Proving Simpson's Four-Slot Algorithm Using Ownership Transfer

Shuling Wang
wsl@iist.unu.edu
, Xu Wang
wx@iist.unu.edu
and International Institute for Software Technology, The United Nations University, Macao

Abstract

Simpson's four-slot algorithm has been an instructive example in studying various assertional proof methods/logics geared towards shared variable concurrency. Previously, techniques like rely-guarantee, data refinement and resource separation have been applied to simplify the construction of its correctness proof. Still, an elegant, concise and insightful proof is elusive.

Recently with the new generation of logics coming of age which are, for the first time, equipped with ownership transfer, it becomes imperative to ask *to what extent can ownership transfer facilitate a nice proof of the algorithm.* Ownership transfer is especially promising here because the conflict resolution mechanism in the four-slot algorithm can be easily recast as an implementation based on ownership transfer.

1 Introduction

Reasoning about concurrent programs with interference (a.k.a. data race) is notoriously difficult. Traditionally, the well-known approach to such reasoning is that of Owicki-Gries [13] and the rely-guarantee methods [7, 19]. The rely-guarantee methods improve Owicki-Gries approach by advocating compositionality [8, 7] in the proof. The interpretation of compositionality varies in different works of assertional reasoning, but a widely accepted syntactical criterion is that: in a compositional proof the assertions annotating a thread should only refer to the local variables of the thread and the shared variables while its rely/guarantee conditions should only refer to the shared variables; using local variables private to other threads or auxiliary variables should be disallowed.

Simpson's four-slot algorithm [16] is an extreme example of programming with interference known as wait-free programs. Interference is abundant in the algorithm, but is cleverly constrained not to overflow the entire system. The algorithm actually *uses interference on one subset of shared variables to implement interference-free accesses on another subset of shared variables*.

Achieving a simple proof of the four-slot algorithm with assertional reasoning is not easy. It can become even harder if compositionality is also desirable. Building on top of rely-guarantee methods, people have added techniques like data refinement [6, 5, 9, 10] and resource separation [1, 3] to ease the construction of a nice proof. However, their results are mixed; an elegant, concise and insightful proof is still elusive.

On the other hand, an important intuition gathered in studying the four-slot algorithm is that the central mechanism for achieving interference-freedom on data slots can be recast as an implementation based on *ownership transfer*. It looks promising that the adoption of ownership transfer in recent logics might lead to a simpler and more intuitive proof.

Ownership transfer in concurrent systems is an important research topic. Previous work on such reasoning are for interference-free systems with mutual exclusion, e.g. concurrent separation logic [2, 12]. *RGSep logic* [18, 17], which is a combination of concurrent separation logic and the rely-guarantee method, is one of the first works to extend such reasoning to concurrent systems with interference. Moreover, the authors in [10] have proposed the challenge of fully exploiting ownership transfer in proving Simpson's algorithm using these logics.

In this paper based on RGSep logic we will test to what extent can ownership transfer facilitate the construction of a nice proof. Furthermore, we will raise the bar slightly by trying to find a syntactically compositional proof. Note that a simple proof sometimes goes against a syntactically compositional proof. But the process of constructing a syntactically compositional proof can force us to use ownership transfer in RGSep to extreme so that we can 1) explore the expressiveness boundary of ownership transfer and 2) expose limitations in current implementation of ownership transfer in the logic.

Our effort is only partially successful. Firstly we confirm that ownership transfer can indeed improve the proof though not to a great extent. Then we realise that the existing ownership transfer facility in RGSep cannot directly lead to a syntactically compositional proof. After modifying the interpretation of the *local guard* extension in RGSep and the corresponding stability to be conditional on an invariant over shared states, we manage to construct a syntactically compositional proof outline eventually.

The rest of the paper is organized as follows. Section 2 introduces Simpson's four-slot algorithm and Section 3 introduces RGSep logic. In Section 4, we define action sets for the reader and writer after explaining the modification required on RGSep, based on which Section 5 gives a syntactical compositional proof outline of the algorithm, which is also shown to be reducible to a valid RGSep proof after giving up syntactical compositionality. Finally, Section 6 discusses related works and concludes the paper.

2 Simpson's Four-Slot Algorithm

Simpson's algorithm, as shown in Figure 1, uses four control bits (l, r, $li[0]$ and $li[1]$) and two pairs of slots ($d[0][0]$ and $d[0][1]$, and $d[1][0]$ and $d[1][1]$) as *shared variables* to achieve asynchronous communication between two threads: the reader and the writer.

The body of the reader is a loop that repeatedly calls procedure $read()$ while the body of the writer repeatedly calls procedure $write(w)$. Commands of the form $\langle c \rangle$ in the code of $read()$ and $write(w)$ are atomic commands in which c will be executed in one indivisible step. The four control bits are *atomic registers* [11]: operations like writing and reading on them are atomic commands. The two pairs of slots are *safe registers* [11]: they obtain the correct values if they are not concurrent with any write; their writing and reading are not atomic commands.

In addition to shared variables $read()$ and $write(w)$ also use *local variables*, e.g. wp and ri. However, the operations on them are invisible to the other thread and can thus coalesce with adjacent operations in the control flow without affecting atomicity.

The cleverness of Simpson's algorithm lies in that the reader and writer can coordinate, via the four atomic control bits, to channel simultaneous requests on the slots to different copies. Thus the accesses to one slot will always be serial and safe registers suffice to implement it.

In $write(w)$ the local variables wp and wi act as pointers pointing to resp. a pair and a slot in the pair. Thus $d[wp][wi]$ is the slot the writer is going to write to. The values of these pointers come from the values of control bits r and li. r is a pointer used by the reader to publish the pair it is going to read from, while the two members of li point to the fresh slots resp. in each of the two pairs, i.e. the slot holding fresher value than the other from the same pair.

The strategy of the writer, upon each invocation of $write(w)$, is to move away from the pair the reader is working on and then select the other slot (than the fresh slot) to write to. After writing to the slot, the writer updates the relevant pointer in li to point to the new freshest and publishes its latest location, i.e. the pair it just worked on, in l.

