in Definition 7 requires. □

We now prove the main theorem of this section.

Theorem 9 (Soundness) *Suppose that quadruples are interpreted by Definition 7. Then, all the proof rules in Figure 5 are sound, even when commands call the nondeterministic allocator* `cons`*.*

Note that this theorem does not require any changes in the semantics of relations R, S in Section 4. Thus, this theorem also ensures that we can use all the proof rules for relations in Section 4 while proving a quadruple.

PROOF. We will consider the embedding rule, the frame rule and Consequence only. Suppose that triples $[P]C[Q]$ and $[P']C'[Q']$ hold, and that commands C, C' and assertions P, P', Q, Q' satisfy the side condition for the embedding rule. We will show that the quadruple

$$\left\{\begin{pmatrix} P \\ P' \end{pmatrix}\right\}\ \begin{matrix} C \\ C' \end{matrix}\ \left\{\begin{pmatrix} Q \\ Q' \end{pmatrix}\right\}$$

holds. Consider a store s and heaps h, h' such that $(s, h, h') \models \binom{P}{P'}$. Then, $(s, h) \models P$ and $(s, h') \models P'$. Since $[P]C[Q]$ and $[P']C'[Q']$ hold, (s, h, C) and (s, h', C') are safe. To show the second condition in Definition 7, consider heaps h_0 and h'_0 such that $h_0 \# h$ and $h'_0 \# h'$. Because of termination monotonicity, neither $(s, h * h_0, C)$ nor $(s, h' * h'_0, C')$ can diverge, and thus, the second condition holds. For the third condition in Definition 7, suppose that $(s, h, C) \rightsquigarrow^* (t, h_1)$ and $(s, h', C') \rightsquigarrow^* (t', h'_1)$. Then, since $[P]C[Q]$ and $[P']C'[Q']$ hold, $(s, h) \models P$, and $(s, h') \models P'$, we have that $(t, h_1) \models Q$ and $(t', h'_1) \models Q'$. Now, because $\mathsf{Mod}(C') \cap \mathsf{FV}(Q) = \emptyset$ and $\mathsf{Mod}(C) \cap \mathsf{FV}(Q') = \emptyset$, we have

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_1,\ h'_1) \models \binom{Q}{Q'}.$$

For the frame rule, suppose that a quadruple

$$\{R\}\begin{matrix} C \\ C' \end{matrix}\{S\}$$

holds and that $\mathsf{FV}(R_1) \cap \mathsf{Mod}(C, C') = \emptyset$. Consider a store s and heaps h, h' such that $(s, h, h') \models R * R_1$. Then, there exist splittings $m * n = h$ and $m' * n' = h'$ such that $(s, m, m') \models R$ and $(s, n, n') \models R_1$. Because of safety monotonicity, the first condition about the safety of (s, h, C) and (s, h', C') follows from the safety of (s, m, C) and (s, m', C'). For the second condition, consider heaps h_0 and h'_0 such that $h_0 \# h$ and $h'_0 \# h'$. Then, $h_0 * n$ and $h'_0 * n'$ are, respectively, disjoint from m and m'. Thus, the second condition in the definition of $\{R\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S\}$ implies that $(s, h_0 * n * m, C)$ can diverge if and only if $(s, h'_0 * n' * m', C')$ can diverge. For the third condition, suppose that $(s, h, C) \rightsquigarrow^* (t, h_1)$ and $(s, h', C') \rightsquigarrow^* (t', h'_1)$. Then, by the frame property, there exist heaps h_2 and h'_2 such that

$$h_1 = h_2 * n, \quad h'_1 = h'_2 * n', \quad (s, m, C) \rightsquigarrow^* (t, h_2), \quad \text{and} \quad (s, m', C') \rightsquigarrow^* (t', h'_2).$$

Moreover, since $\{R\}\begin{smallmatrix} C \\ C' \end{smallmatrix}\{S\}$ and $(s, m, m') \models R$, we have that

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_2,\ h'_2) \models S.$$

Note that the side condition about the modified variables ensures that R_1 is satisfied by $(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ n,\ n')$. Thus,

