Separation and Information Hiding

Peter W. O'Hearn
Queen Mary, University of London

Hongseok Yang
Queen Mary, University of London

John C. Reynolds
Carnegie Mellon University

We investigate proof rules for information hiding, using the formalism of separation logic. In essence, we use the separating conjunction to partition the internal resources of a module from those accessed by the module's clients. The use of a logical connective gives rise to a form of dynamic partitioning, where we track the transfer of ownership of portions of heap storage between program components. It also enables us to enforce separation in the presence of mutable data structures with embedded addresses that may be aliased.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*class invariants*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*modules, packages*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Separation Logic, Modularity, Resource Protection

1. INTRODUCTION

Modularity is a key concept which programmers wield in their struggle against the complexity of software systems. When a program is divided into conceptually distinct modules or components, each of which owns separate internal resources (such as storage), the effort required for understanding the program is decomposed into circumscribed, hopefully manageable, parts. And, if separation is correctly maintained, we can regard the internal resources of one module as hidden from its clients, which results in a narrowing of interface between program components. The flipside, of course, is that an ostensibly modular program organization is undermined when internal resources are accessed from outside a module.

It stands to reason that, when specifying and reasoning about programs, if we can keep track of the separation of resources between program components, then the resultant decomposition of the specification and reasoning tasks should confer similar benefits. Unfortunately, most methods for specifying programs either severely restrict the programming model, by ruling out common programming features (so as to make the static enforcement of separation feasible), or they expose the internal resources of a module in its specification in order to preserve soundness.

Stated more plainly, information hiding should be the bedrock of modular reasoning, but it is difficult to support soundly, and this presents a great challenge for research in program logic.

To see why information hiding in specifications is important, suppose a program makes use of n different modules. It would be unfortunate if we had to thread

descriptions of the internal resources of each module through steps when reasoning about the program. Even worse than the proof burden would be the additional annotation burden, if we had to complicate specifications of user procedures by including descriptions of the internal resources of all modules that might be accessed. A change to a module's internal representation would necessitate altering the specifications of all other procedures that use it. The resulting breakdown of modularity would doom any aspiration to scalable specification and reasoning.

Mutable data structures with embedded addresses (pointers) have proven to be a particularly obstinate obstacle to modularity. The problem is that it is difficult to keep track of aliases, different copies of the same address, and so it is difficult to know when there are no pointers into the internals of a module. The purpose of this paper is to investigate proof rules for information hiding using separation logic, a formalism for reasoning about mutable data structures [Reynolds 2002].

Our treatment draws on work of Hoare on proof rules for data abstraction and for shared-variable concurrency [Hoare 1972a; 1972b; 1974]. In Hoare's approach each distinct module has an associated resource invariant, which describes its internal state, and scoping constraints are used to separate the resources of a module from those of client programs. We retain the resource invariants, and add a logical connective, the separating conjunction $*$, to provide a more flexible form of separation.

We begin in the next section by describing the memory model and the logic of pre- and post-conditions used in this work. We then describe our proof rules for information hiding, followed by two examples, one a simple memory manager module and the other a queue module. Both examples involve the phenomenon of *resource ownership transfer*, where the right to access a data structure transfers between a module and its clients. We work through proofs, and failed proofs, of client code as a way to illustrate the consequences of the proof rules.

After giving the positive examples we present a counterexample, which shows that our principal new proof rule, the hypothetical frame rule, is incompatible with the usual Hoare logic rule for conjunction; the new rule is thus unsound in models where commands denote relations, which validate conjunction. The problem is that the very features that allow us to treat ownership transfer lead to a subtle understanding where "Ownership is in the eye of the Asserter". The remainder of the paper is occupied with a semantic analysis, and that analysis is the principle technical contribution of the paper. A crucial role is played by the identification of the notion of a *precise* predicate, which requires the Asserter to identify a definite portion of storage, unambiguously.

Familiarity with the basics of separation logic, as presented in [Reynolds 2002], would be helpful in reading the paper. We remind the reader in particular that the rules for disposing or dereferencing an address are such that it must be known to point to something (not be dangling) in the precondition for a rule to apply. For example, in the putative triple $\{\mathtt{true}\}[x] := 7\{???\}$, where the contents of heap address x is mutated to 7, there is no assertion we can use in the postcondition to get a valid triple, because x might be dangling in a state satisfying the precondition. So, in order to obtain any postcondition for $[x] := 7$, the precondition must imply the assertion $x \mapsto - * \mathtt{true}$ that x is not dangling.

The local way of thinking encouraged by separation logic [O'Hearn et al. 2001] is stretched by the approach to information hiding described here. We have found it useful to use a figurative language of "rights" when thinking about specifications, where a predicate p at a program point asserts that "I have the right to dereference the addresses in p here".

1.1 Contextual Remarks

The link between modularity and information hiding was developed in papers of Hoare and Parnas in the early 1970s [Hoare 1972a; 1972b; Parnas 1972a; 1972b]. Parnas emphasized that poor information distribution amongst components could lead to "almost invisible connections between supposedly independent modules", and proposed that information hiding was a way to combat this problem. Hoare suggested using scoping restrictions to hide a particular kind of information, the internal state of a module, and showed how these restrictions could be used in concert with invariants to support proof rules that did not need to reveal the internal data of a module or component. These ideas influenced many subsequent language constructs and specification notations.

Most formal approaches to information hiding work by assuming a fixed, a priori, partitioning between program components, usually expressed using scoping restrictions, or typing, or simply using cartesian product of state spaces. In simple cases fixed partitioning can be used to protect internal resources from outside tampering. But in less simple situations, such as when data is referred to indirectly via addresses, or when resources dynamically transfer between program components, correct separation is more difficult to maintain. Such situations are especially common in low-level systems programs whose purpose is to provide flexible, shared access to system resources. They are also common in object-oriented programs. An unhappy consequence is that modular specification methods are lacking for widely-used imperative or object-oriented programming languages, or even for many of the programming patterns commonly expressed in them.

The essential point is that fixed partitioning does not cope naturally with systems whose resource ownership or interconnection structure is changing over time. A good example is a resource management module, that provides primitives for allocating and deallocating resources, which are held in a local free list. A client program should not alter the free list, except through the provided primitives; for example, the client should not tie a cycle in the free list. In short, the free list is owned by the manager, and it is (intuitively) hidden from client programs. However, it is entirely possible for a client program to hold an alias to an element of the free list, after a deallocation operation is performed; intuitively, the "ownership" of a resource transfers from client to module on disposal, even if many aliases to the resource continue to be held by the client code. In a language that supports address arithmetic the potential difficulties are compounded: the client might intentionally or unintentionally obtain an address used in an internal representation, just by an arithmetic calculation.

A word of warning on our use of "module" before we continue: The concept of module we use is just a grouping of procedures that share some private state. The sense of "private" will not be determined statically, but will be the subject of specifications and proof rules. This allows us to approach modules where correct

protection of module internals would be impossible to determine with a compile-time check in current programming languages. The approach in this paper might conceivably be used to analyze the information hiding in a language that provides an explicit module notation, but that is not our purpose here.

The point is that it is possible to program modules, in the sense of the word used by Parnas, whether or not one has a specific module construct at one's disposal. For example, the pair of `malloc()` and `free`() in C, together with their shared free list, might be considered as a module, even though their correct usage is not guaranteed by C's compile-time checking. Indeed, there is no existing programming language that correctly enforces information hiding of mutable data structures, largely because of the dynamic partitioning issue mentioned above, and this is an area where logical specifications are needed. We emphasize that the issue is not one of "safe" versus "unsafe" programming languages; for instance, middleware programs written in garbage-collected, safe languages, often perform explicit management of certain resources, and there also ownership transfer is essential to information hiding.

Similarly, although we do not consider the features of a full-blown object-oriented language, our techniques, and certainly our problems, seem to be relevant. Theories of objects have been developed that account for hiding in a purely functional context (e.g., [Pierce and Turner 1994]), but mutable structures with embedded addresses, or object id's, are fundamental to object-oriented programming. A thoroughgoing theory should account for them directly, confronting the problems caused when there are potential aliases to the state used within an object.

These contextual remarks do not take into account some recent work that attempts to address the limitations of fixed partitioning and the difficulties of treating mutable data structures with embedded addresses, including work that followed on from the preliminary version of this paper published in the POPL'04 conference proceedings [O'Hearn et al. 2004]. We will say more on some of the closely related work at the end of the paper.

[Note to referees. Compared to the conference version from POPL'04, this paper contains a much more thorough semantic analysis. In fact, the difficulty of the semantic problem was almost completely hidden in POPL'04 due to space limitations, and that is where a significant part of the technical contribution of this paper lies.]

2. THE STORAGE MODEL

We consider a model where a heap is a finite partial function taking addresses to values:

$$H \stackrel{def}{=} \mathsf{Addresses} \rightharpoonup_{\mathrm{fin}} \mathsf{Values}$$

This set has a partial commutative monoid structure, where the unit is the empty function and the partial combining operation

$$* : H \times H \rightharpoonup H$$

is the union of partial functions with disjoint domains. More formally, we say that $h_1 \# h_2$ holds for heaps h_1 and h_2 when $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \{\}$. In that case, $h_1 * h_2$ denotes the combined heap $h_1 \cup h_2$. When $h_1 \# h_2$ fails, $h_1 * h_2$ is undefined.

In particular, note that if $h = h_1 * h_2$ then we must have that $h_1 \# h_2$. The subheap order $\leq$ is subset inclusion of partial functions.

We will work with a RAM model, where the addresses are natural numbers and the values are integers

$$\mathsf{Addresses} \stackrel{def}{=} \{0, 1, 2, \ldots\} \qquad \mathsf{Values} \stackrel{def}{=} \{\ldots, -1, 0, 1, \ldots\}$$

The results of this paper go through for other choices for Addresses and Values, and thus cover a number of other naturally occurring models, such as the cons cell model of [Reynolds 2002] and the hierarchical memory model of [Ahmed et al. 2003]. Our results also apply to traditional Hoare logic, where there is no heap, by taking the trivial model where Addresses is empty (and Values non-empty).

To interpret variables in the programming language and logic, the state has an additional component, the "stack", which is a mapping from variables to values; a state is then a pair consisting of a stack and a heap:

$$S \stackrel{def}{=} \mathsf{Variables} \to \mathsf{Values} \qquad \mathsf{States} \stackrel{def}{=} S \times H.$$

We treat predicates semantically in this paper, so a predicate is just a set of states.

$$\mathsf{Predicates} \stackrel{def}{=} \mathcal{P}(\mathsf{States})$$

The powerset of states has the usual boolean algebra structure, where $\wedge$ is intersection, $\vee$ is union, $\neg$ is complement, `true` is the set of all states, and `false` is the empty set of states. We use p, q, r, sometimes with subscripts and superscripts, to range over predicates. Besides the boolean connectives, we will need the lifting of $*$ from heaps to predicates:

$$p * q \stackrel{def}{=} \{(s,h) \mid \exists h_0, h_1.\, h = h_0 * h_1, \text{ and } (s, h_0) \in p, \text{ and } (s, h_1) \in q\}.$$

As a function on predicates we have a total map $*$ from Predicates $\times$ Predicates to Predicates which, to the right of $\stackrel{def}{=}$, uses the partial map, $* : H \times H \rightharpoonup H$ in its definition. This overloading of $*$ will always be disambiguated by context. $*$ has a unit `emp`, the set $\{(s, []) \mid s \in S\}$ of states whose heap component is empty. It also has an implication adjoint $-\!\!*$, though that will play no role in the present paper. Note that `emp` is distinct from the empty set `false` of states.

We use $x \mapsto E$ to denote a predicate that consists of all pairs (s, h) where h is a singleton in which x points to the meaning of E: $h(s(x)) = [\![E]\!]s$. The points-to relation $x \mapsto E, F$ for binary cons cells is syntactic sugar for $(x \mapsto E) * (x + 1 \mapsto F)$. We will also use quantifiers and recursive definitions in examples in what should be a clear way.

The syntax for the programming language considered in this paper is given by the following grammar.

$$\begin{aligned}
E &::= x, y, \ldots \mid 0 \mid 1 \mid E + E \mid E \times E \mid E - E \\
B &::= \texttt{false} \mid B \Rightarrow B \mid E = E \mid E < E \\
C &::= x := E \mid x := [E] \mid [E] := E \mid x := \texttt{cons}(E, \ldots, E) \\
&\quad \mid \texttt{dispose}(E) \mid \texttt{skip} \mid C; C \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \\
&\quad \mid \texttt{while } B\ C \mid \texttt{letrec } k = C, \ldots, k = C \texttt{ in } C \mid k
\end{aligned}$$

Generic Syntax

$$
\begin{array}{lll}
C & ::= & BC \mid \texttt{skip} \mid C_1; C_2 \\
 & \mid & \texttt{while}\ B\ C \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{letrec}\ k_1 = C_1, ..., k_n = C_n\ \texttt{in}\ C \mid k
\end{array}
$$

RAM-specific Syntax

$$
\begin{array}{lll}
BC & ::= & x := E \mid x := [E] \mid [E] := E_1 \mid x := \texttt{cons}(E_1, \ldots, E_n) \mid \texttt{dispose}(E) \\
E & ::= & x, y, \ldots \mid 0 \mid 1 \mid E_1 + E_2 \mid E_1 \times E_2 \mid E_1 - E_2 \\
B & ::= & \texttt{false} \mid B_1 \Rightarrow B_2 \mid E_1 = E_2 \mid E_1 < E_2
\end{array}
$$

RAM-specific Syntactic Sugar

$$
\begin{array}{lll}
x.i := E & \stackrel{def}{=} & [x + i - 1] := E \\
x := E.i & \stackrel{def}{=} & x := [E + i - 1]
\end{array}
$$

Table I. Programming Language Syntax

For simplicity we consider parameterless procedures only. The extension to all first-order procedures raises no new difficulties, but lengthens the presentation. Higher-order features, on the other hand, are not straightforward. We assume that all the procedure identifiers are distinct in any `letrec` declaration. When procedure declarations do not have recursive calls, we write $\texttt{let}\ k_1 = C_1, ..., k_n = C_n\ \texttt{in}\ C$ to indicate this.

The command $x := \texttt{cons}(E_1, \ldots, E_n)$ allocates n consecutive cells, initializes them with the values of $E_1, \ldots, E_n$, and stores the address of the first cell in x. We could also consider a command for variable-length allocation. The contents of an address E can be read and stored in x by $x := [E]$, or can be modified by $[E] := F$. The command $\texttt{dispose}(E)$ deallocates the address E. In $x := [E]$, $[E] := F$ and $\texttt{dispose}(E)$, the expression E can be an arbitrary arithmetic expression; so, this language allows address arithmetic.