Similarly, in $read()$ rp and ri point to the slot the reader is going to read from. The strategy of the reader is to track the latest location of the writer by reading l and read the freshest value in the location as pointed to by li. However, notice that the reader updates r before the slot read starts, while the writer

```
local d[2][2] = ((0,0),(0,0)), li[2] = (0,0), l = 0, r = 0 in
write(w) =                                  | read() =
    local wp, wi;                           |     local y, rp, ri;
    ⟨wp := 1 − r;⟩                (1)       |     ⟨rp := l;⟩                (6)
    ⟨wi := 1 − li[wp];⟩           (2)       |     ⟨r := rp;⟩                (7)
    d[wp][wi] := w;               (3)       |     ⟨ri := li[rp];⟩           (8)
    ⟨li[wp] := wi;⟩               (4)       |     y := d[rp][ri];           (9)
    ⟨l := wp;⟩                    (5)       |     return y;
```

Figure 1: The four-slot algorithm

updates l and li after the slot write is finished. The order in which the control variables are updated in each thread is very crucial for achieving asynchronous communication.

Two important properties are guaranteed by Simpson's asynchronous communication mechanism. The first property is *data coherence*, which means that the writer and the reader never access the same slot at the same time [1]. Data coherence is mainly achieved in the writer's strategy. The second property is *data freshness*, which means that the reader always reads the value which is most recently written by the writer. Data freshness is mainly achieved in the reader's strategy.

3 RGSep Logic and Ownership Transfer

Separation logic Separation logic [15, 14] aims to reason about pointer programs that manipulate mutable data structures. As an extension to Hoare logic, separation logic introduces new logical operators for describing the heap structures. $\mathbf{emp}$ asserts that the heap is empty. The points-to relation $E \mapsto F$, where E and F are both expressions, asserts that the heap contains only one cell, with address E and content F. Assume P and Q are both separation logic assertions, the separating conjunction $P * Q$ asserts that the heap can be split into two disjoint parts, for which P and Q hold respectively, and its implication adjoint $P -\!\!* Q$ asserts that whenever the heap is extended with a disjoint heap for which P holds, Q holds in the result. The septraction $P -\!\!\circledast Q$ asserts that the heap is the difference between the bigger heap satisfying Q and the smaller one satisfying P. It is used mainly in RGSep logic reasoning. Separation logic has characterized several classes of assertions with special properties. In particular, an assertion P is said to be *precise* if given any heap, there is at most one subpart for which P holds.

The semantics of separation logic is defined over a pair of a store, that maps variables to values, and a heap, that maps addresses to values. We write $s, i \models_{\text{SL}} P$ to represent that the assertion P holds for the heap s and store i. We also write $s \uplus s'$ to represent that the heaps s and s' are domain disjoint.

RGSep logic The state space of a program consists of a dynamically evolving set of state components. A state component may represent a local variable of a thread or a shared variable allocated on the heap or in the store. Separating conjunction gives a natural way to separate one subset of state components from another on the heap.

In a concurrent system with ownership transfer, each state component at a time is *either* owned by a thread or mutual exclusion unit, *or* it is ownerless. In particular, the *ownerless* components can be assessed by all threads. Local variables are permanently owned by their threads; but ownership on shared variables can be dynamically transferred.

[1] Note that slot variables are implemented by safe registers. If a read operation accesses a safe register concurrently with a write operation, the value returned by the read operation will be a partially overwritten, non-coherent, value.

In concurrent separation logic [2, 12], which advocates interference-free programming style, no state component can be ownerless. Each shared variable at a time must either uniquely belong to a thread or is protected (i.e. owned) by a mutual exclusion unit. In principle a state component owned by one thread or mutual exclusion unit cannot be accessible to other threads. Threads, however, can dynamically trade ownership on state components with mutual exclusion units at synchronisation points. So over the timeline a state component can be passed around and shared by several threads.

In RGSep logic [18, 17] it further enhances the paradigm by allowing ownerless state components. Those components are accessible to all the threads, though the accesses must be made through atomic commands. Threads can trade state components with the ownerless part at the point when the atomic accesses are executed. Ownership transfer is essential to enabling non-atomic accesses to shared states in RGSep, e.g. data slots in the four-slot algorithm.

The syntax of RGSep assertions is defined as:

$$p, q \quad ::= \quad P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \forall x.p \mid \exists x.p$$

where p and q stand for RGSep assertions, and P for a separation logic assertion. A RGSep assertion in a thread is usually of form $P * \boxed{Q}$, where P specifies the local state of the thread, and the boxed Q specifies the ownerless state. Non-atomic commands in the thread can only access the P part and the accesses are interference-free. So the rules in RGSep logic for non-atomic commands are similar to those in sequential systems. Atomic commands in the thread can access $\boxed{Q}$ part interferingly. They require complicated rules from the rely-guarantee reasoning.

Stability. In the rely-guarantee methods a thread is specified in the context of a rely condition R and a guarantee condition G, both of which are sets of actions of the form $Q \rightsquigarrow Q'$, which means that the action changes the part of a heap satisfying Q into the one satisfying Q', while keeps the rest unchanged. The rely R specifies the maximal interference (i.e. changes to the ownerless state made by the environment) tolerable by the thread while the guarantee G specifies the maximal interference it can impose on the environment. Any command in the thread is required to respect G while any assertion annotating the thread is required to be stable under R. An assertion p is *stable* under an action $Q \rightsquigarrow Q'$ iff performing $Q \rightsquigarrow Q'$ on states satisfying p results in states satisfying p.

Specification and proof rules. The specification of a command c in a thread is a quadruple (p, R, G, q), where p, q are pre- and post-conditions. The judgement $\vdash\ c\ \mathbf{sat}(p, R, G, q)$ says that, from an initial state satisfying p, the execution of c under the environment R causes changes guaranteed by G, and the resulting states satisfy q. For non-atomic commands c that do not access the ownerless state, the proof rule from separation logic can be adopted directly:

$$\frac{\vdash_{SL} \{P\}\ c\ \{Q\}}{\vdash c\ \mathbf{sat}(P, R, G, Q)} \qquad \text{(PRIM)}$$

where $\vdash_{SL} \{P\}\ c\ \{Q\}$ is a specification in separation logic.