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_2 * n,\ h'_2 * n') \models S * R_1.$$

Finally, we consider the soundness of Consequence. Suppose that a quadruple

$$\{R_1\}\begin{matrix}C\\C'\end{matrix}\{S_1\}$$

holds, and that $R \Rightarrow R_1$ and $S_1 \Rightarrow S$. In this case, we need show the validity of the quadruple:

$$\{R\}\begin{matrix}C\\C'\end{matrix}\{S\}$$

Consider a store s and heaps h, h' such that $(s, h, h') \models R$. Since $R \Rightarrow R_1$, (s, h, h') also satisfies the pre-relation R_1 of the quadruple $\{R_1\}^{C}_{C'}\{S_1\}$. Since the quadruple $\{R_1\}^{C}_{C'}\{S_1\}$ is valid, we have the following three facts by Lemma 8:

(1) Both (s, h, C) and (s, h, C') are safe.
(2) For all heaps h_1 and h'_1 such that $h_1 \# h$ and $h'_1 \# h'$, $(s, h * h_1, C)$ can diverge if and only if $(s, h' * h'_1, C')$ can diverge.
(3) If $(s, h, C) \rightsquigarrow^* (t, h_2)$ and $(s, h', C') \rightsquigarrow^* (t', h'_2)$, then

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_2,\ h'_2) \models S_1.$$

The first two facts show that (s, h, h') satisfies the first and second conditions for $\{R\}^{C}_{C'}\{S\}$, and the third fact implies that (s, h, h') satisfies the third condition for $\{R\}^{C}_{C'}\{S\}$ in Lemma 8, because $S_1 \Rightarrow S$. □

Before ending this section, we provide two justifications of the new definition of quadruples. First, we prove that the new definition of quadruples is a conservative extension of the old one: for deterministic commands, the old and new definitions of quadruples are equivalent, so the strengthened parts in the new definition — the quantifications over extended heaps — play a role only for nondeterministic programs. Second, we show that the new definition of quadruples is the "weakest" definition that validates the frame rule.

Proposition 10 *For deterministic commands, the definition of quadruples in Definition 7 coincides with the definition of quadruples in Definition 3.*

PROOF. Since Definition 7 is stronger than the other definition, it suffices to show that Definition 3 implies Definition 7. Moreover, since the first conditions in the two definitions are identical and the third conditions are equivalent

(Lemma 8), we can focus on the second condition in Definition 7. Pick deterministic commands C and C', a Hoare quadruple $\{R\}^{C}_{C'}\{S\}$, and (s, h, h') such that

(1) $(s, h, h') \models R$, and
(2) the quadruple $\{R\}^{C}_{C'}\{S\}$ holds according to Definition 3.

Consider heaps h_0 and h'_0 such that $h_0 \# h$ and $h'_0 \# h'$. We will show that $(s, h * h_0, C)$ can diverge if and only if $(s, h' * h'_0, C')$ can diverge. Suppose that $(s, h * h_0, C)$ can diverge. Then, by termination monotonicity, (s, h, C) can also diverge. Now, the second condition in Definition 3 implies that (s, h', C') can diverge, too. This divergence of (s, h', C') forces $(s, h' * h'_0, C')$ to diverge; if $(s, h' * h'_0, C')$ cannot diverge, and thus terminates in some state, the frame property implies that (s, h', C') also terminates, but this means that a deterministic command can behave nondeterministically. The other implication about divergence can be proved similarly. □

Proposition 11 *Suppose that we are given some interpretation of a quadruple* $\{R\}^{C}_{C'}\{S\}$ *that validates the frame rule, and implies the following: for all stores* s *and heaps* h *and* h' *that satisfy pre-relation* R *(i.e.,* $(s, h, h') \models R$*),*

(1) (s, h, C) *is safe and* (s, h', C') *is safe;*
(2) (s, h, C) *can diverge if and only if* (s, h', C) *can diverge; and*
(3) if $(s, h, C) \rightsquigarrow^* (t, h_1)$ *and* $(s, h', C') \rightsquigarrow^* (t', h'_1)$*, then*

$$(s[t|_{\mathsf{FV}(C)}][t'|_{\mathsf{FV}(C')}],\ h_1,\ h'_1) \models S.$$

Then, if a quadruple holds in this interpretation, then it holds in Definition 7.