This inclusion of address arithmetic does not represent a general commitment to it on our part, but rather underlines the point that our methods do not rely on ruling it out. In examples it is often clearer to use a field-selection notation rather than arithmetic, and for this we use the following syntactic sugar:

$$E.i := F \stackrel{def}{=} [E + i - 1] := F \quad x := E.i \stackrel{def}{=} x := [E + i - 1].$$

Each command denotes a (nondeterministic) state transformer that faults when heap storage is accessed illegally, and each expression determines a (heap independent) function from stacks to values. The semantics will be given in Section 7.

3. PROOF SYSTEM

The form of judgment we use is the sequent

$$\Gamma \vdash \{p\}C\{q\}$$

which states that command C satisfies its Hoare triple, under certain hypotheses. Hypotheses are given by the grammar

$$\Gamma ::= [] \mid \{p\}k\{q\}[X], \Gamma$$

subject to the constraint that no procedure identifier appears twice. An assumption $\{p\}k\{q\}[X]$ requires the parameterless procedure identifier k to denote a command which modifies only the variables appearing in set X and which satisfies the indicated Hoare triple.

The commands are drawn from a language of while programs with parameterless procedures (Table I). The generic syntax includes procedure calls for parameterless procedures k and a non-terminal BC for basic commands, which can be instantiated in various ways depending on the storage model being used. We give one such instantiation, corresponding to the RAM model.

3.1 Proof Rules for Information Hiding

Our main focus in this paper is the

Hypothetical Frame Rule

$$\frac{\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\Gamma, \{p_1 * r\}k_1\{q_1 * r\}[X_1, Y], \ldots, \{p_n * r\}k_n\{q_n * r\}[X_n, Y] \vdash \{p * r\}C\{q * r\}}$$

where

—C does not modify variables in r, *except through using* $k_1, ..., k_n$; and
—Y is disjoint from judgment "$\Gamma, \{p_1\}k\{q_1\}[X_1], ..., \{p_n\}k\{q_n\}[X_n] \vdash \{p\}C\{q\}$".

The idea behind these conditions is that we must be sure that client code does not alter variables used within a module, but we must also allow some overlap in variables to treat various examples. The side conditions use notions which are as-yet-undefined, in particular the "except through using" clause. The conditions will be made rigorous in Section 10; in the examples in the following sections we will concentrate on the role of $*$, and there will be no harm to understanding if the variable conditions are skated over, or referred back to as necessary.

The hypothetical frame rule is so named because of its relation to the ordinary frame rule from [Isthiaq and O'Hearn 2001; O'Hearn et al. 2001]. The hypothetical rule allows us to place invariants on the hypotheses as well as the conclusion of sequents, whereas the ordinary rule includes invariants on the conclusion alone. (The ordinary frame rule is thus a special case of the hypothetical rule, where $n = 0$.)

To explain the rule intuitively, suppose we have a module, which exports a number of procedures k_i. Here we mean "module" in an informal sense, a grouping of procedures that implements an abstraction, and not necessarily an explicit programming-language construct. There are two views of the module. From the outside, where the internal resource is invisible, one uses interface specifications $\{p_i\}k_i\{q_i\}$ of the procedures, which do not mention the resource invariant r that describes the internal state of the module. The perspective is different from inside the module; the operations operate on a larger state than that visible to the client, preserving the invariant as well as satisfying interface specifications. The hypothetical frame rule ties these two viewpoints together.

The hypothetical frame rule is logician-friendly, suitable for theoretical analysis, but more programmer-friendly, derived rules are useful when reasoning about programs. For example, we can formulate a proof rule for procedure declarations, which perhaps more directly portrays the division between the two sides of a module.

Modular Non-Recursive Procedure Declaration Rule

$$\frac{\begin{array}{c}\Gamma \vdash \{p_1 * r\}C_1\{q_1 * r\} \\ \vdots \\ \Gamma \vdash \{p_n * r\}C_n\{q_n * r\} \\ \Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}\end{array}}{\Gamma \vdash \{p * r\}\texttt{let } k_1 = C_1, \ldots, k_n = C_n \texttt{ in } C\{q * r\}}$$

In this rule $k_1, \ldots k_n$ is a grouping of procedures that share private state described by resource invariant r. In a resource management module, the k_i would be operations for allocating and freeing resources, and r would describe unallocated resources (perhaps held in a free list). The rule distinguishes two views of such a module. When reasoning about the client code C, we ignore the invariant and its area of storage; reasoning is done in the context of *interface specifications* $\{p_i\}k_i\{q_i\}$ that do not mention r. The perspective is different from inside the module; the implementations C_i operate on a larger state than that presented to the client, and verifications are performed in the presence of the resource invariant. The two views, module and client, are tied up in the conclusion of the rule.

The modular procedure rule is subject to variable conditions: we require a set Y (of "private" variables), and the conditions are

—C does not modify variables in r, *except through using* $k_1, ..., k_n$;
—Y is disjoint from judgment "$\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}$";
—C_i only modifies variables in X_i, Y.

An important point is that the free variables of the resource invariant are allowed to overlap with the X_i. This often happens when using auxiliary variables to specify the behaviour of a module, as exemplified by the treatment of the abstract variable Q in the queue module given later (Table IV).

It is also possible to consider initialization and finalization code. For instance, if, in addition to the premises of the modular procedure rule, we have $\Gamma \vdash \{p\}\text{init}\{p{*}r\}$ and $\Gamma \vdash \{q * r\}\text{final}\{q\}$, then we can obtain

$$\Gamma \vdash \{p\}\ \text{init}; (\texttt{let } k_1 = C_1, \ldots, k_n = C_n \texttt{ in } C); \text{ final } \{q\}.$$

In our examples we will not consider initialization or finalization since they present no special logical difficulties.

In the modular procedure rule, the proof of $\{p\}C\{q\}$ about the client in the premises can be used with *any* resource invariant r. As a result, this reasoning does not need to be repeated when a module representation is altered, as long as the alteration continues to satisfy the interface specifications $\{p_i\}k_i\{q_i\}$. This addresses one of the points about reasoning that survives local changes discussed in the Introduction.

Interface Specifications $\{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n]$
Resource Invariant: r
Private Variables: Y
Internal Implementations $C_1, ..., C_n$

Table II. Module Specification Format

However, the choice of invariant r is not specified by programming language syntax $\texttt{let}\ k_1 = C_1, \ldots, k_n = C_n\ \texttt{in}\ C$ in the modular procedure rule. In this it is similar to the usual partial correctness rule for `while` loops, which depends on the choice of a loop invariant. It will be convenient to consider an annotation notation that specifies the invariant, and the interface specifications $\{p_i\}k_i\{q_i\}$, as a directive on how to apply the modular procedure rule; this is by analogy with the use of loop invariant annotations as directives to a verification condition generator.

We will use the format for module specifications in Table II. This instructs us to apply the modular procedure rule in a particular way, to prove

$$\Gamma, \mathsf{Interface\ Specifications} \vdash \{p\}C\{q\}$$

for client code C, and to prove $\Gamma \vdash \{p_i * r\}C_i\{q_i * r\}$ for the bodies. We emphasize that this module format is not officially part of our programming language or even our logic; however, its role as a directive on how to apply the modular procedure rule in examples will, we hope, be clear.

3.2 Other Proof Rules

3.2.1 *Generic Rules.* We have standard Hoare logic rules for various constructs, along with the rule of consequence and the rules for shrinking and extending contexts.

$$\frac{}{\Gamma, \{p\}k\{q\}[X] \vdash \{p\}k\{q\}} \qquad \frac{p \Rightarrow p' \quad \Gamma \vdash \{p'\}C\{q'\} \quad q' \Rightarrow q}{\Gamma \vdash \{p\}C\{q\}}$$

$$\frac{\Gamma \vdash \{p \wedge B\}C\{p\}}{\Gamma \vdash \{p\}\texttt{while}\, B\, C\{p \wedge \neg B\}} \qquad \frac{\Gamma \vdash \{p\}C_1\{q\} \quad \Gamma \vdash \{q\}C_2\{r\}}{\Gamma \vdash \{p\}C_1; C_2\{r\}}$$

$$\frac{\Gamma \vdash \{p \wedge B\}\, C\, \{q\} \quad \Gamma \vdash \{p \wedge \neg B\}\, C'\, \{q\}}{\Gamma \vdash \{p\}\, \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C'\{q\}}$$

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma, \{p'\}k\{q'\}[X] \vdash \{p\}C\{q\}} \qquad \frac{\Gamma, \{p'\}k\{q'\}[X] \vdash \{p\}C\{q\}}{\Gamma \vdash \{p\}C\{q\}}\ (k \text{ does not occur in } C)$$

In addition, we allow for the context Γ to be permuted.

The standard rule for possibly recursive procedure declarations, which *doesn't* hide a resource invariant, uses the procedure specifications in proofs of the bodies as follows:

$$\frac{\begin{array}{c}\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p_1\}C_1\{q_1\} \\ \vdots \\ \Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p_n\}C_n\{q_n\} \\ \Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}\end{array}}{\Gamma \vdash \{p\}\texttt{letrec}\ k_1 = C_1, \ldots, k_n = C_n\ \texttt{in}\ C\{q\}}$$

where C_i only modifies variables in X_i.

In case none of the k_i are free in the C_j we can get a simpler rule, where the $\{p_i\}k_i\{q_i\}[X_i]$ hypotheses are omitted from the sequents for the C_j. Using `let` rather than `letrec` to indicate the case where a procedure declaration happens to have no recursive instances, we can derive the modular non-recursive procedure declaration rule of the previous section from the hypothetical frame rule and the standard procedure rule just given. We can also derive a modular rule for recursive declarations.

The ordinary frame rule is

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma \vdash \{p * r\}C\{q * r\}}$$

where C does not modify any variables free in r.

This is a special case of the hypothetical rule, but we state it separately because the ordinary rule will be used without restriction, while we will place restrictions on the hypothetical rule.

One rule of Hoare logic, which is sometimes not included explicitly in proof systems, is the conjunction rule.

$$\frac{\Gamma \vdash \{p\}C\{q\} \quad \Gamma \vdash \{p'\}C\{q'\}}{\Gamma \vdash \{p \wedge p'\}C\{q \wedge q'\}}$$

The conjunction rule is often excluded because it is an example of an *admissible* rule: one can (usually) prove a metatheorem, which says that if the premises are derivable then so is the conclusion. However, it is not an example of a *derived* rule: one cannot construct a generic derivation, in the logic, of the conclusion from the premises. We will see in Section 6 that the hypothetical frame rule can affect the admissible status of the conjunction rule.

3.2.2 *RAM-specific Axioms.* Specific axioms are needed for any collection BC of basic commands. Here are the "small axioms" appropriate to the RAM model,

Interface Specifications

$\{\texttt{emp}\}\texttt{alloc}\{x \mapsto -, -\}\,[x]$
$\{x \mapsto -, -\}\texttt{free}\{\texttt{emp}\}\,[]$

Resource Invariant: $\mathit{list}(f)$

Private Variables: f

Internal Implementations

$\texttt{if } (f = \texttt{nil}) \texttt{ then } x := \texttt{cons}(-, -)$	(code for `alloc`)
$\texttt{else } x := f;\ f := x.2;$	
$x.2 := f;\ f := x;$	(code for `free`)

Table III. Memory Manager Module

where x, m, n are assumed to be distinct variables.

$$\Gamma \vdash \{E \mapsto -\}\,[E] := E_1\,\{E \mapsto E_1\}$$

$$\Gamma \vdash \{E \mapsto -\}\,\texttt{dispose}(E)\,\{\texttt{emp}\}$$

$$\Gamma \vdash \{x = m \wedge \texttt{emp}\}x := \texttt{cons}(E_1, ..., E_k)\Gamma \vdash \{x \mapsto E_1[m/x], ..., E_k[m/x]\,\}$$

$$\Gamma \vdash \{x = n \wedge \texttt{emp}\}\,x := E\,\{x = (E[n/x]) \wedge \texttt{emp}\}$$

$$\Gamma \vdash \{E \mapsto n \ \wedge\ x = m\}\,x := [E]\,\{x = n \ \wedge\ E[m/x] \mapsto n\}$$

These small axioms describe the effect of each command on only one, or sometimes no, heap cells. Typically, these effects can be extended using the frame rule: for example, we can infer $\{(x \mapsto 3) * (y \mapsto 4)\}[x] := 7\{(x \mapsto 7) * (y \mapsto 4)\}$ by choosing $y \mapsto 4$ as the invariant in the frame rule.

4. A MEMORY MANAGER

We consider an extended example, of an idealized memory manager that doles out memory in chunks of size two. The specifications and code are given in Table III.

The internal representation of the manager maintains a free list, which is a singly-linked list of binary cons cells. The free list is pointed to by f, and the predicate $\mathit{list}(f)$ is the representation invariant, where

$$\mathit{list}(f) \stackrel{def}{\Longleftrightarrow} (f = \texttt{nil} \ \wedge\ \texttt{emp}) \ \vee\ (\exists g.\, f \mapsto -, g \ *\ \mathit{list}(g))$$

This predicate says that f points to a linked list (and that there are no other cells in storage), but it does not say what elements are in the head components.

For the implementation of `alloc`, the manager places into x the address of the first element of the free list, if the list is nonempty. In case the list is empty the manager calls the built-in allocator `cons` to get an extra element. The interaction between `alloc` and `cons` is a microscopic idealization of the treatment of `malloc` in Section 8.7 of [Kernighan and Ritchie 1988]. There, `malloc` manages a free list but, occasionally, it calls a system routine `sbrk` to request additional memory. Besides

fixed versus variable sized allocation, the main difference is that we assume that `cons` always succeeds, while `sbrk` might fail (return an error code) if there is no extra memory to be given to `malloc`. We use this simple manager because to use a more complex one would not add anything to the points made in this section.

When a user program gives a cell back to the memory manager it is put on the front of the free list; there is no need for interaction with a system routine here.

The form of the interface specifications are examples of the local way of thinking encouraged by separation logic; they refer to small pieces of storage. It is important to appreciate the interaction between local and more global perspectives in these assertions. For example, in the implementation of `free` in Table III the variable x contains the same address after the operation completes as it did before, and the address continues to be in the domain of the global program heap. The use of `emp` in the postcondition of `free` does not mean that the global heap is now empty, but rather it implies that the knowledge that x points to something is given up in the postcondition. We say intuitively that `free` transfers ownership to the manager, where ownership confers the right to dereference.