In this paper we adopt the early stability check convention [17]. The stability of an assertion is checked as soon as it is generated as a postcondition of some command, as shown in the rule E-ATOMR below. The proof rule for atomic command $\langle c \rangle$ consists of two sub-rules. In the first E-ATOMR, if the postcondition q is stable under the rely R, then the specification for $\langle c \rangle$ under R can be derived from the one under an interference-free environment. Otherwise q has to be weakened to pass the stability check.

$$\frac{\vdash \langle c \rangle\ \mathbf{sat}(p, \emptyset, G, q) \qquad q \text{ stable under } R}{\vdash \langle c \rangle\ \mathbf{sat}(p, R, G, q)} \qquad \text{(E-ATOMR)}$$

The second rule ATOM checks that the changes to the ownerless state made by the atomic command satisfy the guarantee G. The assertions for $\langle c \rangle$ are divided into two parts, one for the local state, and

another for the ownerless state. The boxed assertion P specifies the part of the original ownerless state that is changed by $\langle c\rangle$ into a resulting ownerless part specified by Q. Both P and Q are required to be precise for the rule to be sound. F specifies the unchanged part of the ownerless state. The unboxed assertions P', Q' are resp. the pre- and post-conditions for the local state.

$$\frac{P, Q \text{ precise} \qquad \vdash c\ \mathbf{sat}(P * P', \emptyset, \emptyset, Q * Q') \qquad (P \rightsquigarrow Q) \subseteq G}{\vdash \langle c\rangle\ \mathbf{sat}(\boxed{P * F} * P', \emptyset, G, \boxed{Q * F} * Q')} \quad (\textsc{Atom})$$

The other proof rules of RGSep can be found in [17]. They are not used in our proof.

Compositionality of proofs In the framework of combined logics like RGSep, compositionality can be interpreted differently following the different sub-logics in it.

Following the classical rely-guarantee method, one interpretation is the syntactical restriction on the use of auxiliary variables and local variables. More specifically, for the four-slot algorithm, it means that 1) the rely/guarantee conditions cannot use local variables wp, wi, rp and ri, and 2) the assertions in the reader thread cannot use wp and wi while the assertions in the writer cannot use rp and ri, and 3) no auxiliary variables can be introduced anywhere in the proof.

However, due to ownership transfer, the division of local state and shared state is not static. Slots can dynamically move between local state and shared state. It creates confusion for a syntactical interpretation of compositionality. For example, can the assertions in the reader refer to a slot that is currently owned by the writer? This is the reason why we need to distinguish local variables from locally owned shared variables.

Following separation logic, the other interpretation of compositionality is based on the use of the frame rule, which requires that only the set of variables directly relevant to a fragment of program, a.k.a. the footprint of the fragment, need to be used in formulating its assertions and rely-guarantee conditions [4]. In combined logics like RGSep, it implies that the use of frame rule should be restricted as little as possible.

4 Actions of Four-Slot Algorithm

Guarded actions In rely-guarantee reasoning, a rely (or guarantee) condition R (or G) consists of a set of actions to describe interference. Each action corresponds to an atomic command from a thread and specifies the 'parcel' of interference caused by the command to other threads, i.e. the visible effect of the command on ownerless states. In RGSep logic there are two ways to make visible changes to the ownerless state: modifying the values of ownerless state components or performing ownership transfer between the ownerless state and the local state.

In an ideal situation, an action should be of the form $P \rightsquigarrow Q$, where P and Q are precise separation logic assertions specifying ownerless states. The action semantics then is a state transition relation [18]:

$$[\![P \rightsquigarrow Q]\!] \quad \widehat{=} \quad \{(s \uplus s_0, s' \uplus s_0) \mid \exists i.\ s, i \models_{\text{SL}} P\ \wedge s', i \models_{\text{SL}} Q\}$$

where s_0 denotes an arbitrary state, and i represents the store for which P and Q hold.

However, in general actions are intrinsically linked with local states since the original command may refer to local variables or locally owned shared variables of the thread; especially, actions, via ownership transfer, can change the locally owned shared states. The above definition of actions is not complete since the changes to the locally owned shared states are not specified. The complete action semantics is:

$$\begin{aligned}[\![P \rightsquigarrow Q]\!] \quad \widehat{=} \quad & \{(s \uplus s_0, ls \uplus ls_0), (s' \uplus s_0, ls' \uplus ls_0) \mid \exists i.\ s, i \models_{\text{SL}} P\ \wedge s', i \models_{\text{SL}} Q\ \wedge \\ & \mathbf{dom}\ (ls \uplus s) = \mathbf{dom}\ (ls' \uplus s') \wedge ls' \subseteq (ls \uplus s) \wedge ls \subseteq (ls' \uplus s')\}\end{aligned}$$

where s_0 denotes an ownerless state, and ls, ls', ls_0 denote locally owned shared states. The action changes the ownerless state s to be s', and at the same time, the locally owned shared state ls to be ls', while keeps the domain of the union unchanged, i.e. $\mathbf{dom}\ (ls \uplus s) = \mathbf{dom}\ (ls' \uplus s')$, and the content of locally owned shared states unchanged. In another word, the only change to ls by the action is the transfer of components between it and the ownerless s.

In [17] an extended form of local guarded actions, $G|P \rightsquigarrow Q$, is introduced to improve the expressiveness of actions. G is the local guard specifying the local states necessary to enable the action. The semantics is not explicitly given in [17]. However, it seems to be a direct extension of the above semantics. One of the major limitation is that it cannot specify properties like *the reader currently owns no slot*. Here we introduce a different form of local guarded actions, $G|P \rightsquigarrow G'|Q$, defined as:

$$\begin{array}{rcl}[\![G|P \rightsquigarrow G'|Q]\!] & \widehat{=} & \{(s \uplus s_0, ls), (s' \uplus s_0, ls') \mid \exists i.\ s, i \models_{\text{SL}} P \ \wedge s', i \models_{\text{SL}} Q \\ & & \quad \wedge ls, i \models_{\text{SL}} G \wedge ls', i \models_{\text{SL}} G' \wedge \mathbf{dom}\ (ls \uplus s) = \mathbf{dom}\ (ls' \uplus s')\}\end{array}$$

Note that G and G' specify the whole locally owned shared states, not part of it as in the original local guarded actions of [17]. It improves the expressiveness of local guarded actions but restricts the use of frame rule over locally owned shared states[2].

