Verifying Concurrent First Order Imperative Programs with Separation Logic and Graphs

Carsten Varming

April 12, 2012

Contents

1 Motivation

Reasoning about concurrent first-order programs can be difficult and time consuming. Often the challenging part is to capture the programmer's intuition about all the possible executions of a set of concurrent processes. More specifically, it is often difficult to limit the assumptions made about the interaction between a process and the environment it executes in, i.e., the other processes concurrently executing.

Existing logics such as Concurrent Separation Logic and Rely-Guarantee both have a rigid notion of the interaction between a process and its environment, and a programmer's intuition can be very different from this notion. For instance, in concurrent separation logic a valid specification $r(X) : I \vdash \{P\}c\{Q\}$ assumes that the environment may acquire and subsequently release the region r as many times as it pleases as long as the invariant I is preserved separately. Similarly, the rely-relation used in Rely-Guarantee logics is required to be reflexive and transitively closed, and any pre- or post-condition describing shared state must be stable, i.e., invariant under the rely-relation. This condition implies that if a process in the environment updates the shared state following some relation, then any pre- and post-condition for the current process must remain valid no matter how many times the environment updates the shared state. If a programmer wants to use a different protocol for the interaction between his process and its environment, then he must encode his protocol into the protocol specified by the logic he is using. CSL and Rely-Guarantee logics enable such reasoning via auxiliary variables, but the encodings can be difficult and time consuming to find. And the resulting specification of each process often ends up encoding all the effects of the environment. Hence if you change the environment, then you may need a new specification for each process.

We can illustrate these problems by trying to reason about a program first introduced by Owicki [Owi75] to illustrate the need for auxiliary variables:

$$\mathsf{resource\ r\ in\ (with\ r\ do}\ x := x + 1 \,\|\, \mathsf{with\ r\ do}\ x := x + 1)$$

If we start with a state where x is 0 and if we deem that r protects x, then you expect to be able to deduce that x is 2 when the program terminates.

In CSL you must find an assertion I that is strong enough to imply $x = 2$ after both processes have terminated, and is invariant under the action of each process. But if I implies that $x = v$ for any given v and I is invariant under $x := x + 1$, then I must imply $x > v$ and thus I must imply false. The problem is that when we try to give a specification for the left process, CSL assumes that the right process may increment x an arbitrary number of times. It is well-known that the solution to this problem is to add an auxiliary variable that counts the number of times the right process increments x. This variable will then be free in the invariant I for r. Symmetrically, we need to add an auxiliary variable to keep track of the left process. As the invariant I is shared between both processes the auxiliary variable needed to keep track of one process slips into the specification of the other process.

In Rely-Guarantee logics the pre-condition must be stable under the rely-relation, i.e., the effect of the other process, whence $x = 0$ is not a valid pre-condition. Again auxiliary variables can be used to keep track of the processes in the environment, but then again the auxiliary variables needed for one process slip into the specification of the other processes.

The problem is that the encoding of the *protocol* for updating x is part of the specification of each process.

In the example above, it may appear easy to use auxiliary variables to to limit the assumptions about the interaction between the environment and a process, but with multiple resources the complexity quickly increases. The following program illustrates the problem. It uses two different resources r_0 and r_1, and is safe when executed in isolation from a state where $a_0 = a_1 = 0$:

$$\big(\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ a_0 := 1;\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ a_1 := 1\big) \parallel \\ \big(\mathsf{with}\ \mathsf{r}_1\ \mathsf{when}\ a_1 \neq 0\ \mathsf{do}\ \mathsf{skip};\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ \mathsf{if}\ a_0 = 0\ \mathsf{then}\ \mathsf{crash}\ \mathsf{else}\ \mathsf{skip}\big). \quad (1.1)$$

Here crash is a command that aborts when executed from any state.[1] To show that the program is safe, the following protocol must be encoded: ownership of a_1 is split between the left process and r_1, ownership of a_0 is split between the left process and r_0, and a_1 is set to 1 after a_0 has been set to 1. If we can encode the protocol, then we should be able to deduce that $a_0 \neq 0$ after the right process has acquired r_1, and we should be able to pass this information on to the following if-statement, and hence to deduce that the program is safe.

In concurrent separation logic we have the following problem: Let (1.2) be a template for a CSL specification for (1.1) annotated with fresh auxiliary variables and commands c_0 and c_1 assigning to these auxiliary variables:

$$\mathsf{r}_0 : I_0, \mathsf{r}_1 : I_1 \vdash \{-\}\ \big(\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (a_0 := 1;\ c_0);\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (a_1 := 1;\ c_1)\big) \parallel \\ \big(\mathsf{with}\ \mathsf{r}_1\ \mathsf{when}\ a_1 \neq 0\ \mathsf{do}\ c_2;\ \ \{P\}\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ \mathsf{if}\ a_0 = 0\ \mathsf{then}\ \mathsf{crash}\ \mathsf{else}\ \mathsf{skip}\big)\ \{-\}, \quad (1.2)$$

Let P be an assertion that must hold of the state local to the right process in between its last release of r_1 and its first acquire of r_0. As the specification above is supposed to be derived in CSL, any variable free in P cannot be assigned in the left process (a side-condition present in [O'H07], missing in [Bro04, Bro07], and enforced by the proof system in [Bro11]). Likewise, any variable free in I_0 cannot be assigned in c_2. Hence any variable that both P and I_0 depend on cannot be assigned in the program, and thus reasoning about the value of the auxiliary variables in $I_0 * P$ cannot directly be used to show $a_0 \neq 0$.

At first glance it might seem impossible to show $I_0 * P \Rightarrow a_0 \neq 0$, but there are solutions. One solution is to pass around a token represented as a precise formula T such that $T * T \Rightarrow \mathsf{false}$. For instance, if T is $x \mapsto -$, I_0 is $(a_0 = 0 \wedge T) \vee (a_0 = 1 \wedge \mathsf{emp})$, and I_1 is

[1] crash could be $\mathsf{dispose}\ x;\ [x]{:=}0$

$(a_1 = 0 \land b = 0 \land \mathsf{emp}) \lor (a_1 = 1 \land b = 0 \land T) \lor (a_1 = 1 \land b = 1)$, then we can prove:

$$
\begin{array}{l}
\mathsf{r}_0 : I_0, \mathsf{r}_1 : I_1 \vdash \{a_0 = a_1 = b = 0 \land \mathsf{emp}\} \\
\qquad\qquad\qquad\qquad \big(\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ a_0 := 1;\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ a_1 := 1\big) \parallel \\
\quad \big(\mathsf{with}\ \mathsf{r}_1\ \mathsf{when}\ a_1 \neq 0\ \mathsf{do}\ b := 1;\ \ \{T\}\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ \mathsf{if}\ a_0 = 0\ \mathsf{then\ crash\ else\ skip}\big) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{a_0 = a_1 = b = 1 \land \mathsf{emp}\},
\end{array}
$$

but this introduces a heap cell whose only purpose is to facilitate the encoding of the protocol for updating a_0 and a_1. Reasoning with such encodings can quickly get complicated.

Finally we note that the program, being independent of the heap, fits in the programming language from Owicki's thesis [Owi75] and thus by her completeness theorem we can find a proof that doesn't use the heap. Unfortunately this also results in very complicated proofs.[2]

In Rely-Guarantee logics the problem with multiple regions owning separate heaps is often ignored. For example, in [DYDG$^+$10, DFPV09, Fen09, FFS07, VP07] the shared state is read and updated by atomic blocks of commands, i.e., commands executed in isolation. This enables the programmer to give *global* specifications of the shared state, but it also prevents him from reasoning about non-atomic changes to the shared state.

2 Proposed Thesis

I propose to develop a modular proof methodology for verification of first-order concurrent programs operating on mutable state, based on a formalization of *protocol safety* and *heap safety* as distinct properties. A key new idea is that it is possible to reason separately about these two properties, and that this leads to proofs that can easily be reused with many different environments.

I will introduce a new logic with two levels, a level to reason about *heap safety* and a level to reason about *protocol safety*. I will use a trace semantics for the programming language familiar from Brookes' works on concurrent separation logic [Bro07], and I will introduce a semantics for each new component in the specifications used on each level. Then I will define validity of the judgments of the logic and show soundness of the rules by showing that they preserve validity. Finally, I will show that valid specifications can be used to reason about the execution of programs using the semantics familiar from [Bro07].

I will show that encoding protocols for interaction between processes as graph structures enables the programmer to directly represent the different phases that the shared state passes through during execution and when processes must synchronize. Ideally, the ability of the programmer to design concurrent programs by first specifying the protocol, i.e., the

[2] The proof constructed in [Owi75] uses non-trivial encodings of the commands in the environment.

intended interaction between processes, and then to find commands suitable for these processes, will make it easier to write correct concurrent programs. And given a concurrent program, the ability to express the interaction between processes directly as graph-like structures will lead to more intuitive specifications and proofs. I intend to demonstrate this method by reconstructing well-known concurrent programs such as Peterson's mutual exclusion algorithm, from specifications of their protocols. I will then compare the proof and specifications to proofs and specifications for the same programs in Concurrent Separation Logic and Rely-Guarantee logics.

3 Separation Logic

To show my thesis I will primarily use separation logic. First, I will give a program logic for a concurrent first-order programming language with identifiers and a heap. The program logic will relate an interference command L, expressing assumptions about changes to the *shared* heap, to a Hoare triple. I will then give a logic to show that L satisfies a protocol specified by the programmer using graph-like structures to represent a protocol abstractly. This results in a two-step verification method for programs. The first step is to give a proof of a Hoare triple annotated with an interference command L. Such a proof shows *heap safety* under the assumption of *protocol safety* of L. The second step is then to show *protocol safety* of L. The result of the second step can then be applied to the annotated Hoare triple to get a simple Hoare triple that describes the behavior of the program when executed in isolation.

The programming language that I will use is a concurrent first-order imperative language with identifiers and a heap. Identifiers are statically scoped, and each free identifier stands for a value of a given type. We use contexts, i.e., finite functions from identifiers to types, to keep track of the type of each free identifier. We use Γ to denote contexts. The use of identifiers instead of assignable variables simplifies reasoning by eliminating most side-conditions seen in CSL, and we do not lose any generality as we will extend the programming language with the ability to read heap locations when evaluating boolean expressions used in conditional critical regions and while loops. We do so by defining a new class of extended boolean expressions. These extended expression will then replace the boolean expressions in while-loops and conditional critical regions. Another effect of using identifiers instead of variables is that communication between processes is conducted exclusively via the heap.

The assertion logic is higher-order separation logic [BBTS07] extended with permissions as in [BCOP05]. This is a sub-structural logic with spatial connectives that is suitable for specifying and reasoning about shared mutable data structures [Rey02]. As in [BBTS07] assertions well-typed in a context Γ will, given a store from the meaning of Γ, represent sets of heaps. Notice that I use the word "store" instead of the more common "environment", as I will use the later word as a reference to processes executing concurrently with a given process. The logic has spatial connectives: points-to $e \overset{p}{\mapsto} e'$ and spatial conjunction $P * Q$.