It is interesting to see how transfer works logically, by considering a proof outline for the implementation of `free`.

$\{\mathit{list}(f) * (x \mapsto -, -)\}$
$x.2 := f;$
$\{\mathit{list}(f) * (x \mapsto -, f)\}$
$\{\mathit{list}(x)\}$
$f := x;$
$\{\mathit{list}(f)\}$
$\{\mathit{list}(f) * \mathtt{emp}\}$

The most important step is the middle application of the rule of consequence. At that point we still have the original resource invariant $\mathit{list}(f)$ and the knowledge that x points to something, separately. But since the second field of what x points to holds f, we can obtain $\mathit{list}(x)$ as a consequence. It is at this point in the proof that the original free list and the additional element x are bundled together; the final statement simply lets f refer to this bundled information.

A similar point can be made about how `alloc` effects a transfer from the module to the client.

We now give several examples from the client perspective. Each proof, or attempted proof, is done in the context of the interface specifications of `alloc` and `free`.

The first example is for inserting an element into the middle of a linked list.

$\{(y \mapsto a, z) * (z \mapsto c, w)\}$
`alloc`;
$\{(y \mapsto a, z) * (z \mapsto c, w) * (x \mapsto -, -)\}$
$\{(y \mapsto a, z) * (x \mapsto -, -) * (z \mapsto c, w)\}$
$x.2 := z;\ x.1 := b; y.2 := x$
$\{(y \mapsto a, x) * (x \mapsto b, z) * (z \mapsto c, w)\}$

Here, in the step for `alloc` we use the interface specification, together with the ordinary frame rule.

If we did not have the modular procedure rule we could still verify this code, by threading the free list through and changing the interface specification. That is, the interface specifications would become

$\{list(f)\}\texttt{alloc}\{list(f) * x \mapsto -, -\}$
$\{list(f) * x \mapsto -\}\texttt{free}\{list(f)\}$

thus exposing the free list, and the proof would be

$\{(y \mapsto a, z) * (z \mapsto c, w) * list(f)\}$
$\texttt{alloc};$
$\{(y \mapsto a, z) * (z \mapsto c, w) * (x \mapsto -, -) * list(f)\}$
$\{(y \mapsto a, z) * (x \mapsto -, -) * (z \mapsto c, w) * list(f)\}$
$x.2 := z;\ x.1 := b; y.2 := x$
$\{(y \mapsto a, x) * (x \mapsto b, z) * (z \mapsto c, w) * list(f)\}.$

Although technically correct, this inclusion of the free list in the proof of the client is an example of the breakdown of modularity described in the Introduction.

One might wonder whether this hiding of invariants could be viewed as a simple matter of syntactic sugar, instead of being the subject of a proof rule. We return to this point in Section 6.

We can similarly reason about deletion from the middle of a linked list, but it is more interesting to attempt to delete wrongly.

$\{(y \mapsto a, x) * (x \mapsto b, z) * (z \mapsto c, w)\}$
$\texttt{free};$
$\{(y \mapsto a, x) * (z \mapsto c, w)\}$
$y := x.2;$
$\{???\}$

This verification cannot be completed, because after doing the `free` operation the client has given up the right to dereference x.

This is a very simple example of the relation between ownership transfer and aliasing; after the `free` operation x and f are aliases in the global state, and the incorrect use of the alias by the client has been rightly precluded by the proof rules.

Similarly, suppose the client tried to corrupt the manager, by sneakily tying a cycle in the free list.

$\{\texttt{emp}\}\ \texttt{alloc; free; x.2} := \texttt{x}\ \{???\}$

Once again, there is no assertion we can find to fill in the ???, because after the `free` statement the client has given up the right to dereference x (`emp` will hold at this program point). And, this protection has nothing to do with the fact that knotting the free list contradicts the resource invariant. For, suppose the statement $x.2 := x$ was replaced by $x.1 := x$. Then the final assignment in this sequence would not contradict the resource invariant, when viewed from the perspective of the system's global state, because the $list(f)$ predicate is relaxed about what values are in head components. However, from the point of view of the interface specifications, the client has given up the right to dereference even the first component of x. Thus, separation prevents the client from accessing the internal storage of the module in any way whatsoever.

Finally, it is worth emphasizing that this use of $*$ to enforce separation provides protection even in the presence of address arithmetic which, if used wrongly, can wreak havoc with data abstractions. Suppose the client tries to access some memory address, which might or might not be in the free list, using $[42] := 7$. Then, for this statement to get past the proof rules, the client must have the right to dereference 42, and therefore 42 cannot be in the free list (by separation). That is, we have two cases

$$\{42 \mapsto - * p\}\, [42] := 7;\, \texttt{alloc}\, \{42 \mapsto 7 * p * x \mapsto -, -\}$$

and

$$\{p\}\, [42] := 7;\, \{???\}\, \texttt{alloc}\, \{???\}$$

where p does not imply that 42 is in the domain of its heap. In the first case the client has used address arithmetic correctly, and the $42 \mapsto -$ in the precondition ensures that 42 is not one of the cells in the free list. In the second case the client uses address arithmetic potentially incorrectly, and the code might indeed corrupt the free list, but the code is (in the first step) blocked by the proof rules.

5. THE EYE OF THE ASSERTER

In Table IV we give a queue module. In the specification we use a predicate $\mathit{listseg}(x, \alpha, y)$ which says that there is an acyclic linked list from x to y that has the sequence α in its head components. The variable Q denotes the sequence of values currently held in the queue; it is present in the resource invariant, as well as in the interface specifications. (Technically, we would have to ensure that the variable Q was added to the s component of our semantics.) This exposing of "auxiliary" variables is standard in module specifications, as is the inclusion of assignment statements involving auxiliary variables whose only purpose is to enable the specification to work.

This queue module keeps a sentinel at the end of its internal list, as is indicated by $(y \mapsto -, -)$ in the resource invariant. The sentinel does not hold any value in the queue, but reserves storage for a new value.

An additional feature of the treatment of queues is the predicate P, which is required to hold for each element of the sequence α. By instantiating P in various ways we obtain versions of the queue module that transfer different amounts of storage.

—$P(v) =$ `emp`: plain values are transferred in and out of the queue, and no storage is transferred with any of these values;

—$P(v) = v \mapsto -, -$: binary cons cells, and ownership of the storage associated with them, are transferred in and out of the queue;

—$P(v) = \mathit{list}(v)$: linked lists, and ownership of the storage associated with them, are transferred in and out of the queue.

To illustrate the difference between these cases, consider the following attempted proof steps in client code.

$$\{Q = \langle n \rangle \cdot \alpha \wedge \texttt{emp}\}$$
$$\texttt{deq}$$

Interface Specifications

$\{Q = \alpha \wedge z = n \wedge P(z)\}\ \texttt{enq}\ \{Q = \alpha{\cdot}\langle n\rangle \wedge \texttt{emp}\}\ [Q]$
$\{Q = \langle m\rangle{\cdot}\alpha \wedge \texttt{emp}\}\ \texttt{deq}\ \{Q = \alpha \wedge z = m \wedge P(z)\}\ [Q, z]$
$\{\texttt{emp}\}\ \texttt{isempty?}\ \{(w = (Q = \epsilon)) \wedge \texttt{emp}\}\ [w]$

Resource Invariant: $listseg(x, Q, y) * (y \mapsto -, -)$

Private Variables: x, y, t

listseg Predicate Definition

$$listseg(x, \alpha, y) \stackrel{def}{\Longleftrightarrow} \texttt{if } x = y \texttt{ then } (\alpha = [] \wedge \texttt{emp}) \texttt{ else } \Big(\exists v, z, \alpha'.\ (\alpha = \langle v\rangle{\cdot}\alpha' \wedge x \mapsto v, z)\ *\ P(v)\ *\ listseg(z, \alpha', y)\Big)$$

Internal Implementations

$Q := Q{\cdot}\langle z\rangle$;	(code for `enq`)
$t := \texttt{cons}(-,-)$; $y.1 := z$; $y.2 := t$; $y := t$	
$Q := \texttt{cdr}(Q)$;	(code for `deq`)
$z := x.1$; $t := x$; $x := x.2$; $\texttt{dispose}(t)$	
$w := (x = y)$	(code for `isempty?`)

Table IV. Queue Module, Parametric in $P(v)$

$$\{Q = \alpha \wedge z = n \wedge P(z)\}$$
$$z.1 := 42$$
$$\{???\}$$

In case $P(v)$ is either `emp` or $list(v)$ we cannot fill in ??? because we do not have the right to dereference z in the precondition of $z.1 := 42$. However, if $P(v)$ is $v \mapsto -, -$ then we will have this right, and a valid postcondition is $(Q = \alpha \wedge z = n \wedge z \mapsto 42, -)$. Conversely, if we replace $z.1 := 42$ by code that traverses a linked list then the third definition of $P(v)$ will enable a verification to go through, where the other two will not.

On the other hand there is no operational distinction between these three cases: the queue code just copies values.

The upshot of this discussion is that the idea of ownership transfer we have alluded to is not determined by instructions in the programming language alone. Just what storage is, or is not, transferred depends on which definition of P we choose. And this choice depends on what we want to prove.

This phenomenon, where "Ownership is in the eye of the Asserter", can take some getting used to at first. One might feel ownership transfer might be made an explicit operation in the programming language. In some cases such a programming practice would be useful, but the simple fact is that in real programs the amount of resource transferred is not always determined operationally; rather, there is an understanding between a module writer, and programmers of client code. For example, when you call `malloc()` you just receive an address. The implementation of `malloc()` does not include explicit statements that transfer each of several cells

to its caller, but the caller understands that ownership of several cells comes with the single address it receives.

6. A CONUNDRUM

In the following 0 is the assertion `emp` that the heap is empty, and 1 says that it has precisely one active cell, say x (so 1 is $x \mapsto -$).

Consider the following instance of the hypothetical frame rule, where `true` is chosen as the invariant:

$$\frac{\{0 \vee 1\}k\{0\}[] \vdash \{1\}k\{\mathtt{false}\}}{\{(0 \vee 1) * \mathtt{true}\}k\{0 * \mathtt{true}\}[] \vdash \{1 * \mathtt{true}\}k\{\mathtt{false} * \mathtt{true}\}}$$

The conclusion is definitely false in any sensible semantics of sequents. For example, if k denotes the do-nothing command, `skip`, then the antecedent holds, but the consequent does not.

However, we can derive the premise $\{0 \vee 1\}k\{0\}[] \vdash \{1\}k\{\mathtt{false}\}$ as follows.

$$\dfrac{\dfrac{\dfrac{\{0 \vee 1\}k\{0\}}{\{1\}k\{0\}}\ \text{Consequence} \qquad \dfrac{\dfrac{\dfrac{\{0 \vee 1\}k\{0\}}{\{0\}k\{0\}}\ \text{Consequence}}{\{0 * 1\}k\{0 * 1\}}\ \text{Ordinary Frame Rule}}{\{1\}k\{1\}}\ \text{Consequence}}{\{1 \wedge 1\}k\{1 \wedge 0\}}\ \text{Conjunction}}{\{1\}k\{\mathtt{false}\}}\ \text{Consequence}$$

This shows that we cannot have all of: the usual rule of consequence, the ordinary frame rule, the conjunction rule, and the hypothetical frame rule. It also shows that the idea of treating information hiding as syntactic sugar for proof and specification forms should be approached with caution: one needs to be careful that introduced sugar does not interact badly with expected rules, in a way that contradicts them.

The counterexample can also be presented as a module, as in Table V. This can be used to show a similar problem with the modular procedure rule.

Given this counterexample, the question is where to place the blame. There are several possibilities.

(1) The specification $\{0 \vee 1\}k\{0\}$. This is an unusual specification, since in the programming languages we have been using there is no way to branch on whether the heap is empty.

(2) The invariant `true`. Intuitively, a resource invariant should precisely identify an unambiguous area of storage, that owned by a module. The invariant $list(f)$ in the memory manager is unambiguous in this sense, where `true` is perhaps not.

(3) One of the rules of conjunction, consequence, or the ordinary frame rule.

The remainder of the paper is occupied with a theoretical analysis of this situation. Our main results concentrate on possibilities (1) and (2) just listed. We identify a technical notion of "precise" predicate, which picks out an unambiguous area of storage. By restricting the preconditions of procedure specifications to be precise, we obtain a sound semantics, and this reaction corresponds to point (1).

Interface Specifications

$\{0 \vee 1\}k\{0\}$ []

Resource Invariant

`true`

Internal Implementation

```
{true ∗ (0 ∨ 1)}          (code for k)
 skip;
{true ∗ 0}
```

Table V. Counterexample Module

By restricting resource invariants to be precise we also obtain a sound semantics, and this reaction corresponds to point (2).

We also briefly point out a model of the hypothetical frame rule which places no restrictions on it, but which invalidates the conjunction rule. This corresponds to reaction (3). The reaction (3) is not one we would attempt to defend on conceptual grounds, but it is interesting to know that reaction (3) can be upheld, technically. (Furthermore, some follow-up works adopt reaction (3), starting with [Birkedal et al. 2005].)

7. THE PROGRAMMING LANGUAGE MODEL

Until now in work on separation logic we have used operational semantics, but in this paper we use a denotational semantics. By using denotational semantics we will be able to reduce the truth of a sequent $\Gamma \vdash \{p\}C\{q\}$ to the truth of a single semantic triple $\{p\}[\![C]\!]\eta\{q\}$ where η maps each procedure identifier in Γ to a "greatest" or "most general" relation satisfying it [Schwarz 1974; 1977; Morgan 1988]. In the case of the hypothetical frame rule, we will be able to compare two denotations of the same command for particular instantiations of the procedure identifiers, rather than having to quantify over all possible instantiations. Our choice to use denotational semantics here is entirely pragmatic: The greatest relation is not always definable by a term in a given programming language, but the ability to refer to it leads to significant simplifications in proofs about the semantics.

The model we use is chosen for its simplicity. Only basic domain theory is required, essentially, continuity and fixed-points. Commands are modelled as relations, which are ordered by inclusion to form a complete partial order in terms of which the constructs of our language are continuous. The model is adequate for partial correctness, but not total correctness.