Stability With the non-standard extension of guarded actions, the rule for stability needs to be updated as well. Furthermore we introduce a new form of stability that is *conditional* w.r.t. an invariant on the whole shared state. It enables us to exploit reasoning like *if neither the reader nor the ownerless part holds a slot, it must be held by the writer.*

Now given an action $\alpha = G|P \rightsquigarrow G'|Q$ from thread A, and an assertion $p = \boxed{L|S}$ from B, G and L specify the locally owned shared states of A and B respectively, and S specifies the ownerless state. $G * L * S$ then defines 'the region of concerns' for checking p's stability against α. Such a region is often associated with some invariants. Note that the local state of B other than the locally owned shared part is not changed by α, so is not included in p.

In order to check the stability of p against α w.r.t. I, we use the following definition: The assertion $\boxed{L|S}$ is *stable* under the action $G|P \rightsquigarrow G'|Q$ w.r.t. an invariant I iff $\neg(L * G * P)$ or $\neg(I \wedge (L * S * G))$ or $\neg(I \wedge (L * ((P \mathrel{-\!\!\circledast} S) * Q) * G'))$ or $(P \mathrel{-\!\!\circledast} S) * Q \Rightarrow S$. Actually, the first condition says that the locally owned shared state of B is not disjoint from the state of A; the second and third conditions say that the invariant I does not hold for the global states before and after the action is performed respectively. If one of the first three conditions holds, we consider the action is not enabled. Otherwise, after the action is performed, S still holds for the shared state, represented by the fourth condition.

The invariant for the four-slot algorithm is : $d \mapsto \_, \_, \_, \_ * \mathbf{true}$, denoted by *DInv*, meaning that at any time, the global state consists of four data slots. In the following we will construct the set of actions for atomic commands in $write(w)$ and $read()$. We will use uppercase letters for representing existentially quantified constants. Their scope spans the entire action. For simplicity we omit the quantifiers, the same convention as in the following proof outlines of the writer and reader.

Actions of $write(w)$ For $write(w)$, the atomic commands (1) and (2) do not modify the ownerless state while (4) and (5) modify the ownerless state components li and l. On the other hand, commands (2) and (4) transfer the ownership on slot $d[wp][wi]$ in order to make the non-atomic writing of $d[wp][wi]$ in command (3) possible.

[2]Note that in the four-slot algorithm the whole locally owned shared states are involved in the actions, so the restriction on the use of frame rule will not affect us.

Command (2) reads li, and it also acquires the ownership on slot $d[wp][wi]$ from the ownerless state. The ownership transfer is the visible effect and can be specified using three actions. Each action corresponds to a different case how local variables wp and wi can be represented by shared variables (i.e li, l and r). It is a price we have to pay in disallowing local variables for the sake of the compositionality.

Note that l and r form a handshake pair between the reader and writer[3]. If $\neg(l = r)$ holds at command (2) , it implies that, since the writer publishes its new location in l, the reader has not yet followed and moved to the new location (by changing r in command (7)). So the values held by r, when the writer is at command (1) or (2), should be the same, i.e. the negation of l; and wp should be equal to l at command (2). Notice that after command (2) wi becomes equal to the negation of $!li[wp]$. So slot $d[wp][wi]$ after command (2) is actually slot $d[!r][!li[!r]]$ at command (2). The ownership transfer in such a scenario is defined by the following action (10).

$$\neg(l = r) \wedge d[!r][!li[!r]] = A \;\rightsquigarrow\; \neg(l = r) \tag{10}$$

$$\mathbf{emp} \mid \begin{pmatrix} l = r * (d[r][!li[r]] = B \\ \vee d[r][li[r]] = C) \\ * d[!r][!li[!r]] = A \end{pmatrix} \;\rightsquigarrow\; d[!r][!li[!r]] = A \mid \begin{pmatrix} l = r \\ *(d[r][!li[r]] = B \\ \vee d[r][li[r]] = C) \end{pmatrix} \tag{11}$$

$$\begin{pmatrix} l = r * d[!r][!li[!r]] = A \\ * d[r][!li[r]] = B \end{pmatrix} \;\rightsquigarrow\; l = r * d[!r][!li[!r]] = A \tag{12}$$

On the other hand, if $l = r$ holds at command (2), there are two cases to consider. The first action (11) is when $wp = !r$ holds; it is the case that the reader has moved to the new location and the move (command (7)) happens before command (1) is executed. In such a scenario slot $d[wp][wi]$ after command (2) is slot $d[!r][!li[!r]]$, and in the pair of slots pointed to by r at least one of them is still ownerless since the writer is moving to the pair pointed to by $!r$ and the reader can own at most one slot at a time. The local guard of (11) says that before command (2) the writer owns no slot, but after owns $d[!r][!li[!r]]$. It is useful in proving the stability of assertion (31).

The other action (12) is when $wp = r$ holds; it is the scenario that the reader's move happens between command (1) and (2). Then slot $d[wp][wi]$ after command (2) is slot $d[r][!li[r]]$, and slot $d[!r][!li[!r]]$ and $d[!r][li[!r]]$ should be ownerless since there is no thread working on the pair pointed to by $!r$. In (12) only slot $d[!r][!li[!r]]$ is required to be ownerless. The weaker specification is strong enough for the proof in the sequel. The enabling conditions of actions (11) and (12) are not disjoint and they together with (10) form a non-deterministic action, which is an over-approximation of command (2).