The points-to assertion $e \overset{p}{\mapsto} e'$ is satisfied by the heap defined on the single location $\{e\}$ which is mapped to the value e' with permission p. The assertion $P * Q$ is satisfied by any heap that can be split into two separate heaps such that the first heap satisfies P and the second heap satisfies Q. The separation between the two heaps is what enables the compact treatment of aliasing found in separation logic [Rey02].

Unlike the language used in CSL, I introduce extended expressions that may lookup values in the heap. This implies that an extended expression may abort. I introduce a simple Separation Logic with specifications: $\vdash \{P\}\ e\ \{\lambda x : \tau .\ Q\}$ to reason about safety and the value of an extended expression. Here, P is a pre-condition, e is an extended expression that may lookup values in the heap, and Q is a post-condition. The binder $\lambda x : \tau$ names the value of e in Q (see [KAB$^+$09]).

The following is an example of a valid specification for an extended boolean expression that looks up the value store at x and compares it to 2:

$$\vdash \{x \hookrightarrow y\}\ \mathsf{let}\ z{=}[x]\ \mathsf{in}\ z = 2\ \{\lambda r : \mathsf{bool}.\ x \hookrightarrow y \wedge r = (y = 2)\}$$

The pre-condition $x \hookrightarrow y$ is short for $\exists p.\ x \overset{p}{\mapsto} y * \mathsf{true}$. Validity of the Hoare triple shows that the expression doesn't abort when evaluated from a heap where location x has value y with some permission p. The post-condition asserts that the expression evaluates to $y = 2$ and the heap still has value y at location x.

3.1 Heap Safety

Next I introduce a specification $L \vdash \{P\}\ c\ \{Q\}$ to show *heap safety* of the command c. Here, L is an *interference command* that specifies the assumptions of the shared heap made by the programmer and the updates to the shared heap performed by c, P is the pre-condition, and Q is the post-condition. The specification is well-typed in a context Γ if and only if each component is well-typed in Γ. The specification $L \vdash \{P\}\ c\ \{Q\}$ deems the heap to be a separate union of a *local* heap and a *shared* heap. The Hoare-triple $\{P\}\ c\ \{Q\}$ from the specification is used to reason about the *local* heap and the interference command L is used to reason about the *shared* heap.

The interference command L summarizes the effect on the *shared* heap of executing a command. The simplest interference command is nil. It is used to indicate that a command is independent of any environment and that the command doesn't cause any interference. The interference command $\mathsf{with}\ r\ \mathsf{do}\ (\Gamma', P', Q')$ is a triple (Γ', P', Q') annotated with a resource identifier r. I will use r and r_i to denote resource names and let r and r' be identifiers ranging over resource names. The assertion Q' is precise, i.e., for each heap h and each store defining the values of the free variables in Q' at most one subheap of h can be satisfied by Q'. The context Γ' binds in P' and Q' every identifier in $\mathrm{dom}(\Gamma')$, and the assertion $\exists\Gamma'.\ P'$, i.e., the assertion P' with every identifier in Γ' existential quantified, is precise. The triple is well-typed in the context Γ if and only if P' and Q' are well-typed in Γ, Γ', i.e., the context Γ extended with every mapping $x \mapsto \tau$ from Γ'. The triple represents a partial function from heaps to sets of heaps. If such a function f is undefined on a given heap h, then we say f aborts when executed from h, otherwise we say that f is *safe* when executed from h. The partial function f represented by (Γ', P', Q') is defined as the local action (see [COY07])

$$f(h) = \bigsqcap \{Q'(\sigma') * \{h''\} \mid \exists h'.\ h = h' \uplus h'' \wedge h' \text{ satisfies } P'(\sigma') \wedge \sigma' \text{ has type } \Gamma'\},$$

whence f is defined on a heap h if and only if h can be split into separate heaps h' and h'' such that h' satisfies P' for some store σ' defining the identifiers bound in Γ'. The result $f(h)$ is then a set of heaps where each member h_r can be split into the heap h'' and a heap satisfying Q' with the identifiers bound in Γ' defined by the store σ'. The store σ' may not be unique, in which case $f(h)$ is a subset of every set $Q'(\sigma') * \{h''\}$. By precision of $\exists\Gamma'.\ P'$, h' and h'' are unique. Notice that the result of an execution of a triple from a shared heap is either abort or a set of heaps. The resource annotation r on the triple shows that the triple is guarded by acquiring the resource r before the beginning of its execution, and releasing r after the execution terminates. We say that the execution of the triple has a *duration* starting when r is acquired and ending when r is released. To capture that r guards a triple represented by f and the duration of the execution of f, we will use a simple trace semantics with actions for acquiring and releasing resources. This semantics is similar to the trace semantics found in [Bro04]. The acquire action will be annotated with r and f and the release action will be annotated with r. The notion of execution is then lifted to sets of traces.

Validity of the specification:

$$\mathsf{with}\ r\ \mathsf{do}\ (\Gamma', P', Q') \vdash \{P\}\ \mathsf{with}\ r\ \mathsf{do}\ c\ \{Q\}$$

implies that if the *local* heap satisfies P and if after the process has acquired r the triple (Γ', P', Q') doesn't abort when executed from the *shared* heap, and if c terminates, then before r is released the *shared* heap has a part satisfying Q' and the *local* heap satisfies Q. The assumption that "(Γ', P', Q') doesn't abort when executed from the shared heap" captures the idea that when the program acquires r it assumes ownership of a part of the shared heap that satisfies $[S/\Gamma']P'$ for some substitution S defined on Γ', by moving this part from the shared heap to its local heap. The body c is then executed and if it terminates, then the resulting local heap can be split into two parts, one part satisfying Q and one part satisfying $[S/\Gamma']Q'$. The second part is then moved back to the shared heap before r is released. If the result of executing the triple on the shared heap is non-deterministic, i.e., $[S/\Gamma']Q'$ satisfies more than one heap, then c can non-deterministically choose which of the heaps it will transfer back to the shared heap. The transfer of ownership, i.e., the moving of heaps from the shared heap to the local heap and vice versa only takes place in the logic (the specification $L \vdash \{P\}\ c\ \{Q\}$ deems the global heap to be a union of a local and a shared heap) as in [O'H07]. Precision of $(\exists\Gamma'.\ P')$ and Q' ensures that the ownership transfers are well-defined. Here $\exists\Gamma'$ is the iterated existential quantification over each mapping $x \mapsto \tau$ from Γ'. Notice that our notion of execution of triples, and in general interference commands, is different from our notion of execution of commands. The execution of a triple from a heap can either return a set of heaps or abort, but the execution of a command c can either not terminate, return a heap, or abort.

The following is an example of a valid specification for a program that acquires the resource r_0, adds one to the shared value at location x, and then releases r_0:

$$\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \vdash \{\mathsf{emp}\}\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (\mathsf{let}\ y{=}[x]\ \mathsf{in}\ [x]{:=}y+1)\ \{\mathsf{emp}\}$$

The interference command $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1)$ shows that the programmer assumes that after the program has acquired r_0, it can take ownership of a part of the shared heap satisfying $x \overset{\top}{\mapsto} j$ for some j, i.e, a heap $[x \mapsto (j, \top)]$ with a singleton domain $\{x\}$ whose element is mapped to j with full permission, and before the program releases r_0, ownership of a heap satisfying $x \overset{\top}{\mapsto} j+1$ is transferred back to the shared heap. Validity of the specification implies that if the programmer's assumption is valid for the shared heap, then the Hoare triple is valid for the local heap. The specification above is an instance of the rule for conditional critical regions:

$$\frac{\vdash \{P * P'\}\ e\ \{\lambda b : \mathsf{bool}.\ P''\} \qquad \mathsf{nil} \vdash \{[\mathsf{true}/b]P''\}\ c\ \{Q * Q'\}}{\mathsf{with}\ r\ \mathsf{do}\ (\Gamma', P', Q') \vdash \{P\}\ \mathsf{with}\ r\ \mathsf{when}\ e\ \mathsf{do}\ c\ \{Q\}}$$

provided: $(\exists \Gamma'.\ P)$ is precise, Q' is precise, and $\mathrm{dom}(\Gamma') \cap \mathsf{fv}(P, Q, e, c) = \{\}$

Besides nil and the basic annotated triple, interference commands can be formed using sequential composition: $L;\ L$ and parallel composition: $L \parallel L$. The traces of $L_0;\ L_1$ are the traces of L_0 concatenated with the traces of L_1. The traces of $L_0 \parallel L_1$ are the interleavings of the traces from L_0 with the traces from L_1. The notion of execution of sets of traces is fairly simple. For example, the interference command

$$\mathsf{with}\ r\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1);\ \mathsf{with}\ r'\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1 \vee x \overset{\top}{\mapsto} j+2)$$

will, when executed from the heap $[x \mapsto (i, \top)]$, return the set

$$\{[x \mapsto (i+2, \top)], [x \mapsto (i+3, \top)]\},$$

and the interference command

$$\mathsf{with}\ r\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1 \vee x \overset{\frac{1}{2}}{\mapsto} j);\ \mathsf{with}\ r'\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1)$$

will abort when executed from $[x \mapsto (i, \top)]$ as the heap $[x \mapsto (i, \frac{1}{2})]$ is in the result of executing the first triple, but the second triple aborts when executed from this heap. In general, if $L \vdash \{P\}\ c\ \{Q\}$ and the execution of L from the share heap returns a set X, then every terminating execution of c in isolation must terminate with a shared heap in X.

With the interleaving of traces, the duration of the execution of two triples $\mathsf{with}\ r\ \mathsf{do}\ (\Gamma, P, Q)$ and $\mathsf{with}\ r'\ \mathsf{do}\ (\Gamma', P', Q')$ may overlap, and in that case the two triples must be executed in separate heaps. If $r = r'$ then the duration of the execution of the two triples will never overlap. For example, the execution of the interference command:

$$\mathsf{with}\ r\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \parallel \mathsf{with}\ r'\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1)$$

from the heap $[x \mapsto (i, \top)]$ aborts when $r \neq r'$, as there is an interleaving where the two triples overlap and no heap is in $x \overset{\top}{\mapsto} j * x \overset{\top}{\mapsto} j'$ for any j and j'. If $r = r'$, then the triples will never overlap and the execution is safe.

We can use the parallel and sequential composition of interference commands to compose specifications for commands. For instance, the following is a valid specification:

$$\begin{array}{l}\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1);\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1)\\ \vdash \{\mathsf{emp}\}\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (\mathsf{let}\ y{=}[x]\ \mathsf{in}\ [x]{:=}y+1);\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (\mathsf{let}\ y{=}[x]\ \mathsf{in}\ [x]{:=}y+1)\ \{\mathsf{emp}\}\end{array}$$

The sequential composition of the triples indicate that the command will have released r_0 before it acquires r_1 and thus the interference caused in the first critical region is finished before the interference from the second critical region begins.