PROOF. Suppose that we are given an interpretation of quadruples that satisfies the requirements in this proposition, and that a quadruple $\{R\}^{C}_{C'}\{S\}$ holds in this given interpretation. We need to show that $\{R\}^{C}_{C'}\{S\}$ holds in Definition 7. By Lemma 8, it suffices to show that for all stores s and heaps h, h', if $(s, h, h') \models R$, the second condition in Definition 7 holds. Consider a store s and heaps h, h', h_1, h'_1 such that $(s, h, h') \models R$, $h \# h_1$, and $h' \# h'_1$. We first construct assertion P and P' that precisely describe the heaps h_1 and h_2: $(s_0, h_0) \models P$ if and only if $h_0 = h_1$; and $(s_0, h_0) \models P'$ if and only if $h_0 = h'_1$. The assertion P is the separating conjunction of $(n \mapsto h_1(n))$ for all n in $\mathsf{dom}(h_1)$, and P' the separating conjunction of $(n \mapsto h'_1(n))$ for all n in

$\mathsf{dom}(h'_1)$. We now add $\binom{P}{P'}$ to the quadruple $\{R\}^{C}_{C'}\{S\}$, and obtain

$$\left\{R * \binom{P}{P'}\right\}\ {\scriptstyle C}\ {\scriptstyle C'}\ \left\{S * \binom{P}{P'}\right\}.$$

This quadruple holds in the given interpretation, because the frame rule is sound in the interpretation. The pre-relation of this quadruple is satisfied by $(s, h * h_1, h' * h'_1)$ because $(s, h, h') \models R$ and $(s, h_1, h'_1) \models \binom{P}{P'}$. Thus, by the requirement in the proposition, $(s, h * h_1, C)$ can diverge if and only if $(s, h' * h'_1, C')$ can diverge. This equivalence is precisely what we are required to show. □

8 Conclusion

In this paper, we developed a Hoare-style logic for verifying a relationship between two programs, and applied the resulting logic to show the equivalence between the Schorr-Waite graph marking algorithm and the depth-first traversal. The main features of our logic are Hoare quadruples, separating conjunction for relations, and the frame rule for quadruples. Hoare quadruples simplify formal specification, by allowing the equivalence specification; and separating conjunction and the frame rule simplify formal verification, by supporting local reasoning about quadruples, and allowing the "smooth" inclusion of separation logic. Indeed, our verification of the Schorr-Waite algorithm fully used all these features, and resulted in a formal proof simpler than the other existing proofs of the algorithm, including the one in separation logic [8].

Our work is closely related to Necula's translation validation [17,19], Rinard's credible compilation [20,18], and Zuck *et al*'s "VOC" [21]. The goal of these three is to ensure the correctness of compiler optimization by showing the equivalence between source code and optimized code. To achieve this goal, all these approaches provide methods for specifying a relationship between two flowchart programs, and for showing a specified relationship. The basic ideas of these methods are similar to those of our logic. However, because of a different motivation, all of the three approaches consider very limited languages for relations, and use complicated methods to handle pointers. Thus, they are not appropriate for comparing two programs which use very different data structures with pointers, such as the depth-first traversal and the Schorr-Waite algorithm. On the other hand, they can handle programs with very different control structures. They allow relations between any two points of flowchart programs, and provide a proof rule for verifying these relations.

We are considering to adopt their proof rule in our logic, in order to handle programs with different control structures.

Benton independently developed almost the same logic as ours [22], which he called relational Hoare logic. For deterministic commands, his typing judgment means the same thing as our Hoare quadruple except that his judgment does not require the safety. His proof rules for loops, conditional statements, and sequential composition also coincide with ours. The main difference is that he does not consider pointers seriously. Thus, the relationship between his work and ours is similar to that of Hoare logic and separation logic. In fact, the name of our logic comes from this comparison.