7.1 Domains

To begin with, we divide the program state into two components, the "stack" which maps variables into values, and the "heap" which is the H component from

the previous section.

$$\begin{aligned} S &\stackrel{def}{=} \mathsf{Variables} \to \mathsf{Values} \\ H &\stackrel{def}{=} L \rightharpoonup_{fin} R \\ \mathsf{States} &\stackrel{def}{=} S \times H \end{aligned}$$

Often the set Values can be taken to be the same as R, but we do not require this.

A command is interpreted as a relation

$$\mathsf{States} \leftrightarrow \mathsf{States} \cup \{\mathtt{fault}\}$$

satisfying certain properties defined below. Because we use a fault-avoiding interpretation of triples, it would be possible to use the domain

$$\mathsf{States} \to \mathcal{P}(\mathsf{States}) \cup \{\mathtt{fault}\}$$

instead. Using the more general domain lets us see clearly that if a command nondeterministically chooses between `fault` and some state, then the possibility of faulting will mean that the command is not well specified. This is not an essential point; the more constrained domain could be used without affecting any of our results.

This domain of relations is inappropriate for total correctness because it does not include a specific result for non-termination, and our semantics will not distinguish a command C from one that nondeterministically chooses C or divergence. Note that divergence is distinguished from `fault`, which typically occurs when an l-value not in the current state is accessed.

We say that a relation $c\colon \mathsf{States} \leftrightarrow \mathsf{States} \cup \{\mathtt{fault}\}$ is *safe* at a state (s,h) when $\neg(s,h)\,[c]\,\mathtt{fault}$. Intuitively, this means that when started at (s,h), the "command" c does not dereference dangling pointers. The two locality properties are:

(1) Safety Monotonicity: for all states (s,h) and heaps h_1 such that $h\#h_1$, if c is safe at (s,h), it is also safe at $(s,h*h_1)$.

(2) Frame Property: for all states (s,h) and heaps h_1 such that $h\#h_1$, if c is safe at (s,h) and $(s,h*h_1)[c](s',h')$, then there is a subheap $h'_0 \le h'$ such that

$$h'_0\#h_1,\ h'_0 * h_1 = h' \text{ and } (s,h)[c](s',h'_0).$$

The poset LRel of "local relations" is the set of all such c satisfying the above conditions, ordered by subset inclusion:

$$\mathsf{LRel} \stackrel{def}{=} \{c\colon \mathsf{States} \leftrightarrow \mathsf{States} \cup \{\mathtt{fault}\} \mid c \text{ satisfies safety monotonicity and the frame property}\}$$

These local relations are exactly those that satisfy the ordinary frame rule. Local relations inherit the domain structure from relations, ordered by subset inclusion, which gives us access to fixed-points.

LEMMA 1. *The poset* LRel *is a chain-complete partial order with the least element. The least element is the empty relation, and the least upper bound of a chain is given by the union of all the relations in the chain.*

Proof: Since the empty relation is in LRel, the poset LRel has the least element. To see that LRel is chain-complete, consider a chain $\{c_i\}_i$ in LRel. We need to show that $\bigcup_i c_i$ satisfies safety monotonicity and the frame property. Suppose that $\bigcup_i c_i$ is safe at a state (s,h). Then, all of c_i's are safe at (s,h). The safety monotonicity of c_i, then, says that c_i is safe at all bigger states $(s, h * h_1)$ with $h\#h_1$. This gives the safety monotonicity of $\bigcup_i c_i$. For the frame property, suppose that $\bigcup_i c_i$ is safe at a state (s,h), and that $(s, h * h_1)[\bigcup_i c_i](s', h')$ for some h_1, s', h' with $h\#h_1$. Then, there is some c_i such that $(s, h * h_1)[c_i](s', h')$. The frame property of c_i gives a subheap $h'_0 \leq h'$ such that $(s,h)[c_i](s', h'_0)$ and $h' = h'_0 * h_1$. This h'_0 is the subheap required for the frame property of $\bigcup_i c_i$ because $(s,h)[\bigcup_i c_i](s', h'_0)$ and $h' = h'_0 * h_1$. □

7.2 The General Semantics

We consider a simple imperative programming language extended with parameterless procedures. For the moment, we assume that we are given an unspecified set of basic commands, and a fixed element $\overline{A} \in$ LRel for each basic command A; the development of the semantics of the surrounding language is parametric in the choice of the sets L, R, and Values, and the basic commands. We will separately instantiate the basic commands as appropriate to the RAM model.

The semantics is in Table VI. The meaning of a command is given in the context of an environment η, which maps procedure identifiers to relations in LRel. We presume a semantics which assigns a truth value to $[\![B]\!]s$.

LEMMA 2. *For each command C, $[\![C]\!]$ is well-defined: for all environments η, $[\![C]\!]\eta$ is in* LRel*, and $[\![C]\!]\eta$ is continuous for η when environments are ordered pointwise.*

Proof: We use induction on the structure of C to show the lemma. When C is a basic command the result holds by presumption.

When $C = C_1; C_2$, we first show that $[\![C_1; C_2]\!]\eta$ satisfies safety monotonicity and the frame property so that it is in LRel; then, we will prove that $[\![C_1; C_2]\!]$ is continuous. Consider a state (s,h) and an environment η such that $[\![C_1; C_2]\!]\eta$ is safe at the state (s,h). Suppose that this $[\![C_1; C_2]\!]\eta$ is not safe at some bigger state $(s, h * h_1)$ with $h\#h_1$. Then, we have either $(s, h * h_1)[[\![C_1]\!]\eta]\texttt{fault}$, or $(s, h * h_1)[[\![C_1]\!]\eta](s', h')$ and $(s', h')[[\![C_2]\!]\eta]\texttt{fault}$ for some state (s', h'). We get the required contradiction in both cases. In the first case, since $[\![C_1; C_2]\!]\eta$ is safe at (s,h), the $[\![C_1]\!]\eta$ is also safe at (s,h). Therefore, because of the induction hypothesis, the "command" $[\![C_1]\!]\eta$ can not generate `fault` when run in the bigger state $(s, h * h_1)$. This contradicts the assumption, $(s, h * h_1)[[\![C_1]\!]\eta]\texttt{fault}$. In the second case, note that since $[\![C_1]\!]\eta$ is safe at (s,h), the induction hypothesis gives a subheap h'_0 of h' such that the subheap h'_0 is disjoint from h_1 (i.e., $h'_0\#h_1$) and satisfies $h' = h'_0 * h_1$ and $(s,h)[[\![C_1]\!]\eta](s', h'_0)$. Since $[\![C_1; C_2]\!]\eta$ is safe at (s,h), $[\![C_2]\!]\eta$ should be safe at (s', h'_0). Now, the induction hypothesis for C_2 implies that $[\![C_2]\!]\eta$ is also safe at $(s', h'_0 * h_1)$; this contradicts our assumption that $(s', h')[[\![C_2]\!]\eta]\texttt{fault}$ holds. For the frame property, consider a heap h_1 disjoint from the heap h, and a state (s'', h'') such that $(s, h * h_1)[[\![C_1; C_2]\!]\eta](s'', h'')$. We must find a subheap h''_0 of h''

Atomic Commands (Assumed)

$\overline{A} \in \mathsf{LRel}$

Functionality of Environments

$\eta \in \mathtt{ProcIds} \to \mathsf{LRel}$

Functionality of Valuations

$[\![C]\!]\eta \in \mathsf{LRel}$

Valuations

for all $(s,h) \in \mathsf{States}$ and $a \in \mathsf{States} \cup \{\mathtt{fault}\}$,

$[\![A]\!]\eta \stackrel{def}{=} \overline{A}$

$(s,h)[[\![\mathtt{skip}]\!]\eta]a \stackrel{def}{\iff} a = (s,h)$

$(s,h)[[\![C_1; C_2]\!]\eta]a \stackrel{def}{\iff} (s,h)[seq([\![C_1]\!]\eta, [\![C_2]\!]\eta)]a$

$(s,h)[[\![\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2]\!]\eta]a \stackrel{def}{\iff} (s,h)\Big[[\![B]\!] \rightsquigarrow [\![C_1]\!]\eta; [\![C_2]\!]\eta\Big]a$

$(s,h)[[\![\mathtt{while}\ B\ C]\!]\eta]a \stackrel{def}{\iff} (s,h)\Big[fix\ \Big(\lambda c \in \mathsf{LRel}.\ ([\![B]\!] \rightsquigarrow seq([\![C]\!]\eta, c); [\![\mathtt{skip}]\!]\eta)\Big)\Big]a$

$(s,h)[[\![k]\!]\eta]a \stackrel{def}{\iff} (s,h)[\eta(k)]a$

$(s,h)[[\![\mathtt{letrec}\ k_1 = C_1, \ldots, k_n = C_n\ \mathtt{in}\ C]\!]\eta]a \stackrel{def}{\iff} (s,h)\Big[[\![C]\!]\eta[k_1 \mapsto d_1, \ldots, k_n \mapsto d_n]\Big]a$

where *fix* f gives the least fixed-point of f, and $seq(c_1, c_2)$, $b \rightsquigarrow c_1; c_2$ and $d_1, ..., d_n$ are defined as follows:

$$\begin{array}{rcl}
(d_1, \ldots, d_n) & \stackrel{def}{=} & fix(\lambda d_1, \ldots, d_n \in \mathsf{LRel}^n.\ (F_1, ..., F_n)) \\
 & & \text{where } F_i = [\![C_i]\!]\eta[k_1 \mapsto d_1, \ldots, k_n \mapsto d_n] \\
(s,h)\,[seq(c_1, c_2)]\,a & \stackrel{def}{\iff} & \Big(\exists (s',h').\ (s,h)\,[c_1]\,(s',h') \wedge (s',h')\,[c_2]\,a\Big) \\
 & & \vee \Big((s,h)\,[c_1]\,\mathtt{fault} \wedge a = \mathtt{fault}\Big) \\
(s,h)\,[b \rightsquigarrow c_1; c_2]\,a & \stackrel{def}{\iff} & \text{if } b(s) = true \text{ then } (s,h)[c_1]a \text{ else } (s,h)[c_2]a
\end{array}$$

Table VI. The General Semantics

that satisfies

$$h_0'' \# h_1, \quad h_0'' * h_1 = h'', \quad \text{and} \quad (s,h)[[\![C_1; C_2]\!]\eta](s'', h_0'').$$

Since $(s, h * h_1)[[\![C_1; C_2]\!]\eta](s'', h'')$ holds, there is an intermediate state (s', h') for this computation. That is, the state (s', h') satisfies $(s, h * h_1)[[\![C_1]\!]\eta](s', h')$ and $(s', h')[[\![C_2]\!]\eta](s'', h'')$. Applying the induction hypothesis for C_1 and then for C_2, we can find a subheap h_0' of h' and h_0'' of h'' such that the heap h_0' satisfies $h' = h_0' * h_1$ and $(s,h)[[\![C_1]\!]\eta](s', h_0')$, and the heap h_0'' satisfies $h'' = h_0'' * h_1$ and $(s', h_0')[[\![C_2]\!]\eta](s'', h_0'')$. The heap h_0'' is the required subheap of h'' for the frame property of $[\![C_1; C_2]\!]\eta$.

For the continuity of $[\![C_1; C_2]\!]$, it suffices to show that *seq* preserves the least upper bound of a chain in $\mathsf{LRel} \times \mathsf{LRel}$. Let $\{(c_i, c_i')\}_i$ be a chain in $\mathsf{LRel} \times \mathsf{LRel}$. By Lemma 1, their least upper bound is $(\bigcup_i c_i, \bigcup_j c_j')$. The following shows that *seq*

preserves this least upper bound:

$$
\begin{array}{ll}
& (s,h)[seq(\bigcup_i c_i, \bigcup_j c'_j)]a \\
\iff & \\
& \big(\exists(s',h').\,(s,h)[\bigcup_i c_i](s',h') \wedge (s',h')[\bigcup_j c'_j]a\big) \vee \big((s,h)[\bigcup_i c_i]a \wedge a = \texttt{fault}\big) \\
\iff & (\because \{(c_i, c'_i)\}_i \text{ is a chain}) \\
& \exists i.\, \big(\exists(s',h').\,(s,h)[c_i](s',h') \wedge (s',h')[c'_i]a\big) \vee \big((s,h)[c_i]a \wedge a = \texttt{fault}\big) \\
\iff & \\
& \exists i.\,(s,h)[seq(c_i, c'_i)]a \\
\iff & \\
& (s,h)[\bigcup_i (seq(c_i, c'_i))]a.
\end{array}
$$

When $C = \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$, we note that the boolean expression B does not depend on the heap so that the value of B at a smaller state (s,h) is the same as that at a bigger state $(s, h * h_1)$. Moreover, both $[\![C_1]\!]\eta$ and $[\![C_2]\!]\eta$ already satisfy safety monotonicity and the frame property by the induction hypothesis. Therefore, $[\![C]\!]\eta$ also satisfies these two properties. For the continuity, it suffices to show that for all boolean functions $b \in [S \to \{true, false\}]$, and all chains $\{c_i\}_i$ and $\{c'_i\}_i$ of local relations, we have

$$(b \rightsquigarrow \bigcup_i c_i; \bigcup_i c'_i) \;=\; \bigcup_i (b \rightsquigarrow c_i; c'_i).$$

We show this equality as follows:

$$
\begin{array}{ll}
& (s,h)\,[(b \rightsquigarrow \bigcup_i c_i; \bigcup_i c'_i)]\,a \\
\iff & \mathsf{if}\ b(s) = true\ \mathsf{then}\ (\exists i.\,(s,h)[c_i]a)\ \mathsf{else}\ (\exists i.\,(s,h)[c'_i]a) \\
\iff & \exists i.\,\mathsf{if}\ b(s) = true\ \mathsf{then}\ (s,h)[c_i]a\ \mathsf{else}\ (s,h)[c'_i]a \\
\iff & (s,h)[\bigcup_i (b \rightsquigarrow c_i; c'_i)]a.
\end{array}
$$

The meaning of a procedure call k is a projection, which is a well-defined continuous map from environments to local relations.

For the remaining cases of loop and procedure definition, we just observe that the domain LRel^n is a cpo, and that the least fixed-point operator *fix* is, itself, continuous from the domain of continuous functions of type $\mathsf{LRel}^n \to \mathsf{LRel}^n$ to LRel^n. The conclusion follows from this because the functionals, for which we take the least fixed-point in both cases, are well-defined continuous functions from environments to continuous functions of type $\mathsf{LRel}^n \to \mathsf{LRel}^n$. □

7.3 A Particular Semantics: The RAM Model

We now instantiate the previous development for the RAM model. The domains are set out, together with the valuations for basic commands, in Table VII.