After the update of slot $d[wp][wi]$ at command (3), command (4) swings the pointer $li[wp]$ to point to the newly written slot and simultaneously transfers its ownership back to the ownerless state. The specification of command (4) consists of two actions:

$$li[!r] = LI \;\rightsquigarrow\; li[!r] = !LI * d[!r][li[!r]] = W \tag{13}$$

$$l = r * li[r] = LI \;\rightsquigarrow\; l = r * li[r] = !LI * d[r][li[r]] = W \tag{14}$$

They correspond resp. to the two cases of $wp = r$ and $wp = !r$. For $wp = !r$ (action (13)), the writer is working on the pair pointed to by $!r$, so pointer $li[!r]$ is swung and the ownership of slot $d[!r][!li[!r]]$ is released. For $wp = r$ (action (14)), the reader and the writer are working on the same pair pointed to by r. So $l = r$ holds, $li[r]$ gets swung, and the ownership of $d[r][!li[r]]$ is released.

Command (5) publishes the location of the pair the writer has been working on in l. The action is visible to the reader only if the published location is different from the current location held in l. Such

[3] l is modified only by the writer and only when $l = r$ while r is modified only by the reader and only when $\neg(r = l)$.

a scenario, defined by (15), is possible only if the reader has moved to the same location as the writer's before the writer executes command (1). So $l = r = !wp$ holds, and at least one slot in the pair pointed to by r is ownerless since at command (5) the writer is not owning any slot, as specified by the local guard. The local guard is crucial for proving the stability of assertions in $read()$, e.g. (31).

$$\begin{array}{l}\mathbf{emp} \mid l = r * (d[r][!li[r]] = A \vee d[r][li[r]] = B) \rightsquigarrow \\ \mathbf{emp} \mid l = !r * (d[r][!li[r]] = A \vee d[r][li[r]] = B)\end{array} \qquad (15)$$

Actions of $read()$ For $read()$ the atomic commands (6) and (8) do not modify the ownerless state while (7) modifies the ownerless state component r. Commands (6) and (8) transfer the ownership on slot $d[rp][ri]$.

Command (7) publishes in r the location of the pair the reader is going to work on. It is visible only if the current location held in r is different from the location to be published. Since if $\neg(r = l)$ the writer will never move to a new location by modifying l and thus never own any slot pointed to by r, and plus the fact that at command (7) the reader does not own any slot too, the pair of slots pointed to by r is ownerless at command (7).The specification thus is:

$$\neg(r = l) * \left(\begin{array}{l} d[r][li[r]] = A \\ *d[r][!li[r]] = B \end{array}\right) \rightsquigarrow r = l * \left(\begin{array}{l} d[!r][li[!r]] = A \\ *d[!r][!li[!r]] = B \end{array}\right) \qquad (16)$$

The atomic command (8) acquires slot $d[rp][ri]$ from the ownerless state. Since slot $d[rp][ri]$ after command (8) is slot $d[r][li[r]]$ at command (8), the specification is:

$$\mathbf{emp} \mid d[r][li[r]] = A \rightsquigarrow d[r][li[r]] = A \mid \mathbf{emp} \qquad (17)$$

The local guards say that the reader acquires the slot $d[r][li[r]]$ from the ownerless state if the reader owns no slot before. They are needed in proving the stability of assertions in the $write(w)$, e.g. (20).

After command (9) finishes reading the slot, later commands should release the slot back to the ownerless state. However, there is no atomic command after command (9) in $read()$. The transfer has to be delayed till the next invocation of $read()$ executes its first atomic command, i.e. command (6). However, during the time interval, the writer might be able to sneak in and modify pointer $li[r]$. Thus slot $d[rp][ri]$ may become slot $d[r][!li[r]]$ and the specification of command (6) includes two possibilities:

$$\mathbf{emp} \rightsquigarrow d[r][!li[r]] = B \qquad \mathbf{emp} \rightsquigarrow d[r][li[r]] = A \qquad (18-19)$$

5 Main Proof with RGSep

The assertions for each thread specify both the visible and invisible effects of commands. In the four-slot algorithm, procedures $write(w)$ and $read()$ are invoked repeatedly by the writer and reader. The terminating state of one invocation is the starting state of the next invocation. Therefore, in the proof outline of $write(w)$ and $read()$, the entry precondition should be implied by the exit postcondition.

For an atomic command $\langle C \rangle$ under the rely and guarantee conditions R and G, the construction of its postcondition from a given precondition $P * \boxed{P'}$ takes three steps:

- Apply the inference rules of separation logic to derive postcondition $Q * Q'$, where Q and Q' specify resp. the local state and the ownerless state. The separation logic specification $\{P * P'\}C\{Q * Q'\}$ leads to C **sat**$(P * P', \emptyset, \emptyset, Q * Q')$ in RGSep.

- If the change to the ownerless state, i.e. $P' \rightsquigarrow Q'$, is contained within the guarantee G, apply rule (ATOM) to derive the specification of $\langle C\rangle$ under the empty rely and G: $\langle C\rangle$ **sat**$(\boxed{P'*F}*P, \emptyset, G, \boxed{Q'*F}*Q)$, where F complements P' and Q' and specifies the part of the ownerless state not accessed by $\langle C\rangle$.

- Apply rule (E-ATOMR) to stabilise the postcondition $Q'*F$ under the rely condition R and derive the final specification:
$$\langle C\rangle \textbf{ sat}(\boxed{P'*F}*P, R, G, \boxed{Q'*F}*Q)$$

Proof for $write(w)$ Figure 2 (on page 13) presents the proof outline of the writer. The precondition (20) for $write(w)$ asserts that, before the writer starts $write(w)$, at most one slot which is in the pair pointed to by r can be locally owned and the writer cannot be the owner. It is trivial to check that (20) is stable under actions (16), (18) and (19). The action (17) is enabled only when 1) the reader does not own any slot and 2) $d[r][li[r]]$ is ownerless. Thus all four slots should be ownerless when (17) is executed, and (20) still holds after the execution, so is stable.

Atomic command (1) sets wp to be the negation of r, giving rise to postcondition (21). However, (21) is not stable due to action (16), which modifies r and is enabled under some states covered by (21). Actually (16) will change the ownerless state from a state satisfying
$$r = !l * d[r][li[r]] = \_ * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * \mathbf{true}$$
to one satisfying
$$r = l * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * d[r][li[r]] = \_ * d[r][!li[r]] = \_ * \mathbf{true}$$
from which action (17) becomes enabled and can change it further to one satisfying
$$r = l * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * d[r][!li[r]] = \_ * \mathbf{true}$$
Then the new assertion is stable. Weakening (21) by adding the new assertion as disjunct gives rise to the stable assertion (22).