Another valid specification is:

$$\begin{array}{l}\mathsf{with}\ r\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \parallel \mathsf{with}\ r'\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1)\\ \vdash \{\mathsf{emp}\}\ \mathsf{with}\ r\ \mathsf{do}\ (\mathsf{let}\ y{=}[x]\ \mathsf{in}\ [x]{:=}y+1) \parallel \mathsf{with}\ r'\ \mathsf{do}\ (\mathsf{let}\ y{=}[x]\ \mathsf{in}\ [x]{:=}y+1)\ \{\mathsf{emp}\}\end{array}$$

In this case we have seen that the interference command aborts when executed from a heap $[x \mapsto (i, \top)]$. When this happens the meaning of the specification is essentially vacuously true. The culprit is that validity of the specification assumes that the interference command does not abort when executed from a shared heap. In the next section I will introduce a logic to show the absence of such aborts, and we will see an easy proof for the case where $r = r'$. And of course, there will not be a proof for the case where $r \neq r'$.

The specifications given above are instances of these two rules:

$$\frac{L_0 \vdash \{P\}\ c_0\ \{I\} \qquad L_1 \vdash \{I\}\ c_1\ \{Q\}}{L_0; L_1 \vdash \{P\}\ c_0;\ c_1\ \{Q\}}$$

$$\frac{L_0 \vdash \{P_0\}\ c_0\ \{Q_0\} \qquad L_1 \vdash \{P_1\}\ c_1\ \{Q_1\}}{L_0 \parallel L_1 \vdash \{P_0 * P_1\}\ c_0 \parallel c_1\ \{Q_0 * Q_1\}}$$

3.1.1 A pre-order on interference commands

In a specification $L \vdash \{P\}\ c\ \{Q\}$ the interference command L lets the programmer describe the effect of c on the shared heap. The description can contain a lot of information that may not be needed. For instance, from the interference commands in the following valid specification:

$$\begin{array}{l}\mathsf{with}\ r\ \mathsf{do}\ (j:\mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1);\ \mathsf{with}\ r'\ \mathsf{do}\ (j:\mathsf{int}, y \overset{\top}{\mapsto} j, y \overset{\top}{\mapsto} j+1)\\ \quad \vdash \{\mathsf{emp}\}\ \mathsf{with}\ r\ \mathsf{do}\ (\mathsf{let}\ z{=}[x]\ \mathsf{in}\ [x]{:=}z+1);\ \mathsf{with}\ r'\ \mathsf{do}\ (\mathsf{let}\ z{=}[y]\ \mathsf{in}\ [y]{:=}z+1)\ \{\mathsf{emp}\}\end{array}$$

we can see that r is acquired and released before r' is acquired. This information may not be relevant and in that case we can erase the information and still have a valid specification by replacing the semi-colon with a parallel bar:

$$\begin{array}{l}\mathsf{with}\ r\ \mathsf{do}\ (j:\mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \parallel \mathsf{with}\ r'\ \mathsf{do}\ (j:\mathsf{int}, y \overset{\top}{\mapsto} j, y \overset{\top}{\mapsto} j+1)\\ \quad \vdash \{\mathsf{emp}\}\ \mathsf{with}\ r\ \mathsf{do}\ (\mathsf{let}\ z{=}[x]\ \mathsf{in}\ [x]{:=}z+1);\ \mathsf{with}\ r'\ \mathsf{do}\ (\mathsf{let}\ z{=}[y]\ \mathsf{in}\ [y]{:=}z+1)\ \{\mathsf{emp}\}\end{array}$$

With the parallel bar all we know is that the update to the shared heap, guarded by r, may happen before, after, or concurrently with the update to the shared heap guarded by r'. We say that

$$\mathsf{with}\ r\ \mathsf{do}\ (j:\mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \parallel \mathsf{with}\ r'\ \mathsf{do}\ (j:\mathsf{int}, y \overset{\top}{\mapsto} j, y \overset{\top}{\mapsto} j+1)$$

is above

$$\mathsf{with}\ r\ \mathsf{do}\ (j:\mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1);\ \mathsf{with}\ r'\ \mathsf{do}\ (j:\mathsf{int}, y \overset{\top}{\mapsto} j, y \overset{\top}{\mapsto} j+1).$$

I define $L \sqsubseteq L'$ when L is below L', i.e., when L' is more non-deterministic than L or when L' aborts when executed from states that L is safe on. The relation $- \sqsubseteq -$ will be a pre-order and I will define it in terms of the semantics of interference commands. Finally, I will give inference rules to derive $L \sqsubseteq L'$.

Say we have an interference command L with an L-typed hole and $L[x]$ is L with the hole replaced with x well-typed in Γ_0. Then the interference command $L[\mathsf{with}\ r\ \mathsf{do}\ (\Gamma', P', Q')]$ is above the interference command $L[\mathsf{with}\ r\ \mathsf{do}\ (\Gamma, P, Q)]$ if the triple (Γ', P', Q'), well-typed in Γ_0, is less deterministic or aborts on heaps that the triple (Γ, P, Q), also well-typed in Γ_0, is safe on. We can show the triple (Γ', P', Q') to be above the triple (Γ, P, Q) if $P' \Rightarrow \exists\Gamma.\ (P * R)$ and $(\exists\Gamma.\ (Q * R)) \Rightarrow Q'$, for some assertion R. The first condition captures that if a heap h satisfies P' then (Γ', P', Q') doesn't abort when executed from h, and thus (Γ, P, Q) must not abort when executed from h, i.e., h must satisfy $\exists\Gamma.\ (P * R)$ for some R. The second condition captures that the result of executing (Γ, P, Q) from h must be a subset of the result of executing (Γ', P', Q') from h, i.e., (Γ, P, Q) is less deterministic than (Γ', P', Q').

Interference commands $L[x]$ respect the pre-order $- \sqsubseteq -$, i.e, $L[L_0] \sqsubseteq L[L_1]$ if $L_0 \sqsubseteq L_1$, and with transitivity of $- \sqsubseteq -$ we can easily give proofs of $L \sqsubseteq L'$ for compound expressions L and L'. With a *rule of consequence*:

$$\frac{L \vdash \{P\}\ c\ \{Q\}}{L' \vdash \{P'\}\ c\ \{Q'\}} \text{ when } L \sqsubseteq L',\ P' \Rightarrow P, \text{ and } Q \Rightarrow Q'$$

we can easily weaken specifications $L \vdash \{P\}\ c\ \{Q\}$ as described above.

3.2 Protocol Safety

Next I introduce a specification $G \Vdash \{\lambda n : \tau.\ P\}\ L\ \{\lambda n' : \tau.\ Q\}$ to show *protocol safety* of L. Here, G represents a protocol for the interaction between L and an environment. The assertion P is the pre-condition, Q is the post-condition, the binder $\lambda n : \tau$ binds the identifier n in P and $\lambda n'$ binds n' in Q.

A *protocol* is a quintuple consisting of:

- A set N of nodes.
- A finite set R of resource names and possibly a unit.
- A subset L of $N \times N \times R$ defining the *local* edges.
- A subset E of $N \times N \times R$ defining the *environment* edges.
- A function A of type $N \to R \to \mathcal{P}_{\mathsf{prec}}(\mathbb{H})$, where $\mathcal{P}_{\mathsf{prec}}(\mathbb{H})$ is the set of precise subsets of the set $\mathbb{H}$ of heaps.

such that for all r and r' from R and for all nodes n and n', if $r \neq r'$ and $(n, n', r) \in L \cup E$, then

$$A(n)(r') \subseteq A(n')(r'). \tag{Pwf}$$

Given a node n_0 from N and a resource name r_0 from R, $A(n_0)(\mathsf{r}_0)$ contains the part of the shared heap that at node n_0 is either owned by r_0 or a process that possesses r_0. If the unit element uu is a member of R then $A(n_0)(\mathsf{uu})$ contains the part of the shared heap that is not owned by any resource or process. We will refer to A as the heap function of a protocol.

We will let G range over protocol expressions, i.e., quintuples (τ, R, l, e, A) where τ is a type, R is a type inhabited by a finite number of resources and possibly uu, l and e are expressions of type $(\tau \times \tau \times R)$ set, and A is an expression of type $\tau \to R \to \mathsf{passn}$. The meaning of a quintuple G, well-typed in a context Γ, is then a family of *protocols* indexed by the meaning of Γ. We will call such quintuples protocol expressions. A protocol expression (τ, R, l, e, A)

is well-formed if for all r and r' of type R and n and n' of type τ, if $(n, n', r) \in l \cup e$ and $r \neq r'$, then

$$A(n)(r') \Rightarrow A(n')(r').$$

When R contains more than one element the well-formedness condition (Pwf) is non-trivial, and it ensures that if a process changes the heap owned by resource r, and thus takes us from one node to another, then the predicates describing the heaps owned by the other resources stay valid for these heaps. If R is a singleton, then (Pwf) holds vacuously.

The expression l defines the set of *local* edges for each pair of nodes. The boolean expression $(n, n', r) \in l$ is true if and only if there is a *local* edge from n to n' with a label r. Likewise the expression e defines the set of *environment* edges.

The following is an example of a simple family G_3 of protocols, parameterized by j, for the resource r_0:

$$\begin{aligned}
\big(&\{0,1\} \times \{0,1\}, \{\mathsf{r}_0\},\\
&\{(n_0, n_1, r) \mid (\pi_1(n_0) = 0 \wedge \pi_1(n_1) = 1 \wedge \pi_2(n_0) = \pi_2(n_1)) \vee\\
&\qquad\qquad\qquad (\pi_2(n_0) = 0 \wedge \pi_2(n_1) = 1 \wedge \pi_1(n_0) = \pi_1(n_1))\},\\
&\{\},\\
&\lambda n_0.\ \lambda r.\ x \overset{\top}{\mapsto} j + \pi_1(n_0) + 2\pi_2(n_0)\big).
\end{aligned}$$

We can depict the protocol by:

$$\begin{tikzcd}
\{\mathsf{r}_0 : x \overset{\top}{\mapsto} j\}_{(0,0)} \arrow[r, "\mathsf{r}_0"] \arrow[d, "\mathsf{r}_0"'] & \{\mathsf{r}_0 : x \overset{\top}{\mapsto} j+1\}_{(1,0)} \arrow[d, "\mathsf{r}_0"] \\
\{\mathsf{r}_0 : x \overset{\top}{\mapsto} j+2\}_{(0,1)} \arrow[r, "\mathsf{r}_0"'] & \{\mathsf{r}_0 : x \overset{\top}{\mapsto} j+3\}_{(1,1).}
\end{tikzcd}$$

where $\{\mathsf{r}_0 : -\}_n$ depicts at node n the constant function $\lambda r : \{\mathsf{r}_0\}.\ x \overset{\top}{\mapsto} j + \pi_1(n) + 2\pi_2(n)$, i.e., the heap function of the protocol applied to node n. The arrows from node n to node n' depict the *local* edges, and there are no environment edges. Since there are no environment edges the protocol states the assumption that the environment will not perform any actions on the shared heap. Furthermore the local process may update the shared heap from a heap in $x \overset{\top}{\mapsto} i$ to a heap in $x \overset{\top}{\mapsto} i+1$ for $i = j$ or $i = j+2$ by following the horizontal arrows or from a heap satisfying $x \overset{\top}{\mapsto} i$ to a heap satisfying $x \overset{\top}{\mapsto} i+2$ for $i = j$ or $i = j+1$ by following the vertical arrows.