The work on data refinement is also closely related [10–13,23]. Especially, we are influenced by Reynolds's refinement proofs of graph algorithms in [13]. Our work can be seen as a first step to extend Reynolds's idea to pointer programs. However, to explain data refinement properly, we need to change the logic in two aspects. First, we should treat nondeterminism differently, so that we can explain refinement rather than equivalence in our logic. Second, we should find a proof rule for showing the equivalence of the two implementations of an abstract data type, not the equivalence of specific clients of them.

Acknowledgments. When I was a PhD student, my thesis advisor suggested me to read John Reynolds's book, "The Craft of Programming", and formalize the last chapter of the book. It is this reading of Reynolds's book that started the work presented in this paper. Reynolds also gave me insightful comments about the verification of the Schorr-Waite graph marking algorithm and the problem of nondeterministic allocator, which played a crucial role for obtaining the results in this paper. I have benefited greatly from discussions with Peter O'Hearn, Uday Reddy, David Naumann, Josh Berdine, Richard Bornat, Cristiano Calcagno, Noah Torp-Smith, and Ivana Mijajlović. The anonymous referees made helpful suggestions.

References

[1] J. C. Reynolds, Intuitionistic reasoning about shared mutable data structure, in: Millennial Perspectives in Computer Science, Palgrave, 2000.

[2] S. Ishtiaq, P. W. O'Hearn, BI as an assertion language for mutable data structures, in: Proceedings of the 28th ACM Symposium on Principles of Programming Languages, ACM, 2001, pp. 14–26.

[3] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Proceedings of the 17th IEEE Symposium on Logic in Computer Science, IEEE, 2002, pp. 55–74.

[4] P. W. O'Hearn, J. C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: L. Fribourg (Ed.), Proceedings of 15th Annual Conference of the European Association for Computer Science Logic, Vol. 2142 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 1–19.

[5] J. C. Reynolds, Lectures on reasoning about shared mutable data structure, IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic, Tandil, Argentina, September 6-13 (2000).

[6] R. Bornat, C. Calcagno, P. W. O'Hearn, Local reasoning, separation and aliasing, in: Proceedings of the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management, Venice, 2004.

[7] L. Birkedal, N. Torp-Smith, J. C. Reynolds, Local reasoning about a copying garbage collector, in: Proceedings of the 31th ACM Symposium on Principles of Programming Languages, ACM, Venice, 2004, pp. 220–231.

[8] H. Yang, Local reasoning for stateful programs, Ph.D. thesis, University of Illinois at Urbana-Champaign, (Technical Report UIUCDCS-R-2001-2227) (2001).

[9] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: principles, techniques, and tools, Addison-Welsley, 1988.

[10] J. He, C. A. R. Hoare, J. W. Sanders, Data refinement refined, in: European Symposium on Programming, Vol. 213 of Lecture Notes in Computer Science, Springer-Verlag, 1986, pp. 187–196.

[11] C. A. R. Hoare, J. He, J. W. Sanders, Prespecification in data refinement, Information Processing Letter 25 (2) (1987) 71–76.

[12] P. H. B. Gardiner, C. C. Morgan, A single complete rule for data refinement, Formal Aspects of Computing 5 (1993) 367–382, (Reprinted in [24]).

[13] J. C. Reynolds, The Craft of Programming, Prentice-Hall International, London, 1981.

[14] H. Yang, P. W. O'Hearn, A semantic basis for local reasoning, in: Proceedings of the 5th Conference on Foundations of Software Science and Computation Structures, Vol. 2303 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 402–416.

[15] P. W. O'Hearn, D. J. Pym, The logic of bunched implications, Bulletin of Symbolic Logic 5 (2) (99) 215–244.

[16] C. Calcagno, H. Yang, P. W. O'Hearn, Computability and complexity results for a spatial assertion language for data structures, in: Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science, Vol. 2245 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 108–119.

[17] G. Necula, Translation validation, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2000.

[18] M. Rinard, Credible compilation, Technical Report MIT-LCS-TR-776, MIT Laboratory for Computer Science (Mar. 1999).

[19] C. Colby, P. Lee, G. Necula, F. Blau, K. Cline, M. Plesko, Certifying compiler for java, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2000.