The command $x := \texttt{cons}(E_1, \ldots, E_n)$ allocates n consecutive cells, initializes them with the values of $E_1, \ldots, E_n$, and stores the address of the first cell in x. The contents of an address E can be read and stored to x by $x := [E]$, or can be modified by $[E] := E_1$. The command $\texttt{dispose}(E)$ deallocates the address E. Note that in $x := [E]$, $[E] := E_1$ and $\texttt{dispose}(E)$, the expression E can be an arbitrary arithmetic expression; so, this language allows address arithmetic. Also, note that

Expressions (Assumed)

$$[\![E]\!]s \in \mathsf{Ints} \qquad [\![B]\!]s \in \{true, false\} \qquad (\text{where } s \in S).$$

Domains for the RAM model

$$\mathsf{Nats} \stackrel{def}{=} \{0, 1, \ldots, 17, \ldots\} \qquad \mathsf{Ints} \stackrel{def}{=} \{\ldots, -17, \ldots, -1, 0, 1, \ldots, 17, \ldots\}$$
$$\mathsf{Variables} \stackrel{def}{=} \{x, y, \ldots\} \qquad S \stackrel{def}{=} \mathsf{Variables} \to \mathsf{Ints}$$
$$H \stackrel{def}{=} \mathsf{Nats} \rightharpoonup_{fin} \mathsf{Ints} \qquad \mathsf{States} \stackrel{def}{=} S \times H$$

Valuations for Atomic Commands

for all $(s,h) \in \mathsf{States}$ and $a \in \mathsf{States} \cup \{\mathtt{fault}\}$,

$$(s,h)[\overline{x := E}]a \stackrel{def}{\iff} a = (s[x \mapsto [\![E]\!]s], h)$$

$$(s,h)[\overline{x := \mathtt{cons}(E_1, \ldots, E_n)}]a$$
$$\stackrel{def}{\iff} \exists m.\ (m, \ldots, m+n-1 \notin \mathsf{dom}(h)) \wedge \Big(a = (s[x \mapsto m], h * [m \mapsto [\![E_1]\!]s, \ldots, m+n-1 \mapsto [\![E_n]\!]s])\Big)$$

$$(s,h)[\overline{x := [E]}]a \stackrel{def}{\iff} \mathsf{if}\ [\![E]\!]s \in \mathsf{dom}(h)\ \mathsf{then}\ a = (s[x \mapsto h([\![E]\!]s)], h)\ \mathsf{else}\ a = \mathtt{fault}$$

$$(s,h)[\overline{[E] := E_1}]a \stackrel{def}{\iff} \mathsf{if}\ [\![E]\!]s \in \mathsf{dom}(h)\ \mathsf{then}\ a = (s, h[[\![E]\!]s \mapsto [\![E_1]\!]s])\ \mathsf{else}\ a = \mathtt{fault}$$

$$(s,h)[\overline{\mathtt{dispose}(E)}]a \stackrel{def}{\iff} \mathsf{if}\ [\![E]\!]s \in \mathsf{dom}(h)\ \mathsf{then}\ a = (s,h')\ \text{for } h' \text{ s.t. } h' * ([\![E]\!]s \mapsto h([\![E]\!]s)) = h\ \mathsf{else}\ a = \mathtt{fault}$$

Table VII. The RAM Semantics

dereferencing or disposing a pointer not in the domain of the heap leads to a `fault` in the semantics.

We can now specify the element $\overline{A} \in \mathsf{LRel}$ for each basic command. It is a matter of straightforward checking to show that each of these definitions satisfies safety monotonicity and the frame property, so we state

LEMMA 3. *For each basic command A, we have $\overline{A} \in \mathsf{LRel}$.*

It is entertaining to see the nondeterminism at work in the semantics of `cons` in this model. In particular, since we are aiming for partial correctness, the semantics does not record whether a command terminates or not; for instance, $x := 1; y := 1$ has the same denotation as a command that nondeterministically picks either $x := 1; y := 1$ or divergence. Such a nondeterministic command can be expressed in our language as

$$x := \mathtt{cons}(0); \mathtt{dispose}(x); y := \mathtt{cons}(0); \mathtt{dispose}(y);$$
$$\mathtt{if}\ (x = y)\ \mathtt{then}\ (x := 1; y := 1)$$
$$\mathtt{else}\ (\mathtt{while}\ (x = x)\ \mathtt{skip})$$

The reader may enjoy verifying that this is indeed equivalent to $x := 1; y := 1$ in the model.

8. SEMANTICS OF SEQUENTS

In this section we give a semantics where a sequent

$$\Gamma \vdash \{p\}C\{q\}$$

says that if every specification in Γ is true of an environment mapping procedure identifiers to local relations, then so is $\{p\}C\{q\}$

To interpret sequents we define semantic cousins of the modifies clauses and Hoare triples. If $c \in \mathsf{LRel}$ is a relation then

—$\mathsf{modifies}(c, X)$ holds if and only if whenever $(s,h)[c](s',h')$ and $y \notin X$, we have that $s(y) = s'(y)$.

—$\{p\}c\{q\}$ holds if and only if for all states (s,h) in p,
(1) $\neg((s,h)\,[c]\,\texttt{fault})$; and
(2) for all states (s',h'), if $(s,h)\,[c]\,(s',h')$, the state (s',h') is in q.

Intuitively, triple $\{p\}c\{q\}$ says that when started at a state in p, the relation c does not generate a `fault` so that it only accesses cells already in h or newly allocated during execution, and, if c ever terminates, it always produces a state in q.

The interpretation of a sequent now works by quantifying over values in environments, and appealing to the semantic counterparts of triples and modifies clauses.

Definition 4. [Validity of Sequents]
A sequent

$$\{p_1\}k_1\{q_1\}[X_1] \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}$$

holds in the standard semantics if and only if

> *for all environments η, if both $\{p_i\}\eta(k_i)\{q_i\}$ and $\mathsf{modifies}(\eta(k_i), X_i)$ hold for all $1 \leq i \leq n$, the triple $\{p\}([\![C]\!]\eta)\{q\}$ also holds.*

It is interesting to see the position of this semantics on the sequents used in the counterexample from Section 6. The standard semantics validates the conjunction rule

$$\frac{\Gamma \vdash \{p\}C\{q\} \quad \Gamma \vdash \{p'\}C\{q'\}}{\Gamma \vdash \{p \wedge p'\}C\{q \wedge q'\}}$$

and so also the instance of it

$$\frac{\{0 \vee 1\}k\{0\}[] \vdash \{1\}k\{1\} \quad \{0 \vee 1\}k\{0\}[] \vdash \{1\}k\{0\}}{\{0 \vee 1\}k\{0\}[] \vdash \{1 \wedge 1\}k\{1 \wedge 0\}}$$

used in the derivation of $\{0 \vee 1\}k\{0\}[] \vdash \{1\}k\{\texttt{false}\}$. This concluding sequent is true because any relation $c \in \mathsf{LRel}$ satisfying $\{0 \vee 1\}k\{0\}$ must diverge given a state satisfying 1; thus, the triple $\{1\}k\{0\}$ will hold since divergence makes a triple true in partial correctness.

The standard semantics, however, invalidates the hypothetical frame rule: this can be seen, as indicated in Section 6, by taking k to be the identity relation in LRel and r to be the entire set of states.

9. PRECISE PREDICATES

We know from the counterexample in Section 6 that we must restrict the hypothetical frame rule in some way, if it is to be used with the standard semantics. Before describing the restriction, let us retrace some of our steps. We had a situation where

ownership could transfer between a module and a client, which made essential use of the dynamic nature of $*$. But we had also got to a position where ownership is determined by what the Asserter asserts, and this put us in a bind: when the Asserter does not precisely specify what storage is owned, different splittings can be chosen at different times using the nondeterministic semantics of $*$; this fools the hypothetical frame rule (it is perhaps fortuitous that the nondeterminism in $*$ has not gotten us into trouble in separation logic before now). A way out of this problem is to insist that the Asserter precisely nail down the storage that he or she is talking about.

DEFINITION 5. [Precision] *A predicate p is* precise *if and only if for all states (s,h), there is at most one subheap h_p of h for which $(s,h_p) \in p$.*

Intuitively, this definition says that for each state (s,h), a precise predicate unambiguously specifies the portion of the heap h that is relevant to the predicate. Formulae that describe data structures are often precise. Indeed, the definition might be viewed as a formalization of a point of view stressed by Richard Bornat, that for practically any data structure one can write a formula or program that searches through a heap and picks out the relevant cells. Bornat used this idea to expose the cells in a data structure in order to give an approach to spatial separation in traditional logic [Bornat 2000]. In separation logic we express spatial separation using $*$, and have until now had less reason to expose, or insist on unambiguous identification of, the cells in a data structure.

An example of a precise predicate is the following one for list segments:

$$\mathit{listseg}(x,y) \stackrel{def}{\Longleftrightarrow} (x = y \wedge \mathtt{emp}) \vee \big(x \neq y \wedge \exists z.\, (x \mapsto -, z) \ast \mathit{listseg}(z,y)\big)$$

This predicate is true when the heap contains a non-circular linked list (and nothing else), which starts from the cell x and ends with y. Note that because of $x \neq y$ in the second disjunct, the predicate $\mathit{listseg}(x,y)$ says that if x and y have the same value in a state (s,h), the heap h must be empty. If we had left $x \neq y$ out of the second disjunct, then $\mathit{listseg}(x,y)$ would not be precise: $\mathit{listseg}(x,x)$ could be true of a heap containing a non-empty circular list from x to x (and nothing else), and also of the empty heap, a proper subheap.

If p is a precise predicate then there can be at most one way to split any given heap up in such a way as to satisfy $p * q$; the splitting, if there is one, must give p the unique subheap satisfying it. This leads to an important property of precise predicates.

LEMMA 6. [Distribution Lemma]
A predicate p is precise if and only if $p * -$ distributes over $\wedge$:

for all predicates q and r, we have $p * (q \wedge r) = (p * q) \wedge (p * r)$.

Proof: We show the only-if direction first. By the definition of $*$, predicate $p*(q \wedge r)$ is always included in $(p * q) \wedge (p * r)$ for all predicates p, q, r. So, it suffices to show the other inclusion. Let (s,h) be a state in $(p * q) \wedge (p * r)$. Then, heap h can be split into h_p, h_q, and also into h'_p, h'_r such that

(1) both (s,h_p) and (s,h'_p) are in p,
(2) (s,h_q) is in q and (s,h'_r) is in r, and

(3) $h = h_p * h_r = h'_p * h'_r$ holds.

Since predicate p is precise, $h_p = h'_p$. So, $h_q = h_r$. The conclusion follows from this.

For the if direction, suppose that $p * -$ distributes over $\wedge$ but p is not precise. Then, there is a state (s, h) and two different subheaps h_p, h'_p of h with $(s, h_p) \in p$ and $(s, h'_p) \in p$. Let $q = \{(s, h - h_p)\}$ and let $r = \{(s, h - h'_p)\}$, where for all subheaps $h' \leq h$, heap $h - h'$ denotes h excluding those cells in h'. Then, (s, h) is in $(p * q) \wedge (p * r)$, but it is not in $p * (q \wedge r)$ because $q \wedge r$ is the empty set. This contradicts the distributivity of $p * -$. □

We also have closure properties of precise predicates.

LEMMA 7. *For all precise predicates p and q, all (possibly imprecise) predicates r, and boolean expressions B, all the predicates $p \wedge r$, $p * q$, and $(B \wedge p) \vee (\neg B \wedge q)$ are precise.*

As a first hint of the relevance of the notion of precise predicate to the conundrum from Section 6, consider how an inference using the usual conjunction rule

$$\frac{\{p\}k\{q\} \quad \{p'\}k\{q'\}}{\{p \wedge p'\}k\{q \wedge q'\}}$$

relates to a putative inference after we place invariants on the premises and the conclusion:

$$\frac{\{p * r\}k\{q * r\} \quad \{p' * r\}k\{q' * r\}}{\{(p \wedge p') * r\}k\{(q \wedge q') * r\}} \quad ?$$

We can almost perform this inference

$$\frac{(p \wedge p') * r \Rightarrow (p * r) \wedge (p' * r) \quad \dfrac{\{p * r\}k\{q * r\} \quad \{p'\}k\{q'\}}{\{(p * r) \wedge (p' * r)\}k\{(q * r) \wedge (q' * r)\}}}{\{(p \wedge p') * r\}k\{(q * r) \wedge (q' * r)\}}$$

and if we had one more implication

$$(q * r) \wedge (q' * r) \Rightarrow (q \wedge q') * r$$

then we would obtain the putative conclusion, using the usual rule of consequence.

Alas, as we have seen, this implication fails in general: take $q = 0$, $q' = 1$ and $r = \texttt{true}$. But, if r is precise then the implication goes through, because of the distribution lemma, and the putative conclusion is then fully justified.

This gives us a hint of the relevance of precision; now we undertake to provide a detailed analysis. Before presenting the soundness proof of the hypothetical frame rule in the standard semantics, we clarify the side conditions for the rule.

10. VARIABLE CONDITIONS

We repeat the hypothetical frame rule, with its side conditions.

Hypothetical Frame Rule

$$\frac{\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\Gamma, \{p_1 * r\}k_1\{q_1 * r\}[X_1, Y], \ldots, \{p_n * r\}k_n\{q_n * r\}[X_n, Y] \vdash \{p * r\}C\{q * r\}}$$

where
—C does not modify variables in r, *except through using* $k_1, ..., k_n$; and
—Y is disjoint from "$\Gamma, \{p_1\}k\{q_1\}[X_1], \ldots, \{p_n\}k\{q_n\}[X_n] \vdash \{p\}C\{q\}$".

As a first comment, note that in the rule we are using comma between the X_i and Y for union of disjoint sets; the form of the rule therefore assumes that X_i and Y are disjoint.

The disjointness requirement for Y enforces that we do not observe the changes of a variable in Y while reasoning about C; as a result, reasoning in client code is independent of variables in Y. We give a technical definition of several variants on a notion of disjointness of a set of variables X from a set of variables, a command, a predicate, or a judgment. X is disjoint from a set Y if their variables do not overlap; X is disjoint from a command C if X does not intersect with the free variables of C; X is disjoint from predicate r if the predicate is invariant under changes to values of variables in X; X is disjoint from judgment $\Gamma' \vdash \{p\}C\{q\}$ if it is disjoint from p, q, C in the concluding triple and also from p', q', Y for all $\{p'\}k\{q'\}[Y]$ in Γ'. This defines the second side condition.