Let G be the protocol expression (τ, R, l, e, A). We will use the specification

$$G \Vdash \{\lambda n : \tau.\ P\}\ L\ \{\lambda n' : \tau.\ Q\}$$

to show that L follows the protocol described in G when executed in an environment that follows the protocol described in G. This specification deems the shared heap at node n to be a separation union of a heap from P that is currently local to L, and for each resource r in R that is currently free, a heap from $A(n)(r)$, and if $\mathsf{uu} \in R$, a heap from $A(n)(\mathsf{uu})$.

The following is an example of a valid specification using the protocol G_3:

$$\begin{aligned}
G_3 \Vdash \{\lambda n : \{0,1\} \times \{0,1\}.\ n = (0,0) \wedge \mathsf{emp}\}& \\
\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \parallel \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+2)& \\
\{\lambda n : \{0,1\} \times \{0,1\}.\ n = (1,1) \wedge \mathsf{emp}\}&
\end{aligned} \tag{3.1}$$

The specification implies that if we start in node $(0,0)$ and the shared heap is in $\mathsf{emp} * x \overset{\top}{\mapsto} j$ and if we execute the interference command:

$$\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \parallel \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+2),$$

then we don't abort, and we end up in node $(1,1)$ with the shared heap in $\mathsf{emp} * x \overset{\top}{\mapsto} j+3$.

To show this specification we must reason about the left interference command, with an environment performing the the actions of the right interference command, and vice versa. We first notice that G_3 was constructed such that the effect of the left interference command can be seen as a transition along any horizontal edge, and the effect of the right interference command can be seen as a transition along any vertical edge. Hence by dividing the local edges in G_3 into two disjoint sets, the horizontal edges as new local edges, and the vertical edges as new environment edges, we can construct a protocol G_1 suitable to reason about $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1)$ with an environment $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+2)$:

$$\begin{aligned}
\big(&\{0,1\} \times \{0,1\}, \{\mathsf{r}_0\}, \\
&\{(n_0, n_1, r) \ \mid\ \pi_1(n_0) = 0 \wedge \pi_1(n_1) = 1 \wedge \pi_2(n_0) = \pi_2(n_1)\}, \\
&\{(n_0, n_1, r) \ \mid\ \pi_2(n_0) = 0 \wedge \pi_2(n_1) = 1 \wedge \pi_1(n_0) = \pi_1(n_1)\}, \\
&\lambda n_0.\ \lambda r.\ x \overset{\top}{\mapsto} j + \pi_1(n_0) + 2\pi_2(n_0)\big)
\end{aligned}$$

which can be depicted by:

$$\begin{tikzcd}
\{\mathsf{r}_0 : x \overset{\top}{\mapsto} j\}_{(0,0)} \arrow[r, "\mathsf{r}_0"] \arrow[d, dotted, "\mathsf{r}_0"'] & \{\mathsf{r}_0 : x \overset{\top}{\mapsto} j+1\}_{(1,0)} \arrow[d, dotted, "\mathsf{r}_0"] \\
\{\mathsf{r}_0 : x \overset{\top}{\mapsto} j+2\}_{(0,1)} \arrow[r, "\mathsf{r}_0"'] & \{\mathsf{r}_0 : x \overset{\top}{\mapsto} j+3\}_{(1,1).}
\end{tikzcd} \tag{3.2}$$

where the full arrows are the *local* edges and the dotted arrows are the *environment* edges. From the picture of G_1 it should be easy to see how G_1 encodes the protocol that r_0 owns the heap location x and the *local* process may add one to the value at location x concurrently with an *environment* that adds two to the value at location x. Later we will show:

$$G_1 \Vdash \{\lambda n : \{0,1\} \times \{0,1\}.\ n = (0,0) \wedge \mathsf{emp}\} \\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \\ \{\lambda n : \{0,1\} \times \{0,1\}.\ (n = (1,1) \vee n = (1,0)) \wedge \mathsf{emp}\} \quad (3.3)$$

using a structural rule for annotated triples. We can interpret the specification (3.3) as follows: If we start in node $(0,0)$ then the local process will terminate in either node $(1,0)$ or node $(1,1)$. By inspecting G_1 we can see that this implies that if we execute $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1)$ from a shared heap satisfying $x \overset{\top}{\mapsto} j$ in an environment that may acquire r_0, add two to the value at x, and release r_0, then we don't abort and if we terminate then the shared heap will satisfy either $x \overset{\top}{\mapsto} j+1$ or $x \overset{\top}{\mapsto} j+3$.

Symmetrically we can show:

$$G_2 \Vdash \{\lambda n : \{0,1\} \times \{0,1\}.\ n = (0,0) \wedge \mathsf{emp}\} \\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+2) \\ \{\lambda n : \{0,1\} \times \{0,1\}.\ (n = (1,1) \vee n = (0,1)) \wedge \mathsf{emp}\} \quad (3.4)$$

where G_2 is the protocol:

$$\big(\{0,1\} \times \{0,1\}, \{\mathsf{r}_0\}, \\ \{(n_0, n_1, r) \mid \pi_2(n_0) = 0 \wedge \pi_2(n_1) = 1 \wedge \pi_1(n_0) = \pi_1(n_1)\}, \\ \{(n_0, n_1, r) \mid \pi_1(n_0) = 0 \wedge \pi_1(n_1) = 1 \wedge \pi_2(n_0) = \pi_2(n_1)\}, \\ \lambda(n_0, r).\ x \overset{\top}{\mapsto} j + \pi_1(n_0) + 2\pi_2(n_0)\big)$$

which can be depicted by (3.2) where the full arrows are the *environment* edges and the dotted arrows are the *local* edges.

We will then use a structural rule for parallel composition and a rule of consequence to derive (3.1). The parallel composition rule will spatially conjoin the pre-conditions and the post-conditions, intersect the sets of environment edges, and union the sets of local edges, and we get:

$$G_3' \Vdash \{\lambda n : \{0,1\} \times \{0,1\}.\ (n = (0,0) \wedge \mathsf{emp}) * (n = (0,0) \wedge \mathsf{emp})\} \\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+1) \parallel \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j+2) \\ \{\lambda n : \{0,1\} \times \{0,1\}.\ ((n = (1,1) \vee n = (0,1)) \wedge \mathsf{emp}) * ((n = (1,1) \vee n = (1,0)) \wedge \mathsf{emp})\}$$

where G_3' is:

$$
\begin{aligned}
&\big(\{0,1\} \times \{0,1\}, \{\mathsf{r}_0\},\\
&\ \{(n_0, n_1, r) \ \mid\ (\pi_1(n_0) = 0 \wedge \pi_1(n_1) = 1 \wedge \pi_2(n_0) = \pi_2(n_1))\} \cup\\
&\ \{(n_0, n_1, r) \ \mid\ \pi_2(n_0) = 0 \wedge \pi_2(n_1) = 1 \wedge \pi_1(n_0) = \pi_1(n_1)\},\\
&\ \{(n_0, n_1, r) \ \mid\ \pi_2(n_0) = 0 \wedge \pi_2(n_1) = 1 \wedge \pi_1(n_0) = \pi_1(n_1)\} \cap\\
&\ \{(n_0, n_1, r) \ \mid\ \pi_1(n_0) = 0 \wedge \pi_1(n_1) = 1 \wedge \pi_2(n_0) = \pi_2(n_1)\},\\
&\ \lambda(n_0, r).\ x \overset{\top}{\mapsto} j + \pi_1(n_0) + 2\pi_2(n_0)\big).
\end{aligned}
$$

G_3' is just another representation of G_3, i.e., G_3' and G_3 have the same nodes, edges, etc., and thus we can use the rule of consequence (RoC) given below to conclude (3.1).

The parallel rule has some important side-conditions. First we must ensure that the actions of $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j + 2)$ are captured by the environment edges in G_1. This condition holds if the local edges of G_2 are a subset of the environment edges of G_1. Secondly, even after $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j + 1)$ has terminated in a node n, $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j + 2)$ may continue and terminate in a node n'. If this happens then it is important that the post-condition in (3.3) in not invalidated by this last action. One simple way to ensure this is to require that for every environment edge from node n to node n' in G_1, if the post-condition in (3.3) holds at n for some heap h, then it holds at n' for the same heap h. With these two side-condition and their symmetric counterparts we have a sound reasoning principle for parallel interference commands (see RfP$'$ on page 17).

The Rule of Consequence (RoC) is a non-structural rule that allows the programmer to strengthen the pre-condition, weaken the post-condition, extend the set of local edges, and contract the set of environment edges:

$$
\frac{P \Rightarrow P' \qquad (\tau, R, l', e', A) \Vdash \{\lambda n : \tau.\ P'\}\ L\ \{\lambda n' : \tau.\ Q'\} \qquad Q' \Rightarrow Q}{(\tau, R, l, e, A) \Vdash \{\lambda n : \tau.\ P\}\ L\ \{\lambda n' : \tau.\ Q\}} \qquad \text{(RoC)}
$$

$$
\text{provided } l' \subseteq l, \text{ and } e \subseteq e'
$$

Strengthening the pre-condition $\lambda n.\ P'$ to $\lambda n.\ P$ means that P implies P' for every n. In other words, the set of nodes satisfying P for some heap h is smaller than the set of nodes satisfying P' for h. Hence the set of possible start nodes got smaller. Likewise weakening the post-condition may extend the set of nodes that we may terminate in. Adding local edges is also sound, as these extra edges are simply not used, and making the set of environment edges smaller is sound as we are simply limiting the possible actions of the environment.

The Rule for Parallel composition (RfP) is a structural rule enabling reasoning about parallel composition of interference commands:

$$\frac{\begin{array}{c}(\tau, R, l, e, A) \Vdash \{\lambda n : \tau .\ P\}\ L\ \{\lambda n' : \tau .\ Q\} \\ (\tau, R, l', e', A) \Vdash \{\lambda n : \tau .\ P'\}\ L'\ \{\lambda n' : \tau .\ Q'\}\end{array}}{(\tau, R, l \cup l', e \cap e', A) \Vdash \{\lambda n : \tau .\ P * P'\}\ L \,\|\, L'\ \{\lambda n' : \tau .\ Q * Q'\}} \qquad \text{(RfP)}$$

$$\begin{array}{ll}\text{provided} \quad l \subseteq e', & \text{for all } (n, n') \in \mathit{connected}(e),\ [n/n']Q * Q' \Rightarrow Q * \mathsf{true} \\ \qquad l' \subseteq e, & \text{and for all } (n, n') \in \mathit{connected}(e'),\ [n/n']Q' * Q \Rightarrow Q' * \mathsf{true}\end{array}$$

The set *connected*(e) is the transitive closure of $\{(n, n') \mid \exists r.\ (n, n', r) \in e\}$. Hence the pair (n_0, n_1) is in *connected* (e) if and only if there exists a path from n_0 to n_1 such that $\exists r.\ (n, n') \in e(r)$ for every edge (n, n') in the path. The condition $l' \subseteq e$ in the rule (RfP) ensures that in the proof for L, the effect of L' is captured by e. The condition

$$\text{for all pairs } (n, n') \in \mathit{connected}(e),\ [n/n']Q * Q' \Rightarrow [n'/n]Q * \mathsf{true}$$

ensures that if L terminates in node n and L' terminates in node n', then Q doesn't exclude the possibility that $L \,\|\, L'$ might terminate in n'. With the analogous condition for e' it is safe to conclude that $L \,\|\, L'$ will terminate in node n for some n such that $Q * Q'$ holds.