[20] M. Rinard, D. Marinov, Credible Compilation with Pointers, in: Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, Italy, 1999.

[21] L. Zuck, A. Pnueli, F. Yi, B. Goldberg, Voc: A translation validator for optimizing compilers, Electronic Notes in Theoretical Computer Science 65 (2).

[22] N. Benton, Simple relational correctness proofs for static analyses and program transformations, in: Proceedings of the 31th ACM Symposium on Principles of Programming Languages, ACM, Venice, 2004, pp. 14–25.

[23] J.-R. Abrial, FME 2003: Formal methods, international symposium of formal methods europe, in: Event Based Sequential Program Development: Application to Constructing a Pointer Program, Vol. 2805 of Lecture Notes in Computer Science, Springer, 2003, pp. 51–74.

[24] C. Morgan, T. Vickers (Eds.), On the Refinement Calculus, Springer-Verlag, 1992.

A Operational Semantics

$$\frac{}{x := E, s, h \rightsquigarrow (s \mid x \mapsto [\![E]\!]s), h}$$

$$\frac{}{X := G, s, h \rightsquigarrow (s \mid X \mapsto [\![G]\!]s), h} \quad \frac{}{\alpha := L, s, h \rightsquigarrow (s \mid \alpha \mapsto [\![L]\!]s), h}$$

$$\frac{[\![E]\!]s = n}{\texttt{free}(E), s, h * [n \mapsto m] \rightsquigarrow s, h} \quad \frac{[\![E]\!]s = n \quad n \notin \mathsf{dom}(h)}{\texttt{free}(E), s, h \rightsquigarrow \texttt{fault}}$$

$$\frac{[\![E]\!]s = n \in \mathsf{dom}(h) \quad h(n) = m}{x := [E], s, h \rightsquigarrow (s \mid x \mapsto m), h} \quad \frac{[\![E]\!]s \notin \mathsf{dom}(h)}{x := [E], s, h \rightsquigarrow \texttt{fault}}$$

$$\frac{[\![E_1]\!]s = n \in \mathsf{dom}(h)}{[E_1] := E_2, s, h \rightsquigarrow s, (h \mid n \mapsto [\![E_2]\!]s)} \quad \frac{[\![E_1]\!]s \notin \mathsf{dom}(h)}{[E_1] := E_2, s, h \rightsquigarrow \texttt{fault}}$$

$$\frac{m, ..., n + m - 1 \notin \mathsf{dom}(h) \quad v_1 = [\![E_1]\!]s, ..., v_n = [\![E_n]\!]s}{x := \texttt{cons}(E_1, ..., E_n), s, h \rightsquigarrow (s \mid x \mapsto m), (h * [m \mapsto v_1, ..., n + m - 1 \mapsto v_n])}$$

$$\frac{C_1, s, h \rightsquigarrow C_1', s', h'}{C_1; C_2, s, h \rightsquigarrow C_1'; C_2, s', h'} \quad \frac{C_1, s, h \rightsquigarrow s', h'}{C_1; C_2, s, h \rightsquigarrow C_2, s', h'} \quad \frac{C_1, s, h \rightsquigarrow \texttt{fault}}{C_1; C_2, s, h \rightsquigarrow \texttt{fault}}$$

$$\frac{[\![B]\!]s = \mathit{true}}{\texttt{if } B \texttt{ then } C \texttt{ else } C', s, h \rightsquigarrow C, s, h} \quad \frac{[\![B]\!]s = \mathit{false}}{\texttt{if } B \texttt{ then } C \texttt{ else } C', s, h \rightsquigarrow C', s, h}$$

$$\frac{[\![B]\!]s = \mathit{false}}{\texttt{while } B \texttt{ do } C \texttt{ od}, s, h \rightsquigarrow s, h} \quad \frac{[\![B]\!]s = \mathit{true} \quad (C; \texttt{while } B \texttt{ do } C \texttt{ od}), s, h \rightsquigarrow K}{\texttt{while } B \texttt{ do } C \texttt{ od}, s, h \rightsquigarrow K}$$