The first side condition can be made rigorous with a relativized version of the usual notion of set of variables modified by a command. We describe this using a set $\mathit{Modifies}(C)(\Gamma)$ of variables associated with each command, where we split the context into two parts. The two most important clauses in the definition concern procedure call.

$$\begin{aligned} \mathit{Modifies}(k)(\Gamma) &= X, \quad \text{if } \{p\}k\{q\}[X] \in \Gamma \\ \mathit{Modifies}(k)(\Gamma) &= \{\}, \quad \text{otherwise} \end{aligned}$$

The upshot is that $\mathit{Modifies}(C)(\Gamma)$ reports those variables modified by C, except that it doesn't count any procedure calls for procedures not in Γ.

For the other commands, the relativized notion of modifies set is defined usual. For a compound command C with immediate subcommands $C_1, \ldots, C_n$, the set $\mathit{Modifies}(C)(\Gamma)$ is the union $\cup_i \mathit{Modifies}(C_i)(\Gamma)$. Two of the basic commands are as follows:

$$\mathit{Modifies}(x := E)(\Gamma) = \{x\} \qquad \mathit{Modifies}([x] := E)(\Gamma) = \{\}$$

For $[x] := E$ the modifies set is empty because the command alters the heap but not the stack.

We are now in a position to state the first side condition rigorously: it means

$\mathit{Modifies}(C)(\Gamma)$ is disjoint from r.

The modifies conditions for the the ordinary frame and recursive procedure rules do not mention the "except through" clause. These can be formalized by taking Γ in $\mathit{Modifies}(C)(\Gamma)$ to be the entire context of the premise.

An important point is that the free variables of the resource invariant are allowed to overlap with the X_i. This often happens when using auxiliary variables to specify the behaviour of a module, as exemplified by the treatment of the auxiliary variable Q in the queue module in Table IV.

The complexity of modifies clauses is a general irritation in program logic, and one might feel that this problem with modifies clauses could be easily avoided, simply

by doing away with assignment to variables, so that the heap component is the only part of the state that changes. While this is easy to do semantically, obtaining a satisfactory program logic is not as straightforward. The most important point is the treatment of auxiliary variables. For example, in the queue module the variable Q is used in interface specifications as well as the invariant. If we were to try to place this variable into the heap then separation would not allow us to have it in both an interface specification and an invariant. It is important that Q can appear in client assertions, but cannot be altered by client code, and the detailed syntactic conditions on variables are designed to allow this. It has been suggested that fractional permissions, a method of sharing read access to heap cells, can also deal with these uses of auxiliary variables [Bornat et al. 2005; Parkinson et al. 2006]; while an alluring suggestion, further work is needed to understand the connection between permissions and auxiliary variables.

10.1 On Existentials and Free Variables

In [O'Hearn et al. 2001; Reynolds 2002] there is an inference rule for introducing existential variables in preconditions and postconditions.

$$\frac{\{p\}C\{q\}}{\{\exists x.p\}C\{\exists x.q\}}\ x \notin \mathsf{free}(C)$$

The side condition cannot be stated in the formalism of this paper. For, a procedure specification $\{p\}k\{q\}[X]$ identifies the variables, X, that k might modify, but not those that k might read from.

We can get around this problem by adding a free variable component to the sequent form, thus having

$$(Y)\ \Gamma \vdash \{p\}C\{q\}.$$

This constrains the variables appearing in C and all the procedures k_i, but not the preconditions and postconditions. This would allow us to describe the existential rule as

$$\frac{(Y)\ \Gamma \vdash \{p\}C\{q\}}{(Y)\ \Gamma \vdash \{\exists x.p\}C\{\exists x.q\}}\ x \notin Y$$

Another reasonable approach is to have a distinct class of "logical" variables, that cannot be assigned to in programs. For technical simplicity, we do not explicitly pursue either of these extensions in the current paper.

11. SOUNDNESS OF THE HYPOTHETICAL FRAME RULE

From now on we assume the standard notion of validity of sequents, as in Definition 4. Our task now is to prove the main result:

THEOREM 8. [Soundness Theorem]

(a) The hypothetical frame rule is sound for fixed preconditions $p_1, ..., p_n$ if and only if $p_1, ..., p_n$ are all precise.

(b) The hypothetical frame rule is sound for a fixed invariant r if and only if r is precise.

This result covers the queue and memory manager examples, where the preconditions and invariants are all precise.

The remainder of this section contains the proof of this theorem, aided by several preparatory steps which help to simplify what has to be proven.

—**Rule decomposition**. We decompose the hypothetical frame rule into two simpler rules, one of which involves the addition of modifies sets without adding invariants, and the other of which involves the addition of invariants without extending modifies sets.

—**The greatest relation**. We identify the greatest relation for a specification $\{p\}k\{q\}[X]$, which is the greatest local relation satisfying it [Schwarz 1974; 1977; Morgan 1988]. This allows us to reduce the truth of a sequent, which officially involves quantification over all environments, to the truth of a single triple for a single environment.

—**Simulation**. To show the soundness of the hypothetical frame rule we need to connect the meaning of a command in one context to its meaning in another with an additional invariant and additional modifies sets. We develop a notion of simulation relation between commands to describe this connection.

11.1 Rule Decomposition

Our first simplification decomposes the hypothetical rule into two simpler rules.

Modifies Weakening

$$\frac{\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\Gamma, \{p_1\}k_1\{q_1\}[X_1, Y], \ldots, \{p_n\}k_n\{q_n\}[X_n, Y] \vdash \{p\}C\{q\}}$$

where Y is disjoint from "$\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}$"

Simple Hypothetical Frame Rule

$$\frac{\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\Gamma, \{p_1 * r\}k_1\{q_1 * r\}[X_1], \ldots, \{p_n * r\}k_n\{q_n * r\}[X_n] \vdash \{p * r\}C\{q * r\}}$$

where $\mathit{Modifies}(C)(\Gamma)$ is disjoint from r and for all $\{p'\}k\{q'\}[X]$ in Γ, the modifies set X is disjoint from r

Note that these rules are specific instances of the hypothetical frame rule: the first rule is obtained by taking the set $\{(s, []) \mid s \in S\}$ for a resource invariant, and the second rule by taking the empty set for Y. In fact, the hypothetical frame rule is equivalent to these rules because it is derivable from them:

$$\dfrac{\dfrac{\dfrac{\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\Gamma_0, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}}{\Gamma_0, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}}{\dfrac{\Gamma_0, \{p_1 * r\}k_1\{q_1 * r\}[X_1, Y], \ldots, \{p_n * r\}k_n\{q_n * r\}[X_n, Y] \vdash \{p\}C\{q\}}{\Gamma, \{p_1 * r\}k_1\{q_1 * r\}[X_1, Y], \ldots, \{p_n * r\}k_n\{q_n * r\}[X_n, Y] \vdash \{p * r\}C\{q * r\}}}$$

Here Γ_0 is a subcontext of Γ containing specifications of only those procedures that appear in C. The derivation first uses the rule for shrinking contexts, and removes specifications for uncalled procedures from Γ. Next, it applies the modifies

weakening and the simple hypothetical frame rule, and adds first Y and then r. Finally, the derivation restores Γ by extending Γ_0. In the derivation, we shrink and extend Γ, in order to ensure that the side condition of the hypothetical frame rule implies that of the simple hypothetical rule.

11.2 Greatest Relation

For a specification $\{p\}-\{q\}[X]$, we consider the greatest local relation $\mathsf{great}(p,q,X)$ in LRel satisfying the (semantic) triple $\{p\}\mathsf{great}(p,q,X)\{q\}$ and the modifies clause $\mathsf{modifies}(\mathsf{great}(p,q,X),X)$. The relation $\mathsf{great}(p,q,X)$ exists for all specifications $\{p\}-\{q\}[X]$, and can be defined as follows:

$(s,h)[\mathsf{great}(p,q,X)]\mathtt{fault} \stackrel{def}{\iff} (s,h) \notin p * \mathtt{true}$

$(s,h)[\mathsf{great}(p,q,X)](s',h') \stackrel{def}{\iff}$
(1) $s(y) = s'(y)$ for all variables $y \notin X$; and
(2) $\forall h_p, h_1.\, (h_p * h_1 = h \wedge (s,h_p) \in p) \implies (\exists h'_q.\, h'_q \# h_1 \wedge h'_q * h_1 = h' \wedge (s',h'_q) \in q)$

The first equivalence says that the relation $\mathsf{great}(p,q,X)$ is safe at (s,h) just when p holds in (s,h_p) for some subheap h_p of h. Note that this equivalence implies the safety monotonicity of $\mathsf{great}(p,q,X)$. The second equivalence is about state changes. The first condition in this equivalence means that $\mathsf{great}(p,q,X)$ can modify only those variables in X, and the second condition that the output (s',h') is produced by a deallocation of a subheap of h in p followed by an allocation of a new heap in q: $\mathsf{great}(p,q,X)$ demonically chooses a subheap h_p of the initial heap h that satisfies p (i.e., $(s,h_p) \in p$), and disposes all cells in h_p; then, it angelically picks from q a new heap h'_q (i.e., $(s',h'_q) \in q$) and allocates h'_q to get the final heap h'.

LEMMA 9. *The relation* $\mathsf{great}(p,q,X)$ *is in* LRel.

Proof: We only show that $\mathsf{great}(p,q,X)$ satisfies the frame property. Consider states $(s,h),(s',h')$ and a heap h_0 such that $\mathsf{great}(p,q,X)$ is safe at state (s,h), $h_0 \# h$, and $(s,h*h_0)[\mathsf{great}(p,q,X)](s',h')$. Since $\mathsf{great}(p,q,X)$ is safe at (s,h), there is a subheap $h_p \leq h$ such that $(s,h_p) \in p$. Then, since $(s,h*h_0)[\mathsf{great}(p,q,X)](s',h')$, relation $\mathsf{great}(p,q,X)$ preserves heap $(h-h_p)*h_0$, so the final heap h' is $(h-h_p)*h_0*h'_q$ for some h'_q with $(s',h'_q) \in q$. Now, we claim that $(s,h)[\mathsf{great}(p,q,X)](s',(h-h_p)*h'_q)$ holds. Since s and s' differ only for some variables in X, we will just show that for all splittings $m_p * m = h$ of h, if (s,m_p) is in p, there is a subheap m'_q of $(h-h_p)*h'_q$ such that

$$m'_q \# m, \quad h'_q * (h-h_p) = m'_q * m, \quad \text{and} \quad (s',m'_q) \in q.$$

We use $(s, m_p * m * h_0)[\mathsf{great}(p,q,X)](s',h')$ to obtain such subheap m'_q of h'. Because $(s, m_p * m * h_0)[\mathsf{great}(p,q,X)](s',h')$, there exists a heap m'_q such that

$$m'_q \#(m * h_0), \quad h' = m'_q * m * h_0, \quad \text{and} \quad (s',m'_q) \in q.$$

Since $h'_q * (h-h_p) * h_0 = h' = m'_q * m * h_0$, we have $h'_q * (h-h_p) = m'_q * m$. Thus, this m'_q is the required subheap. □

LEMMA 10 (Greatestness). *The greatest relation* $\mathsf{great}(p,q,X)$ *satisfies* $\{p\}-\{q\}$ *and* $\mathsf{modifies}(-,X)$*, and is the greatest such: for all local relations* c*, we have*

$$\{p\}c\{q\} \wedge \mathsf{modifies}(c,X) \implies c \subseteq \mathsf{great}(p,q,X).$$

Proof: It is straightforward to see, from the definition, that $\mathsf{great}(p, q, X)$ satisfies both $\{p\}-\{q\}$ and $\mathsf{modifies}(-, X)$. To see that $\mathsf{great}(p, q, X)$ is indeed greatest, let's consider a relation c in LRel with $\{p\}c\{q\}$ and $\mathsf{modifies}(c, X)$. When $(s, h)[c]\mathtt{fault}$ holds, no subheap h_0 of h is in p (i.e., $(s, h_0) \notin p$): if (s, h_p) were in p for some subheap h_p of h, then c is safe at (s, h_p) because $\{p\}c\{q\}$ holds; thus, c is also safe at (s, h) by the safety monotonicity, and this gives the required contradiction. Therefore, if $(s, h)[c]\mathtt{fault}$, then $(s, h)[\mathsf{great}(p, q, X)]\mathtt{fault}$. Now, consider states $(s, h), (s', h')$ such that $(s, h)[c](s', h')$. We need to show the two conditions for the state change of $\mathsf{great}(p, q, X)$ hold for the states (s, h) and (s', h'). The condition for the stack holds: because of $\mathsf{modifies}(c, X)$, the stacks s and s' differ only for some variables in X. For the condition for the heap, consider a splitting $h_p * h_0 = h$ of h such that (s, h_p) is in p. Since $\{p\}c\{q\}$ holds, c is safe at (s, h_p). Therefore, the frame property of c implies that there is a subheap h'_q of h' such that $h'_q * h_0 = h'$ and $(s, h_p)[c](s', h'_q)$. Now, since $\{p\}c\{q\}$ holds and the initial state (s, h_p) is in p, the final state (s', h'_q) must be in q. Therefore, (s, h) and (s', h') satisfy the condition for the heap change of $\mathsf{great}(p, q, X)$. □

The *greatest* environment for a context Γ is the greatest environment satisfying all the procedure specifications in Γ. It maps a procedure identifier k to $\mathsf{great}(p, q, X)$ when the context Γ has the specification $\{p\}k\{q\}[X]$; otherwise, it maps k to the greatest relation in LRel, which is $\mathsf{States} \times (\mathsf{States} \cup \{\mathtt{fault}\})$. The greatest environment for Γ is well-defined because of Lemma 9, and it is the greatest environment satisfying Γ because of Lemma 10.

We use the greatest environments to interpret a sequent $\Gamma \vdash \{p\}C\{q\}$ and a proof rule in a simpler way. In this interpretation, a sequent $\Gamma \vdash \{p\}C\{q\}$ just means that $\{p\}([\![C]\!]\eta)\{q\}$ holds for the greatest environment η satisfying Γ. The new interpretation is implied by the old one because it considers just a single environment satisfying Γ; in fact, it is equivalent to the old interpretation, because $[\![C]\!]$ maps a greater environment to a greater relation but a greater relation satisfies fewer triples.