Atomic command (2) sets wi to be the negation of $li[wp]$ and acquires slot $d[wp][wi]$. Precondition (22) ensures the availability of slot $d[wp][wi]$ in the ownerless state. After command (2), we get the stable postcondition (23). Combining precondition (22) and postcondition (23), we can verify that the visible effect implied by $(22) \rightsquigarrow (23)$, i.e. acquiring slot $d[wp][wi]$, is guaranteed by actions (10) and (11) for the first disjunct and (12) for the second disjunct.

For non-atomic command (3), the stability of (24) follows from that of (23).

Atomic command (4) negates pointer $li[wp]$ and releases $d[wp][wi]$ back to the ownerless state, giving rise to the postcondition (25). Combining precondition (24) and postcondition (25), we can verify that the visible effect of $(24) \rightsquigarrow (25)$, i.e. negating $li[wp]$ and releasing $d[wp][wi]$, is guaranteed by action (13) for the first disjunct and (14) for the second disjunct.

However, (25) is not stable due to action (17), which is enabled under some states covered by the second disjunct in (25). By weakening $d[r][li[r]] = \_$ to $(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_)$ in the second disjunct, we arrive at the stable assertion (26).

Atomic command (5) copies the value of wp to l, giving rise to the stable postcondition (27). Combining precondition (26) and postcondition (27), we can verify that the visible effect of $(26) \rightsquigarrow (27)$ is guaranteed by action (15). Finally the exit postcondition (27) implies the entry precondition (20).

Proof for $read()$ Figure 3 (on page 14) presents the proof outline of the reader. The precondition (29) for $read()$ asserts that the reader locally owns a slot before $read()$ starts. The slot is the one that is left over from the last invocation of $read()$ and should be $d[r][li[r]]$ in most cases. However, if the reader and writer are working on the same pair, i.e. the one pointed to by r, $li[r]$ might get swung by

the writer and the leftover slot becomes $d[r][!li[r]]$. The second and third disjuncts in (29) are related to this scenario.

The second disjunct says that slot $d[r][!li[r]]$ is absent from *both* the ownerless part and the reader-owned part while the pair pointed to by $!r$ belong to the ownerless part. It corresponds to the case just before the swing: the reader and writer own $d[r][li[r]]$ and $d[r][!li[r]]$ resp. and $l = r$. The third disjunct says that slot $d[r][!li[r]]$ is owned by the reader while the writer might at most own $d[!r][!li[!r]]$. It corresponds to the case after the swing: the writer has released $d[r][li[r]]$ (i.e. the $d[r][!li[r]]$ slot before the swing) and might have acquired $d[!r][!li[!r]]$ in the next invocation of $write(w)$. The first disjunct in (29) says that slot $d[r][!li[r]]$, as well as slot $d[!r][li[!r]]$, are in the ownerless part. It corresponds to the case that the reader and writer are working on different pairs and own $d[r][li[r]]$ and (probably) $d[!r][!li[!r]]$ resp.

It can be verified that (29) is stable. Especially it should be noted that 1) (11) is not enabled under the second disjunct since the global invariant ***DInv*** is violated, and 2) performing action (12) under states covered by the first disjunct will result in states that are covered by the second disjunct, and 3) performing action (14) under states covered by the second disjunct gives rise to states covered by the third disjunct.

Atomic command (6) copies the value of l to rp and releases the leftover slot from the last invocation, giving rise to the postcondition (30). It is trivial to verify that the visible effect of $(29) \rightsquigarrow (30)$ is guaranteed by actions (18) and (19).

However, (30) is not stable due to action (15), which is enabled under some states covered by the first disjunct of (30). Action (15) is not enabled under the second disjunct of (30) since the writer owns $d[r][!li[r]]$. Performing action (15), which negates the value of l, on states covered by the first disjunct gives rise to states covered by:

$$l = !r = !L * d[r][li[r]] = \_ * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}$$

Weakening (30) by adding it as disjunct gives rise to the stable assertion (31).

Atomic command (7) copies the value of rp to r, giving rise to postcondition (32). The first disjunct of (31) are split into the first two disjuncts of (32). It is easy to verify that the visible effect of $(31) \rightsquigarrow (32)$, i.e. negating r, is guaranteed by action (16).

We can check that (32) is stable. Actions (11) and (15) of the writer are both enabled under states covered by the second disjunct of (32). Performing action (15) on states covered by the second disjunct gives rise to states:

$$d[!r][li[!r]] = \_ * r = L * d[!r][!li[!r]] = \_ * d[r][li[r]] = \_ * d[r][!li[r]] = \_ * \mathbf{true}$$

since action (15) is enabled only when slot $d[r][!li[r]]$ is in the ownerless part. Then performing action (10) or (11) gives rise to states covered by:

$$d[!r][li[!r]] = \_ * r = L * d[r][li[r]] = \_ * d[r][!li[r]] = \_ * \mathbf{true}$$

which is exactly the first disjunct of (32).

Atomic command (8) copies the value of pointer $li[rp]$ to ri and acquires slot $d[r][li[r]]$, giving rise to postcondition (33). It is trivial to verify that the visible effect of $(32) \rightsquigarrow (33)$, i.e. acquiring slot $d[r][li[r]]$, is guaranteed by action (17). However, (33) is not stable due to action (14), which is enabled under the second and third disjuncts of (33). Performing action (14) which swings $li[r]$ on states covered by the second and the third disjuncts resp. gives rise to states covered by:

$$d[r][!li[r]] = \_ * \boxed{r = L * li[L] = !LI * d[r][li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}}$$

Weakening (33) by adding it as disjunct gives rise to the stable assertion (34).

Non-atomic command (3) is a local operation. The stability of its postcondition (35) follows from that of its precondition (34). Finally, further weakening (35) gives rise to the exit postcondition (36) of $read()$, which is equivalent to the entry precondition (29).