If Q is closed under *connected* (e), i.e., if for all edges (n, n') such that $\exists r.\ (n, n') \in e(r)$, Q holds for heap h at node n implies that Q holds for heap h at node n', then we can derive the simpler, but weaker, rule (RfP$'$):

$$\frac{\begin{array}{c}(\tau, R, l, e, A) \Vdash \{\lambda n : \tau .\ P\}\ L\ \{\lambda n' : \tau .\ Q\} \\ (\tau, R, l', e', A) \Vdash \{\lambda n : \tau .\ P'\}\ L'\ \{\lambda n' : \tau .\ Q'\}\end{array}}{(\tau, R, l \cup l', e \cap e', A) \Vdash \{\lambda n : \tau .\ P * P'\}\ L \,\|\, L'\ \{\lambda n' : \tau .\ Q * Q'\}} \qquad \text{(RfP}'\text{)}$$

$$\begin{array}{lll}\text{provided } l \supseteq e', & [n/n']Q \Rightarrow Q & \text{for all } (n, n', r) \in e \\ \qquad\quad l' \supseteq e, & [n/n']Q' \Rightarrow Q' & \text{for all } (n, n', r) \in e'\end{array}$$

Notice that the post-conditions in (3.3) and (3.4) are easily shown to satisfy these side-conditions, thus we can use this simpler rule to reason about the parallel composition in that example.

Finally we should look back at (3.3). The specification claims that if we start in node $(0, 0)$, then we will terminate safely in either node $(0, 1)$ or node $(1, 1)$. To show this we will first show:

$$\begin{array}{l} G_1 \Vdash \{\lambda n : \{0, 1\} \times \{0, 1\}.\ (n = (0, 1) \vee n = (0, 0)) \wedge \mathsf{emp}\} \\ \qquad\qquad \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j + 1) \\ \qquad\qquad\quad \{\lambda n : \{0, 1\} \times \{0, 1\}.\ (n = (1, 1) \vee n = (1, 0)) \wedge \mathsf{emp}\} \end{array} \qquad (3.5)$$

and then use the rule of consequence to conclude (3.3). Notice that the pre-condition $(n = (0, 1) \vee n = (0, 0)) \wedge \mathsf{emp}$ implies $(n' = (0, 1) \vee n' = (0, 0)) \wedge \mathsf{emp}$ for every environment edge from n to n' in G_1 (the only such environment edge goes from node $(0, 0)$

to $(0, 1)$). This means that if at node n we have a heap h satisfying the pre-condition and the environment takes us to a node n', then the pre-condition also holds for h at node n'. Hence the action of the environment doesn't invalidate the description of the heap currently local to $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j + 1)$. In general, in the rule to show

$$(\tau, R, l, e, A) \Vdash \{\lambda n.\ P\}\ \mathsf{with}\ r\ \mathsf{do}\ (\Gamma', P', Q')\ \{\lambda n'.\ Q\}$$

this property of P is guaranteed by the side-condition $P \Rightarrow [n'/n]P$ for all $(n, n', r') \in e$. Next we need to show that $\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (j : \mathsf{int}, x \overset{\top}{\mapsto} j, x \overset{\top}{\mapsto} j{+}1)$ doesn't abort when executed from the separate union of the shared heap owned by r_0 at node n, in the general case, a heap in $A(n)(r)$, and the heap currently local to the process, in the general case, a heap from P. This holds if this heap can be split into two separate heaps such that one of them is in $x \overset{\top}{\mapsto} j$ for some j. In the general case, we require a proof of $A(n)(r) * P \Rightarrow [S/\Gamma']P' * Y$ for some substitution S and assertion Y. The part of the heap from Y is preserved by the triple and thus if the triple terminates with a set X of heaps, then every heap in X must be in $[S/\Gamma']Q' * Y$. Next we must show that for every heap in X there is a local edge from node n to some node n' such that the heap can be split into a heap satisfying the post-condition at node n' and a heap satisfying the assertion for r at node n', i.e., $[S/\Gamma']Q' * Y \Rightarrow \exists n'.\ (n, n', r) \in l \wedge A(n')(r) * Q$.

We can summarize these conditions in the rule for annotated triples:

$$\frac{P * A(n)(r) \Rightarrow [S/\Gamma']P' * Y \qquad [S/\Gamma']Q' * Y \Rightarrow \exists n'.\ (n, n', r) \in l \wedge A(n')(r) * Q}{(\tau, R, l, e, A) \Vdash \{\lambda n.\ P\}\ \mathsf{with}\ r\ \mathsf{do}\ (\Gamma', P', Q')\ \{\lambda n'.\ Q\}}$$

$$\text{provided } r \in R \text{ and } P \Rightarrow [n'/n]P \text{ for all } (n, n', r') \in e. \tag{Res}$$

3.2.1 Multiple Resources

Defining protocols for more than one resource name is fairly easy. The program (1.1) on page 4 is not in our programming language because it uses assignable variables, but a slightly different program has the exact same problem and is in our programming language:

$$\begin{aligned}&\mathsf{resource}\ \mathsf{r}_0, \mathsf{r}_1\ \mathsf{in}\\ &\qquad\qquad\qquad \big(\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ [a_0]{:=}1;\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ [a_1]{:=}1\big) \parallel\\ &\qquad \big(\mathsf{with}\ \mathsf{r}_1\ \mathsf{when}\ [a_1] \neq 0\ \mathsf{do}\ \mathsf{skip};\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ \mathsf{if}\ [a_0] = 0\ \mathsf{then}\ \mathsf{crash}\ \mathsf{else}\ \mathsf{skip}\big).\end{aligned} \tag{3.6}$$

Let L be:

$$\begin{aligned}&\Big(\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (-, a_0 \overset{\top}{\mapsto} -, a_0 \overset{\top}{\mapsto} 1);\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (-, a_1 \overset{\top}{\mapsto} -, a_1 \overset{\top}{\mapsto} 1)\Big) \parallel\\ &\qquad \Big(\mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (j : \mathsf{int}, a_1 \overset{\top}{\mapsto} j, a_1 \overset{\top}{\mapsto} j \wedge j \neq 0);\ \mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (-, a_0 \overset{\top}{\mapsto} 1, a_0 \overset{\top}{\mapsto} 1)\Big)\end{aligned}$$

Then the following specification is valid:

$$L \vdash \{\mathsf{emp}\}\ c\ \{\mathsf{emp}\} \tag{3.7}$$

where $\mathsf{resource}\ \mathsf{r}_0, \mathsf{r}_1\ \mathsf{in}\ c$ is the command in (3.6). Notice how the condition $[a_1] \neq 0$ is represented in the interference command:

$$\mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (j : \mathsf{int}, a_1 \overset{\top}{\mapsto} j, a_1 \overset{\top}{\mapsto} j \wedge j \neq 0).$$

This interference command aborts after it has acquired r_1, unless the shared heap has a part satisfying $a_1 \overset{\top}{\mapsto} j$ for some j, and it terminates if and only if $j \neq 0$, i.e., $[a_1]$ is not 0. Hence if $[a_1] \neq 0$ is false then the interference command will neither abort nor terminate.

Next we would like to specify that the value at a_1 is initially 0 and is updated after the value at a_0 is set to 1. Before we can encode this as a protocol we must provide appropriate ownership information. As the triples annotated with r_0 in the interference command assumes full ownership over location a_0 we will let r_0 have full ownership of the location a_0. Likewise we will let r_1 have full ownership of the location a_1. This way of deeming ownership of a_0 and a_1 works only when $a_0 \neq a_1$. If $a_0 = a_1$ then the program aborts as the right process may read from the heap cell a_1 while the left process writes to the heap cell a_0. The specification (3.7) is still valid, but L aborts.

One way to encode the protocol is the following G:

$$\begin{tikzcd}
\{\mathsf{r}_0 : a_0 \overset{\top}{\mapsto} - \quad \mathsf{r}_1 : a_1 \overset{\top}{\mapsto} 0\}_0 \arrow[r, "\mathsf{r}_0"] & \{\mathsf{r}_0 : a_0 \overset{\top}{\mapsto} 1 \quad \mathsf{r}_1 : a_1 \overset{\top}{\mapsto} -\}_1 \arrow[loop above, "\mathsf{r}_0"] \arrow[loop below, "\mathsf{r}_1"]
\end{tikzcd}$$

This protocol can be represented by:

$$\Big(\{0,1\}, \{\mathsf{r}_0, \mathsf{r}_1\}, \lambda(n, n', r).\ ((n = 0 \wedge r = \mathsf{r}_0) \vee n = 1) \wedge n' = 1, \lambda(n, n', r).\ \mathsf{false},$$
$$\lambda(n, r).\ n = 0 \wedge \Big(r = \mathsf{r}_0 \wedge a_0 \overset{\top}{\mapsto} - \vee r = \mathsf{r}_1 \wedge a_1 \overset{\top}{\mapsto} 0\Big) \vee$$
$$n = 1 \wedge \Big(r = \mathsf{r}_0 \wedge a_0 \overset{\top}{\mapsto} 1 \vee r = \mathsf{r}_1 \wedge a_1 \overset{\top}{\mapsto} -\Big)\Big).$$

One might also use a protocol with three nodes:

$$\begin{tikzcd}
\{\mathsf{r}_0 : a_0 \overset{\top}{\mapsto} - \quad \mathsf{r}_1 : a_1 \overset{\top}{\mapsto} 0\}_0 \arrow[r, "\mathsf{r}_0"] & \{\mathsf{r}_0 : a_0 \overset{\top}{\mapsto} 1 \quad \mathsf{r}_1 : a_1 \overset{\top}{\mapsto} 0\}_1 \arrow[d, "\mathsf{r}_1"] \\
 & \{\mathsf{r}_0 : a_0 \overset{\top}{\mapsto} 1 \quad \mathsf{r}_1 : a_1 \overset{\top}{\mapsto} -\}_2 \arrow[loop above, "\mathsf{r}_0"] \arrow[loop below, "\mathsf{r}_1"]
\end{tikzcd}$$

but as a process transitioning an r-edge can only update the part of the shared heap owned by r, little information is gained by the extra node.

As the R-set in the protocol G contains multiple resource names we must check that the protocol is well-defined, i.e., satisfy (Pwf) on page 12. As there is only one non-trivial edge and $a_1 \overset{\top}{\mapsto} 0 \Rightarrow a_1 \overset{\top}{\mapsto} -$ this is fairly trivial by case analysis on r.