LEMMA 11. *A sequent $\Gamma \vdash \{p\}C\{q\}$ holds if and only if the triple $\{p\}([\![C]\!]\eta)\{q\}$ holds for the greatest environment η satisfying Γ.*

PROPOSITION 12. *For all predicates p, q, p' and q', commands C, and contexts Γ and Γ', we have the following equivalence: the proof rule*

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma' \vdash \{p'\}C\{q'\}}$$

holds if and only if we have

$$\{p\}([\![C]\!]\eta)\{q\} \Longrightarrow \{p'\}([\![C]\!]\eta')\{q'\}$$

for the greatest environments η and η' that, respectively, satisfy Γ and Γ'.

11.3 Simulation

Let $R : \mathsf{States} \leftrightarrow \mathsf{States}$ be a binary relation between states. For c, c' in LRel, we say that c *simulates* c' upto R, denoted $c[\mathsf{sim}(R)]c'$, if and only if the following two properties hold:

—Generalized Safety Monotonicity: if c is safe at (s,h), and $(s,h)[R](s',h')$, then c' is safe at (s',h').

—Generalized Frame Property: if c is safe at (s,h) and we have that $(s,h)[R](s',h')$ and $(s',h')[c'](s'_1,h'_1)$, then there is a state (s_1,h_1) such that $(s,h)[c](s_1,h_1)$ and $(s_1,h_1)[R](s'_1,h'_1)$.

Intuitively, $c[\mathsf{sim}(R)]c'$ says that for R-related initial states (s,h) and (s',h'), when we have enough resources at (s,h) to run c safely, we also have enough resources at (s',h') to run c' safely; in that case, every computation $(s',h')(s'_1,h'_1)\ldots(s'_n,h'_n)\ldots$ of c' from (s',h') can be simulated by some computation $(s,h)(s_1,h_1)\ldots(s_n,h_n)\ldots$ of c with $(s_i,h_i)[R](s'_i,h'_i)$.

We can give an alternate characterization of $c[\mathsf{sim}(R)]c'$ using Hoare triples for c and c'. For each predicate p, let $R(p)$ be the image of p by R, that is, the predicate $\{(s',h') \mid \exists(s,h) \in p.\, (s,h)[R](s',h')\}$.

LEMMA 13. *Local relations c and c' are related by $\mathsf{sim}(R)$ if and only if for all predicates p, q, we have*

$$\{p\}c\{q\} \Longrightarrow \{R(p)\}c'\{R(q)\}.$$

Proof: We show the only-if direction first. Suppose that a triple $\{p\}c\{q\}$ holds. Pick a state (s',h') from the predicate $R(p)$. We need to show that the command c' is safe at this state (s',h'), and that if c' can produce a state (s'_1,h'_1) when run in (s',h') (that is, $(s',h')[c'](s'_1,h'_1)$), this "final" state (s'_1,h'_1) is in $R(q)$. Since (s',h') is in $R(p)$, there is a state (s,h) in p that is related to (s',h') by R. Then c is safe at this state (s,h) because the triple $\{p\}c\{q\}$ holds. Now, generalized safety monotonicity implies that c' is also safe at (s',h'). Consider a state (s'_1,h'_1) that is one of the possible final states of c' from (s',h') (that is, $(s',h')[c'](s'_1,h'_1)$). Then, the generalized frame property says that there is a state (s_1,h_1) such that

$$(s,h)[c](s_1,h_1) \quad \text{and} \quad (s_1,h_1)[R](s'_1,h'_1).$$

It suffices to show that (s_1,h_1) is in q, because, then, the state (s'_1,h'_1) is in $R(q)$. We note that c satisfies the triple $\{p\}c\{q\}$, and that (s,h) is in the precondition p of this triple. Thus, (s_1,h_1), which is one of the possible final states of c from (s,h), is in q.

For the other direction, consider states (s,h) and (s',h') such that c is safe at (s,h), and the states (s,h) and (s',h') are related by R. Then, c satisfies the triple

$$\{\{(s,h)\}\}c\{\{(s_1,h_1) \mid (s,h)[c](s_1,h_1)\}\}.$$

From this triple, we obtain a triple $\{p'\}c'\{q'\}$ for c' by assumption where p' and q' are defined as follows:

$$(s_0,h_0) \in p' \overset{def}{\iff} (s,h)[R](s_0,h_0)$$
$$(s_0,h_0) \in q' \overset{def}{\iff} \exists(s_1,h_1).\, (s,h)[c](s_1,h_1) \wedge (s_1,h_1)[R](s'_1,h'_1).$$

Since c' satisfies the triple $\{p'\}c'\{q'\}$ and the state (s',h') is in the precondition p' of this triple, c' is safe at (s',h'). For the generalized frame property, consider a "final" state (s'_1,h'_1) of c' from (s',h') (that is, $(s',h')[c'](s'_1,h'_1)$). This final state (s'_1,h'_1) is in q' because $\{p'\}c'\{q'\}$ holds and (s',h') is in p'. Now, the definition

of the postcondition q' gives the required state: it says that there exists a state (s_1, h_1) such that $(s,h)[c](s_1,h_1)$ and $(s_1,h_1)[R](s_1',h_1')$. $\square$

The simulation relation for environments, also denoted $\mathsf{sim}(R)$, is defined pointwise: two environments η, η' are related by $\mathsf{sim}(R)$ if and only if for all procedure identifiers k, we have $\eta(k)[\mathsf{sim}(R)]\eta'(k)$.

We use the simulation for environments to further simplify the meaning of a proof rule, building on Proposition 12. Let R be a relation between states. We say that a command C is independent of R if and only if every basic command A in C simulates itself upto R, and every boolean expression B in C maps R-related states to the same value:

$$\overline{A}[\mathsf{sim}(R)]\overline{A} \ \wedge \ \Big(\forall (s,h),(s',h').\ (s,h)[R](s',h') \Longrightarrow [\![B]\!]s = [\![B]\!]s'\Big).$$

PROPOSITION 14. *Let Γ and Γ' be contexts, and let η and η' be the greatest environments satisfying Γ and Γ', respectively. Consider a set $\mathcal{P}$ of commands such that all commands in $\mathcal{P}$ are independent of R. Then, the following proof rule holds for all predicates p, q and all commands C in $\mathcal{P}$*

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma' \vdash \{R(p)\}C\{R(q)\}}$$

if we have $\eta[\mathsf{sim}(R)]\eta'$. Moreover, the converse holds if $\mathcal{P}$ contains the procedure call k for every procedure identifier k.

We note two special cases of this proposition, which are related to the rule of modifies weakening and the simple hypothetical frame rule, respectively. The first case is when the proposition is instantiated with a relation R_Y for a set Y of variables. The relation R_Y relates two states when the states differ only for variables in Y:

$$(s,h)[R_Y](s',h') \stackrel{def}{\Longleftrightarrow} (h = h' \wedge \forall x \in \mathsf{Variables}.\, x \notin Y \Rightarrow s(x) = s'(x)).$$

Note that the image $R_Y(p)$ of a predicate p is just p if the set Y is disjoint from p; and a command C is independent of R_Y if Y is disjoint from the command C. For this relation R_Y, the proposition implies the following: for all contexts Γ and Γ', if the greatest environments for Γ and Γ' are related by $\mathsf{sim}(R_Y)$, then the proof rule

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma' \vdash \{p\}C\{q\}}$$

holds for all predicates p, q and commands C such that Y is disjoint from p, q, and C. Note that the rule of modifies weakening is a special case of the above proof rule.

The second case instantiates the proposition with a relation R_r for a predicate r. The relation R_r relates states (s,h) and (s',h') when we can obtain (s',h') from (s,h) by allocating a new heap in r:

$$(s,h)[R_r](s',h') \stackrel{def}{\Longleftrightarrow} (s = s' \wedge \exists h_1 \in \mathsf{Heaps}.\, h_1 \# h \wedge h_1 * h = h' \wedge (s,h_1) \in r).$$

For all predicates p, the image $R_r(p)$ of a predicate p is $p * r$. Thus, for relation R_r, the proposition says the following: let Γ and Γ' be contexts, and $\mathcal{P}$ a set

of commands such that all the commands in $\mathcal{P}$ are independent of R_r, and all procedure calls k are in $\mathcal{P}$; then, the proof rule

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma' \vdash \{p * r\}C\{q * r\}}$$

holds for all predicates p and q and commands C in $\mathcal{P}$ if and only if the greatest environments for Γ and Γ' are related by $\mathsf{sim}(R)$. We use this instantiation to handle the simple hypothetical frame rule.

We prove Proposition 14 using Proposition 12, Lemma 13, and an additional lemma which says that a command preserves the simulation. Consider a relation R between states, and contexts Γ and Γ' for procedure identifiers. Let η and η' be the greatest environments for the contexts Γ and Γ', respectively. We first simplify the meaning of a proof rule as follows:

$$\begin{array}{lll}
\forall C \in \mathcal{P}, p, q.\ \big(\Gamma \vdash \{p\}C\{q\} \Longrightarrow \Gamma' \vdash \{R(p)\}C\{R(q)\}\big) & & \\
\iff & \forall C \in \mathcal{P}, p, q.\ \big(\{p\}[\![C]\!]\eta\{q\} \Longrightarrow \{R(p)\}[\![C]\!]\eta'\{R(q)\}\big) & (\because \text{Proposition 12}) \\
\iff & \forall C \in \mathcal{P}.\ [\![C]\!]\eta[\mathsf{sim}(R)][\![C]\!]\eta' & (\because \text{Lemma 13}).
\end{array}$$

When the set $\mathcal{P}$ contains k for all procedure identifiers k, this simplified meaning implies that the environments η and η' are related by $\mathsf{sim}(R)$. So, the only-if direction of Proposition 14 holds. For the other direction, we use a lemma which says that C maps $\mathsf{sim}(R)$-related environments to $\mathsf{sim}(R)$-related local relations.

LEMMA 15. *For all commands C and state relations R, if C is independent of R, then we have*

$$\forall \eta, \eta'.\ \eta[\mathsf{sim}(R)]\eta' \Longrightarrow ([\![C]\!]\eta)[\mathsf{sim}(R)]([\![C]\!]\eta').$$

Since all commands in $\mathcal{P}$ are independent of R, this lemma says, $\eta[\mathsf{sim}(R)]\eta'$ is enough to ensure $[\![C]\!]\eta[\mathsf{sim}(R)][\![C]\!]\eta'$ for all commands C in $\mathcal{P}$. This shows the if direction of Proposition 14.

Proof: [Lemma 15] We prove the lemma by induction on the structure of C. Consider the case that C is a basic command A. Since C is independent of R, the relation $\overline{A}$ is related to itself by $\mathsf{sim}(R)$.

For the remaining cases, we consider states (s, h) and (s', h') such that they are related by R, and the relation $[\![C]\!]\eta$ is safe at (s, h). In each case, we show that

(1) the relation $[\![C]\!]\eta'$ is safe at (s', h'); and
(2) for all states (s'_1, h'_1), if $(s', h')[[\![C]\!]\eta'](s'_1, h'_1)$, then there is a state (s_1, h_1) such that

$$(s, h)[[\![C]\!]\eta](s_1, h_1) \quad \text{and} \quad (s_1, h_1)[R](s'_1, h'_1).$$

Consider the case of $C = C_1; C_2$. The safety of $[\![C_1; C_2]\!]\eta$ at (s, h) implies that $[\![C_1]\!]\eta$ is safe at the same state. We show that $[\![C_1; C_2]\!]\eta'$ is safe at (s', h') by contradiction. Suppose that $[\![C_1; C_2]\!]\eta'$ faults when run in (s', h'). Then, either $[\![C_1]\!]\eta'$ is not safe at (s', h'), or the state (s', h') is related to some state (s'_1, h'_1) by $[\![C_1]\!]\eta'$ but $[\![C_2]\!]\eta'$ is not safe at this state (s'_1, h'_1). The first case is impossible: the induction hypothesis for C_1 implies that $[\![C_1]\!]\eta'$ must be safe at (s', h'), because $[\![C_1]\!]\eta$ is safe at (s, h). The second case is not possible, either. The induction

hypothesis says that there exists a state (s_1, h_1) satisfying $(s, h)[[\![C_1]\!]\eta](s_1, h_1)$ and $(s_1, h_1)[R](s'_1, h'_1)$. Then, $[\![C_2]\!]\eta$ is safe at (s_1, h_1), because $[\![C_1; C_2]\!]\eta$ is safe at (s, h) and $[\![C_1]\!]\eta$ can produce (s_1, h_1) when run in (s, h) (that is, $(s, h)[[\![C_1]\!]\eta](s_1, h_1)$). Now, the induction hypothesis for C_2 implies that $[\![C_2]\!]\eta'$ must be safe at (s'_1, h'_1).

For the second requirement for $\mathsf{sim}(R)$, consider states $(s'_1, h'_1), (s'_2, h'_2)$ for which we have

$$(s', h')[[\![C_1]\!]\eta'](s'_1, h'_1) \text{ and } (s'_1, h'_1)[[\![C_2]\!]\eta'](s'_2, h'_2).$$

Because of the induction hypothesis for C_1, there exists a state (s_1, h_1) such that $(s, h)[[\![C_1]\!]\eta](s_1, h_1)$ and $(s_1, h_1)[R](s'_1, h'_1)$. Now, the induction hypothesis for C_2 says that there exists a state (s_2, h_2) for which we have $(s_1, h_1)[[\![C_2]\!]\eta](s_2, h_2)$ and $(s_2, h_2)[R](s'_2, h'_2)$. This state (s_2, h_2) is the required one for the second requirement for $\mathsf{sim}(R)$.

When C is $\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2$ or a procedure call k, it is straightforward to show that the two requirements are satisfied. The case of the conditional statement directly follows from the induction hypothesis for C_1 and C_2, because the independence of C with R ensures that $[\![B]\!]\eta$ and $[\![B]\!]\eta'$ map R-related states to the same values. The case of the procedure call follows from $\eta[\mathsf{sim}(R)]\eta'$.

For the remaining cases of loop and procedure definition, it suffices to show that $\mathsf{sim}(R)$ is a complete relation.