Data coherence of four-slot algorithm In our proof the assertions are both loop invariants for each local thread and stable under interference from the other thread. As an example to demonstrate the power of the assertions it is interesting to note assertions (23) and (34) suffice to prove the data coherence of four-slot algorithm. Basically assertion (23) says every time the writer is going to write a slot it must have owned it, while assertion (34) says the same for the reader. Since at most one thread can own a slot at a time, it implies at any time at most one thread can access a slot.

Discussion We can completely get rid of the local guards in the actions by introducing auxiliary variables. One solution is to introduce two auxiliary variables wa and ra respectively to represent that whether the writer and the reader hold a slot (e.g. $wa, ra = 1$) or not ($wa, ra = 0$). The local guards of actions then can be represented by using wa or ra instead. For instance, the action (11) of the writer can be changed into the following equivalent form:

$$wa = 0 * P \rightsquigarrow wa = 1 * Q$$

where P, Q denote the corresponding formulas in (11) for simplicity. On the reader side, we add an invariant as conjunction to the ownerless part of each assertion,

$$\begin{array}{l}(ra = 0 \Rightarrow ((\exists i, j.\neg d[i][j] = \_) \Rightarrow wa = 1)) \\ \vee (ra = 1 \Rightarrow ((\exists i, j, i', j'.\neg(d[i][j] = \_ * d[i'][j'] = \_)) \Rightarrow wa = 1))\end{array}$$

which means that, when the reader holds no slot, if there is one slot not in ownerless state, it must be owned by the writer, i.e. $wa = 1$; or when the reader holds a slot, if there are two slots not in ownerless state, then $wa = 1$. From this invariant, we can deduce from the second disjunct of assertion (29) that $wa = 1$ and as a result the above action will not be enabled. We can easily verify that (29) is stable under the action. Rewriting all the local guarded actions and adding an invariant for each thread in this way, we can give another proof of the algorithm in RGSep without using local guards.

6 Related Work and Conclusion

Previously, two groups of people have worked on the assertional proof of Simpson's four-slot algorithm. Jones is a proponent of compositionality in assertional proofs [8]. He and his students have used the combined methods of rely-guarantee and data refinement [6, 5, 9, 10] to prove atomicity of four-slot algorithm and to reduce (or even eliminate) the use of auxiliary variables in the proofs. They define an abstract specification of atomic registers and a model of the four-slot algorithm, and then prove the refinement between the two by using Nipkow's retrieve rules. Earlier proofs are quite complicated and need auxiliary variables. Their most recent work is much simpler and removes auxiliary variables [10]. However, a difference from our work is that in [10] commands (7), (8) and (9) (as well as commands (4) and (5)) are assumed to be executed in one atomic block and the protocol of ownership transfer is not illustrated.

The other group of people use RGSep logic. Bornat and Amjad [1] construct an assertional proof for Simpson's algorithm using RGSep. Due to the use of separation logic, the proof is more modular and simpler than by using rely-guarantee methods alone since predicates need only to specify the *footprints* of commands. Still, as conceded by the authors, the proof is complicated and ownership transfer on slots is not really exploited. The proof has to use some special rules to work around the problem of non-atomic accesses to slots, rather than to directly use the rule designed for local states. So it misses the key advantage offered by RGSep: the separation of local state from ownerless state.

Calcagno *et al.* [3] provide a safety checker, called SmallfootRG, for reasoning about a class of fine-grained heap manipulating concurrent algorithms based on RGSep logic. They claim a proof of the four-slot algorithm by the SmallfootRG tool, but auxiliary variables are needed in it as well.

In this paper, we have done an extreme exercise to understand the potential and limit of ownership transfer in facilitating simpler and more intuitive proofs. The exercise finishes with a syntactically compositional proof outline of Simpson's four-slot algorithm. It is arguably simpler than previous works and captures, as well as justifies, our intuition about ownership transfer in four-slot algorithm. The successful elimination of auxiliary variables demonstrates the power of ownership transfer despite that we have to make non-standard extensions to RGSep logic. The places of extensions confirm the importance of local guards in RGSep logic and also point to alternative semantics for local guards and actions.

Finally, we concede that one important limitation of our work is that data freshness of the algorithm is not investigated. It will be our next step work, for which some major challenges are expected in finding a compositional proof.

Acknowledgment

The authors acknowledge support from the Macao Science and Technology Development Fund under the PEARL project (grant number 041/2007/A3) and the HTTS project. The authors would like to thank all the anonymous reviewers for their useful comments.

References

[1] R. Bornat and H. Amjad. Inter-process buffers in separation logic with rely-guarantee. to appear in FACJ, 2009.

[2] S. Brookes. A semantics for concurrent separation logic. In *Proceedings of 15th CONCUR, LNCS*, volume 3170, pages 16–34. Springer, 2004.

[3] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *Proceedings of SAS'07, LNCS 4634*, 2007.

[4] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of POPL '09*, pages 315–327, New York, NY, USA, 2009. ACM.

[5] N. Henderson. Proving the correctness of Simpson's 4-slot ACM using an assertional rely-guarantee proof method. In *Proceedings of FME 2003: Formal Methods, LNCS 2805*, pages 244–263. SpringerLink, 2003.

[6] N. Henderson and S. Paynter. The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In *FME '02: Formal Methods - Getting IT Right*. Springer-Verlag, 2002.

[7] C. B. Jones. Specification and design of (parallel) programs. *IFIP Congress*, pages 321–332, 1983.

[8] C. B. Jones. The role of auxiliary variables in the formal development of concurrent programs. Technical Report CS-TR-1179, School of Comp. Sci., Newcastle Univ., 2009.

[9] C. B. Jones and K. G. Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In *Proceedings of ABZ '08, LNCS 5238*, pages 360–377, 2008.

[10] C. B. Jones and K. G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. Technical Report CS-TR-1166, School of Comp. Sci., Newcastle Univ., 2009.

[11] L. Lamport. The mutual exclusion problem: part I - a theory of interprocess communication. *J. ACM*, 33(2):313–326, 1986.