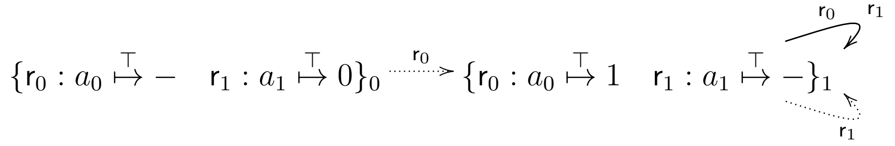
To show the specification:

$$G \Vdash \{\lambda n : \{0,1\}.\ n = 0 \wedge \mathsf{emp}\}\ L\ \{\lambda n : \{0,1\}.\ n = 1 \wedge \mathsf{emp}\} \tag{3.8}$$

we will first show:

$$\begin{aligned} G_1 \Vdash \{\lambda n.\ n = 0 \wedge \mathsf{emp}\}\ &\mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (j : \mathsf{int}, a_1 \overset{\top}{\mapsto} j, a_1 \overset{\top}{\mapsto} j \wedge j \neq 0); \\ &\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (-, a_0 \overset{\top}{\mapsto} 1, a_0 \overset{\top}{\mapsto} 1)\ \{\lambda n.\ n = 1 \wedge \mathsf{emp}\} \end{aligned} \tag{3.9}$$

where G_1 is the protocol:



where the dotted arrows are *environment* edges and the solid arrows are *local* edges. The protocol specifies that the environment may take us from the first to the second node by updating the value at a_0 to 1 while possessing r_0, and subsequently take us from the second node to itself while updating the value at a_1 and possessing r_1, and the local process will not perform any actions on the first node.

A key point to show validity of (3.9) is that the post-condition in the $\mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ -$ clause is false when j is 0 and thus the right process waits until we are in the seconds node (in the first node the pre-condition is only valid when j is 0). This is captured by the rule for annotated triples on page 18 which we instantiate with a substitution mapping j to the node n and the assertion Y to emp, and we conclude:

$$\begin{aligned} G_1 \Vdash \{\lambda n.\ (n = 0 \vee n = 1) \wedge \mathsf{emp}\}& \\ &\mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (j : \mathsf{int}, a_1 \overset{\top}{\mapsto} j, a_1 \overset{\top}{\mapsto} j \wedge j \neq 0)\ \{\lambda n.\ n = 1 \wedge \mathsf{emp}\}. \end{aligned}$$

With a specification for the other half of L using the protocol where the solid arrows are *environment* edges and the dotted arrows are *local* edges, we can use the parallel rule to conclude:

$$G' \Vdash \{\lambda n.\ (n = 0 \wedge \mathsf{emp}) * (n = 0 \wedge \mathsf{emp})\}\ L\ \{\lambda n.\ (n = 1 \wedge \mathsf{emp}) * (n = 1 \wedge \mathsf{emp})\}$$

for the protocol G':

$$\begin{tikzcd}
\{\mathsf{r}_0 : a_0 \overset{\top}{\mapsto} - \quad \mathsf{r}_1 : a_1 \overset{\top}{\mapsto} 0\}_0 \arrow[r, "\mathsf{r}_0"] & \{\mathsf{r}_0 : a_0 \overset{\top}{\mapsto} 1 \quad \mathsf{r}_1 : a_1 \overset{\top}{\mapsto} -\}_1 \arrow[loop above, "\mathsf{r}_0\ \ \mathsf{r}_1"] \arrow[loop below, dotted, "\mathsf{r}_1"]
\end{tikzcd}$$

where solid arrows are *local* edges and the dotted arrows are *environment* edges. We can then use the rule of consequence to show (3.8) by dropping the environment edge.

3.3 Interaction between Protocol Safety and Heap Safety

In section 3.1 and section 3.2 we saw how an interference command is the medium between the logic for heap safety and the logic for protocol safety, and we saw that in a proof of $L \vdash \{P\}\ c\ \{Q\}$ only the conditional critical regions contributed with non-trivial information to L. Sometimes it is useful to add information to L without trying to acquire a critical region. For instance, if a protocol requires that a process must wait for some non-trivial condition to be true, then we might need to move information between the local heap and the shared heap. Say a program must wait until the extended boolean expression $[a] = 0$ holds of the heap owned by r_0 or $[b] = 0$ holds of the heap owned by r_1. If initially x_0 and x_1 points to 1 then the following program performs such a wait:

$$\begin{array}{l}
\mathsf{while}\ (([x_0] \neq 0) \wedge ([x_1] \neq 0))\ \mathsf{do} \\
\ (\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ \mathsf{let}\ x_0'{=}[a]\ \mathsf{in}\ [x_0]{:=}x_0'; \\
\ \ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ \mathsf{let}\ x_1'{=}[b]\ \mathsf{in}\ [x_v]{:=}x_1')
\end{array}$$

We can move the termination condition for the while-loop into the interference specification by deeming half ownership of x_0 and x_1 to the local process and the other half to the shared heap. We can show:

$$\vdash \{x_0 \overset{\frac{1}{2}}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} -\}\ ([x_0] \neq 0) \wedge ([x_1] \neq 0)$$
$$\{\lambda r : \mathsf{bool}.\ \exists x_0', x_1'.\ x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge ((x_0' \neq 0) \wedge (x_1' \neq 0)) = r\}$$

and

$$L \vdash \{\exists x_0', x_1'.\ x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge (x_0' \neq 0) \wedge (x_1' \neq 0)\}$$
$$\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (\mathsf{let}\ x_0'{=}[a]\ \mathsf{in}\ [x_0]{:=}x_0');\ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (\mathsf{let}\ x_1'{=}[b]\ \mathsf{in}\ [x_1]{:=}x_1')$$
$$\{x_0 \overset{\frac{1}{2}}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} -\}$$

where L is

$$\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ (-, a \overset{\top}{\mapsto} - * x_0 \overset{\frac{1}{2}}{\mapsto} -, a \overset{\top}{\mapsto} - * x_0 \overset{\frac{1}{2}}{\mapsto} -);$$
$$\mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ (-, b \overset{\top}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} -, b \overset{\top}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} -).$$

Next we can use the rule for while loops:

$$\frac{\vdash \{I\}\ e\ \{\lambda b : \mathsf{bool}.\ I'\} \qquad L \vdash \{[\mathsf{true}/b]I'\}\ c\ \{I\}}{L^* \vdash \{I\}\ \mathsf{while}\ e\ \mathsf{do}\ c\ \{[\mathsf{false}/b]I'\}}$$

where L^* is equivalent to $(L;\ L^*) + \mathsf{nil}$, and conclude:

$$\begin{array}{l}
L^* \vdash \{x_0 \overset{\frac{1}{2}}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} -\} \\
\qquad\qquad\qquad \mathsf{while}\ ([x_0] \neq 0 \wedge [x_1] \neq 0)\ \mathsf{do} \\
\qquad\qquad\qquad\ \ (\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ \mathsf{let}\ x_0'{=}[a]\ \mathsf{in}\ [x_0]{:=}x_0'; \\
\qquad\qquad\qquad\ \ \ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ \mathsf{let}\ x_1'{=}[b]\ \mathsf{in}\ [x_1]{:=}x_1') \\
\qquad\qquad\qquad\qquad \{\exists x_0', x_1'.\ x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge (x_0' \neq 0 \wedge x_1' \neq 0) = \mathsf{false}\}.
\end{array}$$

To move the fact that the command terminates when $[x_0] \neq 0 \wedge [x_1] \neq 0$ is false, I will introduce the possibility of an atomic repartition of the separation between the local and the shared heap. I will extend the syntax for interference expressions with $\mathsf{with\ uu\ do}\ (\Gamma, P, Q)$. The unit uu is used only in protocols and interference commands where it indicates that the repartitioning is atomic and not related to any particular resource. I provide two rules to introduce repartitionings

$$\frac{L \vdash \{P\}\ c\ \{Q\}}{\mathsf{with\ uu\ do}\ (\Gamma, P_0, P_1);\ L \vdash \{P'\}\ c\ \{Q\}}$$

$$\begin{array}{c}
\text{provided } P' * P_0 \Rightarrow P_1 * P,\ \ \mathrm{dom}(\Gamma) \cap \mathsf{fv}(P, P', Q) = \{\}, \\
(\exists \Gamma.\ P_0) \text{ is precise, and } P_1 \text{ is precise}
\end{array}$$

$$\frac{L \vdash \{P\}\ c\ \{Q\}}{L;\ \mathsf{with\ uu\ do}\ (\Gamma, Q_0, Q_1) \vdash \{P\}\ c\ \{Q'\}}$$

$$\begin{array}{c}
\text{provided } Q * Q_0 \Rightarrow Q_1 * Q',\ \ \mathrm{dom}(\Gamma) \cap \mathsf{fv}(P, P', Q) = \{\}, \\
(\exists \Gamma.\ Q_0) \text{ is precise, and } Q_1 \text{ is precise}
\end{array}$$

I can then use the first rule to move information about the termination of the while loop into the interference command:

$$\begin{array}{l}
L^*;\ (x_0' : \mathsf{int}, x_1' : \mathsf{int}, x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1', x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge (x_0' = 0 \vee x_1' = 0)) \\
\qquad\qquad\qquad\qquad\qquad \vdash \{x_0 \overset{\frac{1}{2}}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} -\} \\
\qquad\qquad\qquad \mathsf{while}\ ([x_0] \neq 0 \wedge [x_1] \neq 0)\ \mathsf{do} \\
\qquad\qquad\qquad\ \ (\mathsf{with}\ \mathsf{r}_0\ \mathsf{do}\ \mathsf{let}\ x_0'{=}[a]\ \mathsf{in}\ [x_0]{:=}x_0'; \\
\qquad\qquad\qquad\ \ \ \mathsf{with}\ \mathsf{r}_1\ \mathsf{do}\ \mathsf{let}\ x_1'{=}[b]\ \mathsf{in}\ [x_1]{:=}x_1') \\
\qquad\qquad\qquad\qquad\qquad \{\exists x_0', x_1'.\ x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge (x_0' = 0 \vee x_1' = 0)\}.
\end{array}$$

Given a protocol G:

$$\begin{tikzcd}
\{\mathsf{r}_0 : a \overset{\top}{\mapsto} - \quad \mathsf{r}_1 : b \overset{\top}{\mapsto} -\}_0 \arrow[loop left, "\mathsf{r}_0{,}\mathsf{r}_1"] \arrow[r, "\mathsf{r}_0"] \arrow[d, "\mathsf{r}_1"'] & \{\mathsf{r}_0 : a \overset{\top}{\mapsto} 0 \quad \mathsf{r}_1 : b \overset{\top}{\mapsto} -\}_1 \arrow[loop right, "\mathsf{r}_0{,}\mathsf{r}_1"] \\
\{\mathsf{r}_0 : a \overset{\top}{\mapsto} - \quad \mathsf{r}_1 : b \overset{\top}{\mapsto} 0\}_2 \arrow[loop below, "\mathsf{r}_0{,}\mathsf{r}_1"] &
\end{tikzcd}$$

we can then prove that if we start in node 0, then the interference command doesn't abort and if it terminates then it must terminate in node 1 or 2. We want to show

$$\begin{aligned}
&G \Vdash \{\lambda n.\ (n = 0 \wedge x_0 \overset{\frac{1}{2}}{\mapsto} 1 * x_1 \overset{\frac{1}{2}}{\mapsto} 1)\} \\
&\qquad L^*;\ (x_0' : \mathsf{int}, x_1' : \mathsf{int}, x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1', x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge (x_0' = 0 \vee x_1' = 0)) \\
&\qquad\qquad \{\lambda n.\ (n = 1 \wedge x_0 \overset{\frac{1}{2}}{\mapsto} 0 * x_1 \overset{\frac{1}{2}}{\mapsto} -) \vee (n = 2 \wedge x_0 \overset{\frac{1}{2}}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} 0)\}
\end{aligned}$$

using the rule for sequential composition:

$$\frac{G \Vdash \{\lambda n.\ P\}\ L_0\ \{\lambda n.\ R\} \quad G \Vdash \{\lambda n.\ R\}\ L_1\ \{\lambda n.\ Q\}}{G \Vdash \{\lambda n.\ P\}\ L_0; L_1\ \{\lambda n.\ Q\}}$$

where R will be

$$\begin{aligned}
&\lambda n.\ (n = 0 \wedge \exists x_0', x_1'.\ x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge x_0' \neq 0 \wedge x_1' \neq 0) \vee \\
&\qquad\qquad (n = 1 \wedge x_0 \overset{\frac{1}{2}}{\mapsto} 0 * x_1 \overset{\frac{1}{2}}{\mapsto} -) \vee (n = 2 \wedge x_0 \overset{\frac{1}{2}}{\mapsto} - * x_1 \overset{\frac{1}{2}}{\mapsto} 0).
\end{aligned}$$

Notice that at node 0 the assertion R insists that x_0 and x_1 both points to non-zero values. Hence when we apply the triple

$$(x_0' : \mathsf{int}, x_1' : \mathsf{int}, x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1', x_0 \overset{\frac{1}{2}}{\mapsto} x_0' * x_1 \overset{\frac{1}{2}}{\mapsto} x_1' \wedge (x_0' = 0 \vee x_1' = 0))$$

to any heap from R at node 0, the post-condition of the triple is false, and thus the triple does not terminate at node 0, and thus the original while loop cannot have terminated at node 0.

We introduce a simple rule

$$\frac{}{G \Vdash \{\lambda n.\ [S/\Gamma]P\}\ \mathsf{with\ uu\ do}\ (\Gamma, P, Q)\ \{\lambda n.\ [S/\Gamma]Q\}}$$

to encapsulate and generalize the above reasoning. Notice that with this rule an atomic repartitioning can only interact with the part of the shared heap that is currently local to the interference command. If the programmer has specified atomic repartitions in his protocol, then he should use the rule for annotated triples (Res on page 18).

Finally we should give a rule to reason about L^*, i.e., an arbitrary number of sequential iterations of L:

$$\frac{G \Vdash \{\lambda n.\ P\}\ L\ \{\lambda n.\ P\}}{G \Vdash \{\lambda n.\ P\}\ L^*\ \{\lambda n.\ P\}}$$

4 Compositionality

The goal of modularity of a program logic is to enable the programmer to divide his program into separate components, give each component a specification, and then derive specifications of entire programs by composing the specifications of the components used in his program. Compositionality of a program logic, i.e., the property that a proof in the program logic follows the syntax for programs, is often seen as way to provide modularity to program logics.

CSL and Rely-Guarantee are compositional program logics and as parallel composition is a simple syntactic construction, it is often said that these logics are *thread modular*, i.e., you should be able to give a specification for each thread and then compose these specification into a specification for the entire program. But as we have seen in section 1 the specifications of threads often need to encode the actions of the environment, i.e., the other threads in the program. Hence the specification you need for a given thread depends on the other threads (or modules) used in the program, and thus the programmer often needs to look at the command of a thread, i.e., open the module, to give a proof of the needed specification. In other words, thread modularity of CSL and Rely-Guarantee does not enable the programmer to forget about the implementations of the threads in the environment.

In the examples given in the previous sections, i.e., the running example in section 3.1 and the example in section 3.2.1, the two level logic enabled us to give *local* specifications $L \vdash \{P\}\ c\ \{Q\}$ that can be used with many different environments. The specifications were *local* in the sense that they never referenced properties of the environment. Instead the interference command L encoded the assumptions that the programmer made about the shared heap. Only at the level with specifications $G \Vdash \{\lambda x.\ P\}\ L\ \{\lambda y.\ Q\}$ did we use properties of the environment. These properties were then encoded in the protocol G, and unlike CSL we often enabled the programmer to simplify the protocol after he has composed specifications of multiple threads (or modules). Hence in the proposed logic, we can often reuse specifications of the form $L \vdash \{P\}\ c\ \{Q\}$ and thus obtain true thread modularity on that level, and protocols do not always need to expose every detail of the computation performed by an interference command.

5 Related Work

The introduction of Separation Logic [IO01, Rey02] by O'Hearn and Reynolds simplified reasoning about heap manipulating sequential programs. Separation logic enables the programmer to implicitly specify non-aliasing constraints using spatial logical connectives: points-to $e \mapsto e'$, and separating conjunction $P * Q$. Separation logic has since been extended to a higher-order logic by Biering et al. [BBTS07], enabling the programmer to define inductive data structures directly in the logic, to reason about commands in a context of procedures with abstract specifications, and to reason polymorphically about data structures.

With concurrent separation logic (CSL) O'Hearn [O'H07] showed how separation logic can be used to reason about concurrent first-order programs with heaps. O'Hearn extended the programming language defined by Owicki in [Owi75] where interaction between processes is limited to conditional critical regions parametrized by a resource name, with operations for looking up values and storing values in a heap. He noticed that if two commands are executing on spatially separated heaps, then they can safely execute in parallel, whence his version of Owicki's rule for parallel composition uses separating conjunction instead of ordinary conjunction. The semantics of the language guarantees that the execution of any two conditional critical regions for the same resource name cannot overlap. To reason about the interaction between processes O'Hearn required, as Owicki did, that every resource has an invariant assertion and a list of variables protected by that resource, i.e., a list of variables where each variable is changed by a process only when the resource is possessed by that process. When entering a critical region the programmer can then assume the invariant for that region holds separately from his current local heap and upon exiting a critical region he must show that the local heap can be split into a new local heap and separately a heap that holds of the invariant. Again, O'Hearn's adoption of Owicki's parallel rule uses separating conjunction instead of ordinary conjunction. With the introduction of separation logic came also the notion of ownership transfer [OYR04] and in CSL ownership transfer happens when entering and exiting critical regions. Reynolds had discovered [OYR04] that precision of invariants is needed to reason soundly about ownership transfer, and soon after Brookes gave a proof of soundness of CSL using trace semantics [Bro04, Bro07, Bro11]. This proof makes it clear how precision of the resource invariants ensures that the transfer of ownership of heap cells from a process to a resource is well-defined.

5.1 Permissions

Extensions of Concurrent Separation Logic with Boyland's fractional permissions [Boy03] and Bornat's counting permissions [BCOP05], enables the programmer to better keep track of "ownership transfer" and "read only" locations and variables. In the programming language of CSL the scope of a variable is static and the rules of CSL has static side-conditions for proper treatment of variables. With the introduction of permissions on variables [BCY06, PBC06] the logic was changed such that it treated variables in much the same way it treated heap cells, i.e., ownership or permission to a variable can change dynamically as the program executes. This created a mismatch between the treatment of variables in the logic and in the programming language which has recently been cleaned up by Reddy and Reynolds with syntactic control of interference for separation logic [RR12].

The separation logic proposed in section 3 combines the higher-order separation logic from [BBTS07] with fractional permissions as in [BCOP05] to get a separation logic suitable to keep track of ownership transfer and abstract specifications of commands. Permissions are easily embedded into the higher-order logic by defining a type perm of permissions and suitable constants and operations on perm. The triple (Γ, P, Q) used to reason about the command in a conditional critical region is essentially an abstract specification of the command, in the sense of [BBTS07].

5.2 Semantics

In the sections above we hinted at the semantics underlying the programming language and the interference commands. Following Brookes [Bro07], the semantics of the programming language is a trace semantics with actions for every little step of the execution of a command. In this semantics only some traces correspond to possible executions from a given store and heap. Brookes defined a global enabling relation to capture if a trace is enabled from a given store, heap, and set of currently possessed resources, and to describe the effect of an enabled trace of such a configuration. This semantics is known as a global semantics and it is independent of the logic. Brookes also defined a *local* enabling relation that accounts for properties of the logic such as ownership transfers and the isolation of processes with the exception of critical regions. He then defined validity of CSL judgments using the local enabling relation and showed soundness of the rules. Next he showed how to lift this result to the global enabling relation. Hence we can use CSL to reason about the execution of programs. I have taken similar steps to show soundness of the logic in section 3. I have defined a trace semantics for commands and a global enabling relation that is independent of the logic. I have defined a trace semantics for interference commands and I have defined a local enabling relation that connects a trace of a command to a trace of an interference command. This local enabling relation is then used to define validity of the judgment $L \vdash \{P\}\ c\ \{Q\}$. The semantics of interference commands uses a special class of *best local actions* (see [COY07]) to give meaning to triples (Γ, P, Q). Local actions [COY07] are a special class of functions defined on heaps that can be used to represent non-deterministic commands that satisfy *safety monotonicity* and the *frame property.* I will further restrict the class of local actions to represent the precision of $(\exists \Gamma.\ P)$ and Q, whence members of this class can be used to reason uniquely about ownership transfers. Next, I have defined a local enabling relation that relates a trace of an interference command to a protocol. This relation is used to define validity of the judgment $G \Vdash \{\lambda n.\ P\}\ L\ \{\lambda n'.\ Q\}$.

5.3 Higher-order Concurrent Separation Logic

In [JP11] Jacobs and Piessens introduced a higher-order version of concurrent separation logic (HCSL). The ability to quantify over commands and specifications enables them to freely compose HCSL specifications (CSL specifications in a context) in the same way that we compose $L \vdash \{P\}\ c\ \{Q\}$ specifications. The constraints of CSL are represented as types in a context, and one can obtain a CSL specification by substituting terms for the variables in this context. This enables them to quantify over predicates describing auxiliary state, resource invariants, commands that update auxiliary state, and specifications of these commands. Hence by introducing numerous variables ranging over predicates, commands, and specifications, they can delay the choice of these until the programmer wants to bind a resource r. At this point they must substitute a precise predicate for the invariant of r and thus determine a protocol for interaction via r. This pattern of specifications provides thread- and procedure-modularity to CSL at the expense of a large number of constraints in a context. A solution to these constraints results in a specification valid in CSL, and thus finding such solutions is as difficult as finding a CSL specification of the program in the first place. The constraints could also be inconsistent, and in that case we cannot construct a CSL specification from the HCSL specification. This is similar to the logic in section 3.1 where the interference commands may abort from the shared heap.