[12] P. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comp. Sci.*, 375(1-3):271–307, 2007.

[13] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Comm. ACM*, 19(5):279–285, 1976.

[14] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proc. of POPL'05*. ACM Press, 2005.

[15] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*. IEEE CS, 2002.

[16] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proc.*, 137(1):17 – 30, 1990.

[17] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge Computer Laboratory, 2008.

[18] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proceedings of 18th CONCUR, LNCS*, volume 4703, pages 256–271. Springer, 2007.

[19] Q. Xu, W. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

$$\boxed{d[!r][!li[!r]] = \_ * d[!r][li[!r]] = \_ * (d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true}} \quad (20)$$

$\langle wp := !r \rangle$

$$wp = !R * \boxed{\begin{array}{l} r = R * d[!r][!li[!r]] = \_ * d[!r][li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \quad (21)$$

$\Longrightarrow$ (stability)

$$\left(\begin{array}{l} \boxed{\begin{array}{l} r = R * d[!r][!li[!r]] = \_ * d[!r][li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \\ \vee \boxed{\begin{array}{l} l = r = !R * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *d[r][!li[r]] = \_ * \mathbf{true} \end{array}} \end{array} \right) \quad (22)$$

$\langle wi := !li[wp] \rangle$

$$\left(\begin{array}{l} \boxed{\begin{array}{l} r = R * li[!R] = LI * d[!r][li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \\ \vee \boxed{\begin{array}{l} l = r = !R * li[!R] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *\mathbf{true} \end{array}} \end{array} \right) \quad (23)$$

$d[wp][wi] := w$

$$\left(\begin{array}{l} \boxed{\begin{array}{l} r = R * li[!R] = LI * d[!r][li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \\ \vee \boxed{\begin{array}{l} l = r = !R * li[!R] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *\mathbf{true} \end{array}} \end{array} \right) \quad (24)$$

$\langle li[wp] := wi \rangle$

$$\left(\begin{array}{l} \boxed{\begin{array}{l} r = R * li[!R] = !LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \\ \vee \boxed{\begin{array}{l} l = r = !R * li[!R] = !LI * d[r][li[r]] = \_ * d[!r][li[!r]] = \_ \\ *d[!r][!li[!r]] = \_ * \mathbf{true} \end{array}} \end{array} \right) \quad (25)$$

$\Longrightarrow$ (stability)

$$\left(\begin{array}{l} \boxed{\begin{array}{l} r = R * li[!R] = !LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \\ \vee \boxed{\begin{array}{l} l = r = !R * li[!R] = !LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \end{array} \right) \quad (26)$$

$\langle l := wp \rangle$

$$\left(\begin{array}{l} \boxed{\begin{array}{l} r = R * l = !R * li[!R] = !LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \\ \vee \boxed{\begin{array}{l} l = r = !R * li[!R] = !LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ \\ *(d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true} \end{array}} \end{array} \right) \quad (27)$$

$\Longrightarrow$

$$\boxed{d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * (d[r][li[r]] = \_ \vee d[r][!li[r]] = \_) * \mathbf{true}} \quad (28)$$

Figure 2: Proof for $write(w)$

$$\left(\begin{array}{l}(d[r][li[r]] = \_ * \boxed{d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li \mapsto \_, \_})\\ \vee(d[r][!li[r]] = \_ * \boxed{d[r][li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\end{array}\right) \tag{29}$$

$\langle rp := l \rangle$

$$\left(\begin{array}{l}\boxed{l = L * d[r][li[r]] = \_ * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}}\\ \vee\boxed{d[r][li[r]] = \_ * l = r = L * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li \mapsto \_, \_}\end{array}\right) \tag{30}$$

$\Longrightarrow$ (stability)

$$\left(\begin{array}{l}\boxed{\begin{array}{l}(l = r = L \vee l = !r = L \vee l = !r = !L)\\ * d[r][li[r]] = \_ * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}\end{array}}\\ \vee\boxed{d[r][li[r]] = \_ * l = r = L * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li \mapsto \_, \_}\end{array}\right) \tag{31}$$

$\langle r := rp \rangle$

$$\left(\begin{array}{l}\boxed{d[r][li[r]] = \_ * r = L * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}}\\ \vee\boxed{d[!r][li[!r]] = \_ * l = r = L * d[!r][!li[!r]] = \_ * d[r][li[r]] = \_ * \mathbf{true}}\\ \vee\boxed{d[r][li[r]] = \_ * l = r = L * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li \mapsto \_, \_}\end{array}\right) \tag{32}$$

$\langle ri := li[rp] \rangle$

$$\left(\begin{array}{l}(d[r][li[r]] = \_ * \boxed{r = L * li[L] = LI * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r = L * li[L] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r = L * li[L] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li[!L] = \_})\end{array}\right) \tag{33}$$

$\Longrightarrow$ (stability)

$$\left(\begin{array}{l}\vee(d[r][li[r]] = \_ * \boxed{r = L * li[L] = LI * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r = L * li[L] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r = L * li[L] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li[!L] = \_})\\ \vee(d[r][!li[r]] = \_ * \boxed{r = L * li[L] = !LI * d[r][li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\end{array}\right) \tag{34}$$

$y := d[rp][ri]$

$$\left(\begin{array}{l}\vee(d[r][li[r]] = \_ * \boxed{r = L * li[L] = LI * d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r = L * li[L] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r = L * li[L] = LI * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li[!L] = \_})\\ \vee(d[r][!li[r]] = \_ * \boxed{r = L * li[L] = !LI * d[r][li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\end{array}\right) \tag{35}$$

$\Longrightarrow$

$$\left(\begin{array}{l}(d[r][li[r]] = \_ * \boxed{d[r][!li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\\ \vee(d[r][li[r]] = \_ * \boxed{l = r * d[!r][li[!r]] = \_ * d[!r][!li[!r]] = \_ * li \mapsto \_, \_})\\ \vee(d[r][!li[r]] = \_ * \boxed{d[r][li[r]] = \_ * d[!r][li[!r]] = \_ * \mathbf{true}})\end{array}\right) \tag{36}$$

Figure 3: Proof for $read()$