5.4 Rely-Guarantee

In the last few years there has been a major effort aimed at the development of Rely-Guarantee logics for concurrent first-order programs with mutable state. At first, separation logic was used to extend the assertions and relations to range over heaps. Rely-Guarantee was traditionally used to reason about concurrent programs with a single shared store [Owi75, Jon81, Jon83a, Jon83b], but in [VP07] and [FFS07] the heap was split into a *local* part and a *shared* part. Assertions described each heap as a separate union of a local heap and a shared heap, and the $*$-operator was defined as ordinary conjunction on the shared heap and separation conjunction on the local heap. This definition enables the programmer to reason about the local heap much like he would in separation logic for sequential programs and special care had to be taken only when reasoning about the shared heap. In [VP07] this was limited to the special $\mathsf{atom}(c)$ program construction. When giving specifications for this construction the programmer is allowed access to the shared heap and as a result the machinery of Rely-Guarantee logics such as Rely sets, Guarantee sets, and, stability checks, are specified using separation logic assertions. With respect to the use of auxiliary variables the logics presented in [VP07] and [FFS07] are only slightly better than CSL. As these logics do not describe the shared state using invariants, but instead describe the shared state in the pre- and post-conditions, subject to stability conditions, the programmer needs auxiliary variables only to limit the effect of the environment beyond what can be described by a reflexive and transitively closed binary relation on heaps. For instance, if the programmer assumes that the shared variable x is greater than 0 and he increments x in an environment that also increments x, then he can conclude that x is greater than 1. This can be shown without the use of auxiliary variables as the assertions $x \geq 0$ and $x \geq 1$ are stable under the reflexive and transitive closure of $\{(i, i+1) \mid i \in \mathbb{Z}\}$. But if the programmer want to show that the effect of the program and its environment is to add two to the value of x, then he must limit the specification of the environment to a single transition along the relation $\{(i, i+1) \mid i \in \mathbb{Z}\}$. For this he needs auxiliary variables. To minimize the need for auxiliary variables Dodds et. al. introduced Deny-Guarantee Reasoning [DFPV09], a version of RGSep [VP07] where features of a protocol are baked into the language for specifying rely and guarantee relations. The authors notice that the interference caused by a command may change as the command executes, and thus the rely- and guarantee-relations need to develop as the command executes. They blame this on the dynamically-scoped threads in their language, but I believe this problem is present in any concurrent language where the programmer can synchronize threads executing in parallel. The Deny-Guarantee logic still needs auxiliary variables to reason about the program that increments x twice in parallel, and thus the problems with modularity caused by the use of auxiliary variables is still present.

To obtain better modularity of Rely-Guarantee logics Wickerson et. al. [WDP10] extended RGSep with explicit rely-sets where assertions are either weakened or strengthened to become stable with respect to some rely-set. Modularity is then obtained by quantifying over the explicit rely-sets using higher-order separation logic [BBTS07] in much the same way that Jacobs and Piessens quantified over resource invariants in [JP11].

Another descendant of RGSep is Concurrent Abstract Predicates [DYDG^{+}10], where each shared resource is annotated with a set of actions that a process may perform on the shared state associated with the resource. The actions a process can perform on the state owned by a shared resource is further restricted by labels being passed around like heap cells. A process can only perform an action A if it has access to the label representing A. This enables the programmer to specify complicated protocols without the use of auxiliary variables. For instance, in [DYDG^{+}10] there is an example where a client of a module for a shared lock can only perform an unlock action after it has performed a lock action or just after the lock has been created.

6 Plan to complete thesis

I intend to complete my thesis in a year and a half.

Programming Language: I intend to use much of the semantics developed by Brookes for his soundness proof of CSL [Bro04, Bro07, Bro11]. This semantics is resource sensitive and in my initial work I discovered that the resource sensitivity causes a problem in the proof of associativity of parallel composition. I will need to either address this issue or use a slightly different semantics.

I will also need to design a semantics for interference commands. I intend to use a trace semantics with actions for acquiring and releasing resources, and performing atomic repartitionings. The effect of a command on the shared state while possessing a resource will be summarized as a local action from a special class of local actions that ensure unique splits of heaps when transferring ownership.

I believe I can work out the required semantics in less than five months.

Logic: All the concepts used in the logic and most of the rules have been presented in this proposal, but the semantics of the judgments, i.e., validity of the judgments and necessary enabling relations, needs to be formally defined. I intend to use a local enabling relation in the style of Brookes [Bro04] to define validity of the judgment $L \vdash \{P\}\ c\ \{Q\}$, and thus I must show an agreement theorem between the local enabling relation and the global enabling relation used to define the execution of a command in isolation. Likewise I intend to give a local enabling relation to define validity of the judgment $G \Vdash \{\lambda n.\ P\}\ L\ \{\lambda n'.Q\}$. The meaning of interference commands uses a special class of local actions that ensures unique splits of heaps. Again, I will intend to formally define this class. Finally I must show soundness of the rules, i.e., that each inference rule preserve validity. I expect to spend about five months on this.

Examples: I intend to show usability of the proposed logic by giving valid specifications of interesting programs. I have already worked out a proof sketch of a specification of Petersen's mutual exclusion algorithm and I intend to find more interesting concurrent algorithms. This could include concurrent updates to B-trees or concurrent garbage collection. In this phase I also intend to find a way to present proofs in a form more readable than trees. I intend to develop a notion of annotated programs which correspond to proof trees. This is supposed to be similar to what Reynolds has proposed for separation logic [Rey08].

I intend to spend about four months on finding examples, giving specifications to them, and finding a nice way to present the proofs.

Limitations and future work: I intend to investigate the limitations and the possibilities opened by this framework. I intend to compare proofs in Concurrent Separation Logic and Rely-Guarantee to proofs in this framework. I suspect that every specification and proof in CSL correspond directly to a specification and proof in this framework. Likewise I suspect that every specification and proof in RGSep correspond to a specification and proof in this framework, but this correspondence will be more complicated than the correspondence to CSL.

In the future I would like to see what benefits the framework can bring to specifications of modules, i.e., lists of procedures with access to resources and heap cells shared exclusively between the procedures of a fixed collection. And I would like to investigate the connection with total correctness. I suspect that a notion of *fairness* expressed directly on the graph-like structures of a protocol can be used to express total correctness of concurrent programs under the assumption of a *fair* scheduler.

I intend to spend two to three month on these investigations.

Finishing up thesis, tying loose ends: Finally, I intend to spend less than two months tying up any loose ends before submitting my thesis.

References

[BBTS07] Bodil Biering, Lars Birkedal, and Noah Torp-Smith, *BI-hyperdoctrines, higher-order separation logic, and abstraction*, ACM Trans. Program. Lang. Syst. **29** (2007).

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson, *Permission accounting in separation logic*, Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (Jens Palsberg and Martín Abadi, eds.), POPL, ACM, 2005, pp. 259–270.

[BCY06] Richard Bornat, Cristiano Calcagno, and Hongseok Yang, *Variables as Resource in Separation Logic*, Electr. Notes Theor. Comput. Sci. **155** (2006), 247–276.

[Boy03] John Boyland, *Checking interference with fractional permissions*, SAS (Radhia Cousot, ed.), Lecture Notes in Computer Science, vol. 2694, Springer, 2003, pp. 55–72.

[Bro04] Stephen Brookes, *A Semantics for Concurrent Separation Logic*, CONCUR (Philippa Gardner and Nobuko Yoshida, eds.), Lecture Notes in Computer Science, vol. 3170, Springer, 2004, pp. 16–34.

[Bro07] Stephen Brookes, *A semantics for concurrent separation logic*, Theor. Comput. Sci. **375** (2007), no. 1-3, 227–270.

[Bro11] ______, *A Revisionist History of Concurrent Separation Logic*, Electronic Notes in Theoretical Computer Science (2011), Proceedings of the 27th Conference on the Mathematical Foundations of Programming Semantics (MFPS).

[BS11] Thomas Ball and Mooly Sagiv (eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 2011.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang, *Local action and abstract separation logic*, LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (Washington, DC, USA), IEEE Computer Society, 2007, pp. 366–378.

[DFPV09] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis, *Deny-Guarantee Reasoning*, ESOP (Giuseppe Castagna, ed.), Lecture Notes in Computer Science, vol. 5502, Springer, 2009, pp. 363–377.

[DYDG$^+$10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis, *Concurrent Abstract Predicates*, ECOOP (Theo D'Hondt, ed.), Lecture Notes in Computer Science, vol. 6183, Springer, 2010, pp. 504–528.

[Fen09] Xinyu Feng, *Local rely-guarantee reasoning*, POPL (Zhong Shao and Benjamin C. Pierce, eds.), ACM, 2009, pp. 315–327.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao, *On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning*, ESOP (Rocco De Nicola, ed.), Lecture Notes in Computer Science, vol. 4421, Springer, 2007, pp. 173–188.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn, *BI as an assertion language for mutable data structures*, Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL, ACM, 2001, pp. 14–26.

[Jon81] C. B. Jones, *Development methods for computer programs including a notion of interference*, Ph.D. thesis, Oxford University, June 1981, Printed as: Programming Research Group, Technical Monograph 25.

[Jon83a] ______, *Specification and design of (parallel) programs*, Proceedings of IFIP'83, North-Holland, 1983, pp. 321–332.

[Jon83b] ______, *Tentative steps toward a development method for interfering programs*, Transactions on Programming Languages and System **5** (1983), no. 4, 596–619.

[JP11] Bart Jacobs and Frank Piessens, *Expressive modular fine-grained concurrency specification*, in Ball and Sagiv [BS11], pp. 271–282.

[KAB$^+$09] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse, *Design patterns in separation logic*, Proceedings of the 4th international workshop on Types in language design and implementation (New York, NY, USA), TLDI '09, ACM, 2009, pp. 105–116.

[O'H07] Peter W. O'Hearn, *Resources, concurrency, and local reasoning*, Theor. Comput. Sci. **375** (2007), no. 1-3, 271–307.

[Owi75] Susan S. Owicki, *Axiomatic Proof Techniques for Parallel Programs*, Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York, 1975.

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds, *Separation and information hiding*, POPL (Neil D. Jones and Xavier Leroy, eds.), ACM, 2004, pp. 268–280.

[PBC06] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno, *Variables as Resource in Hoare Logics*, LICS, IEEE Computer Society, 2006, pp. 137–146.

[Rey02] John C. Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*, LICS, IEEE Computer Society, 2002, pp. 55–74.

[Rey08] ———, *Readable formal proofs*, VSTTE (Natarajan Shankar and Jim Woodcock, eds.), Lecture Notes in Computer Science, vol. 5295, Springer, 2008, p. 1.

[RR12] Uday S. Reddy and John C. Reynolds, *Syntactic Control of Interference for Separation Logic*, to appear at POPL, ACM, 2012.

[VP07] Viktor Vafeiadis and Matthew J. Parkinson, *A Marriage of Rely/Guarantee and Separation Logic*, CONCUR (Luís Caires and Vasco Thudichum Vasconcelos, eds.), Lecture Notes in Computer Science, vol. 4703, Springer, 2007, pp. 256–271.

[WDP10] John Wickerson, Mike Dodds, and Matthew J. Parkinson, *Explicit stabilisation for modular rely-guarantee reasoning*, ESOP (Andrew D. Gordon, ed.), Lecture Notes in Computer Science, vol. 6012, Springer, 2010, pp. 610–629.