(1) $\bot[\mathsf{sim}(R)]\bot$ for the empty local relation $\bot$, and
(2) if $c_i[\mathsf{sim}(R)]c'_i$ for all $i \in I$, we have

$$\bigcup_{i\in I} c_i[\mathsf{sim}(R)]\bigcup_{i\in I} c'_i.$$

The first is straightforward since $\bot$ is safe at all states and it does not relate any states. For the second, consider two families of commands, $\{c_i\}_{i\in I}$ and $\{c'_i\}_{i\in I}$, such that $c_i[\mathsf{sim}(R)]c'_i$. When $\bigcup_{i\in I} c_i$ is safe at a state (s, h), all c_i are safe at (s, h). For all states (s', h'), if (s, h) and (s', h') are related by R, all c'_i are safe at state (s', h') because $c_i[\mathsf{sim}(R)]c'_i$. Therefore, $\bigcup_{i\in I} c'_i$ is also at (s', h'). To complete the proof, we only need to prove the generalized frame property: for all states (s, h), (s', h'), and (s'_1, h'_1), if $\bigcup_{i\in I} c_i$ is safe at state (s, h), and we have $(s, h)[R](s', h')$ and $(s', h')[\bigcup_{i\in I} c'_i](s'_1, h'_1)$, then there exists a state (s_1, h_1) such that $(s, h)[\bigcup_{i\in I} c_i](s_1, h_1)$ and $(s_1, h_1)[R](s'_1, h'_1)$. Such a state (s_1, h_1) can be obtained as follows. Since $(s', h')[\bigcup_{i\in I} c'_i](s'_1, h'_1)$, there is some i in I such that c'_i produces (s'_1, h'_1) when run in (s', h') (that is, $(s', h')[c'_i](s'_1, h'_1)$). Moreover, the corresponding c_i is safe at (s, h), because $\bigcup_{i\in I} c_i$ is safe at (s, h). Now, we use $c_i[\mathsf{sim}(R)]c'_i$ and $(s, h)[R](s', h')$, and get a state (s_1, h_1) such that $(s, h)[c_i](s_1, h_1)$ and $(s_1, h_1)[R](s'_1, h'_1)$. This state is the required one because $(s, h)[c_i](s_1, h_1)$ entails $(s, h)[\bigcup_{i\in I} c_i](s_1, h_1)$. □

11.4 Soundness of Modifies Weakening

We use a particular simulation relation to prove that

PROPOSITION 16. *The rule of modifies weakening is sound.*

Suppose that we apply the rule to extend the modifies clauses by a set Y of variables. To simplify the rule in this case, we instantiate Proposition 14 with a relation

$R_Y : \mathsf{States} \leftrightarrow \mathsf{States}$, contexts

$$\text{“}\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n]\text{”}$$
$$\text{and “}\Gamma, \{p_1\}k_1\{q_1\}[X_1, Y], \ldots, \{p_n\}k_n\{q_n\}[X_n, Y]\text{”},$$

and a set $\mathcal{P}_Y$ of commands. The relation R_Y relates two states when they differ at most for variables in Y

$$(s,h)R_Y(s_1,h_1) \stackrel{def}{\Longleftrightarrow} h = h_1 \wedge (\forall x \in \mathsf{Variables}.\, x \notin Y \Rightarrow s(x) = s_1(x)),$$

and the set $\mathcal{P}_Y$ consists of commands C such that Y is disjoint from C. The set $\mathcal{P}_Y$ satisfies the side condition of Proposition 14 for the if direction, because $\mathcal{P}_Y$ contains only those commands from which Y is disjoint, and all such commands are independent of R_Y. Proposition 14 implies that the following weakening rule holds for all commands C in $\mathcal{P}_Y$, and all predicates p, q such that Y is disjoint from p, q

$$\frac{\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\Gamma, \{p_1\}k_1\{q_1\}[X_1, Y], \ldots, \{p_n\}k_n\{q_n\}[X_n, Y] \vdash \{p\}C\{q\}}$$

if for each $\{p\}k\{q\}[X]$ in Γ, we have $\mathsf{great}(p,q,X)[\mathsf{sim}(R_Y)]\mathsf{great}(p,q,X)$, and for all i, we have $\mathsf{great}(p_i,q_i,X_i)[\mathsf{sim}(R_Y)]\mathsf{great}(p_i,q_i,X_i \cup Y)$. Here, we write $\{p\}C\{q\}$ in the conclusion instead of $\{R(p)\}C\{R(q)\}$, because if Y is disjoint from a predicate p', the image $R(p')$ is the same as p'. Also, we ignore procedure identifiers that do not appear in antecedents of the sequents, because the greatest environments map such procedure identifiers to the greatest local relation $\mathsf{States} \times (\mathsf{States} \cup \{\mathtt{fault}\})$ in LRel, and this local relation is related to itself by $\mathsf{sim}(R)$ for all R. We show that this sufficient condition is implied by the side condition of modifies weakening.

LEMMA 17. *Let Y be a set of variables. For all predicates p, q and sets X of variables, if Y is disjoint from all of p, q and X, then for all subsets Y_0 of Y, we have*

$$\mathsf{great}(p,q,X)[\mathsf{sim}(R_Y)]\mathsf{great}(p,q,X \cup Y_0).$$

Proof: Consider states (s,h) and (s_1,h_1) such that they are related by R_Y and $\mathsf{great}(p,q,X)$ is safe at (s,h). We first show that $\mathsf{great}(p,q,X \cup Y_0)$ is safe at (s_1,h_1). Since $\mathsf{great}(p,q,X)$ is safe at (s,h), heap h has a subheap h_p such that (s,h_p) is in p. This subheap h_p is also a subheap of h_1, because $(s,h)[R_Y](s_1,h_1)$ implies that h and h_1 are the same. Moreover, state (s_1,h_p) is in p. The reason is that, because $(s,h)[R_Y](s_1,h_1)$, the stacks s and s_1 differ only for some variables in Y, and Y is disjoint from p. Thus, $\mathsf{great}(p,q,X \cup Y_0)$ is safe at (s_1,h_1).

For the generalized frame property, suppose that $\mathsf{great}(p,q,X \cup Y_0)$ produces a state (s_1',h_1') when run in (s_1,h_1). Let s' be the stack that is the same as s_1' except that s' stores $s(y)$ for each variable y in Y. Then, we have $(s',h_1')[R_Y](s_1',h_1')$. We will show that (s,h) and (s',h_1') are related by $\mathsf{great}(p,q,X)$. The definition of $\mathsf{great}(p,q,X)$ requires that s and s' differ only for some variables in X. This requirement is satisfied because for every variable $y \in Y$, we have $s(y) = s'(y)$ by definition; and for every variable $z \notin X \cup Y$, we have the following:

$$\begin{array}{rcll} s(z) & = & s_1(z) & (\because (s,h)[R_Y](s_1,h_1)) \\ & = & s_1'(z) & (\because (s_1,h_1)[\mathsf{great}(p,q,X \cup Y_0)](s_1',h_1')) \\ & = & s'(z). & \end{array}$$

For the other requirement of $\mathsf{great}(p, q, X)$, consider a splitting $m_p * m = h$ of h such that (s, m_p) is in p. Since Y is disjoint from p, state (s_1, m_p) is also in p; therefore, $(s_1, h_1)[\mathsf{great}(p, q, X \cup Y_0)](s_1', h_1')$ gives a splitting $m_q' * m' = h_1'$ of h_1' such that $(s_1', m_q') \in q$ and $m' = m$. Since Y is disjoint from q, state (s', m_q') is also in q; thus, the splitting $m_q' * m' = h_1'$ is the required one for $(s, h)[\mathsf{great}(p, q, X)](s', h_1')$. □

11.5 Soundness of the Simple Hypothetical Frame Rule

We again use particular simulation relations, this time to prove results about the simple hypothetical frame rule.

PROPOSITION 18.

(a) The simple hypothetical frame rule is sound for fixed preconditions $p_1, ..., p_n$ if and only if $p_1, ..., p_n$ are all precise.

(b) The simple hypothetical frame rule is sound for a fixed invariant r if and only if r is precise.

Theorem 8 follows from this proposition, and the soundness of modifies weakening. The remainder of the section is devoted to proving the proposition.

Consider a predicate r, and a context "$\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n]$", such that all the modifies clauses in Γ are disjoint from r (i.e., for every $\{p'\}k'\{q'\}[X']$ in Γ, the set X' is disjoint from r). We will use these predicate and context for a resource invariant and procedure specifications, respectively. To prove the propositions we instantiate Proposition 14 with the following relation R_r: $\mathsf{States} \leftrightarrow \mathsf{States}$, contexts Γ_1 and Γ_2, and set $\mathcal{P}_r$ of commands:

$$\begin{aligned}
(s, h)[R_r](s_1, h_1) &\stackrel{def}{\iff} (s = s_1) \wedge (\exists h_r.\ h_1 = h * h_r \wedge (s, h_r) \in r) \\
\Gamma_1 &\stackrel{def}{=} \Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \\
\Gamma_2 &\stackrel{def}{=} \Gamma, \{p_1 * r\}k_1\{q_1 * r\}[X_1], \ldots, \{p_n * r\}k_n\{q_n * r\}[X_n] \\
C \in \mathcal{P}_r &\stackrel{def}{\iff} \mathit{Modifies}(C)(\Gamma) \text{ is disjoint from } r
\end{aligned}$$

Intuitively, R_r relates states (s, h) and (s_1, h_1) just when (s_1, h_1) can be obtained from (s, h) by adding a new heap in r; so, the image $R_r(p)$ of a predicate p is just $p * r$.

The set $\mathcal{P}_r$ satisfies the side condition of Proposition 14 for the if direction, because for each command C in $\mathcal{P}_r$, the set $\mathit{Modifies}(C)(\Gamma)$ is disjoint from r, and such command C is independent of R_r. To see why independence holds, suppose that the set $\mathit{Modifies}(C)(\Gamma)$ of a command C is disjoint from r. Every boolean expression depends only on the stack, and R_r-related states have the same stack, so C trivially satisfies the constraint that its boolean expression maps R_r-related states to the same value. Thus, for the independence of C with R_r, we only need to show that every basic command A in C is related to itself by $\mathsf{sim}(R_r)$. Note that since $\mathit{Modifies}(C)(\Gamma)$ is disjoint from r, each basic command A in C changes only those variables that are disjoint from r. The following lemma says that every such local relation is related to itself by $\mathsf{sim}(R_r)$.

LEMMA 19. *Let r be a predicate and c a local relation in* LRel. *If there is a set X of variables such that* $\mathsf{modifies}(c, X)$ *holds and X is disjoint from r, we have $c[\mathsf{sim}(R_r)]c$.*

Proof: Consider R_r-related states (s,h) and (s_1,h_1) such that c is safe at (s,h). By the definition of $(s,h)[R_r](s_1,h_1)$, stacks s and s_1 are the same and there exists a subheap h_r of h_1 such that

$$h_r \# h, \quad h_r * h = h_1, \quad \text{and} \quad (s_1, h_r) \in r.$$

For generalized safety monotonicity, we need to show that c is also safe at (s_1,h_1); the local relation c is safe at that state because c satisfies safety monotonicity and state (s_1,h_1) is an extension of (s,h) (that is, $(s_1,h_1) = (s, h * h_r)$). For the generalized frame property, consider a state (s_1',h_1') that can be produced by c when c is run in (s_1,h_1) (that is, $(s_1,h_1)[c](s_1',h_1')$). We must find a state (s',h') such that $(s,h)[c](s',h')$ and $(s',h')[R_r](s_1',h_1')$. Note that we have $(s,h*h_r)[c](s_1',h_1')$ since (s_1,h_1) and $(s,h*h_r)$ are the same. The frame property of c for $(s,h*h_r)[c](s_1',h_1')$, then, gives a heap m' such that

$$m' \# h_r, \quad m' * h_r = h_1', \quad \text{and} \quad (s,h)[c](s_1',m').$$

Moreover, since c only changes variables that are disjoint from r, and (s,h_r) is in r, the state (s_1',h_r) is also in r; so, $(s_1',m')[R_r](s_1',h_1')$ holds. This state (s_1',m') is the required (s',h'). □

The set $\mathcal{P}_r$ also satisfies the side condition for the only-if direction of Proposition 14, because all the modifies clauses in Γ are disjoint from r. The side condition requires that $\mathcal{P}_r$ should contain the call of every procedure k, and it can be proved by the case analysis on k as follows. If k appear in Γ, by the choice of r and Γ, the set $\mathit{Modifies}(k)(\Gamma)$ should be disjoint from r, which gives $k \in \mathcal{P}_r$. Otherwise, the set $\mathit{Modifies}(k)(\Gamma)$ is empty, so it is disjoint from r. This means that k belongs to $\mathcal{P}_r$, as required.

Thus, using Proposition 14, we have established the following for R_r, Γ_1, Γ_2, and $\mathcal{P}_r$:

LEMMA 20. *The following rule holds for all predicates p,q and all commands C in $\mathcal{P}_r$*

$$\frac{\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\Gamma, \{p_1 * r\}k_1\{q_1 * r\}[X_1], \ldots, \{p_n * r\}k_n\{q_n * r\}[X_n] \vdash \{p * r\}C\{q * r\}}$$

if and only if, for all i, we have $\mathsf{great}(p_i,q_i,X_i)[\mathsf{sim}(R_r)]\mathsf{great}(p_i * r, q_i * r, X_i)$ *and for each $\{p'\}k\{q'\}[X]$ in Γ, we have* $\mathsf{great}(p',q',X)[\mathsf{sim}(R_r)]\mathsf{great}(p',q',X)$.

We use the sufficient, and sometimes necessary, condition in the lemma to prove Propositions 18. The proof also uses the following properties, which will be shown below.

—If a set X of variables is disjoint from a predicate r, then for all predicates p,q, $\mathsf{great}(p,q,X)[\mathsf{sim}(R_r)]\mathsf{great}(p,q,X)$ (Proposition 21).

—A predicate p is precise if and only if $\mathsf{great}(p,q,X)[\mathsf{sim}(R_r)]\mathsf{great}(p*r, q*r, X)$ holds for all predicates q,r and all sets X of variables (Proposition 23(a)).

—A predicate r is precise if and only if $\mathsf{great}(p,q,X)[\mathsf{sim}(R_r)]\mathsf{great}(p*r, q*r, X)$ holds for all predicates p,q and all sets X of variables (Proposition 23(b)).

Postponing the proofs of these facts for a moment, we complete the main proof.

Proof: [of Proposition 18]

Proposition 21 establishes the side condition for each $\{p\}k\{q\}[X]$ in Γ in Lemma 20, and the only-if direction of Proposition 23(a) establishes that

$$\mathsf{great}(p_i, q_i, X_i)[\mathsf{sim}(R_r)]\mathsf{great}(p_i * r, q_i * r, X_i).$$

Thus, we may apply the first (sufficient) part of Lemma 20 to show the soundness (if direction) part of Proposition 18(a). Similarly, we can use Proposition 23(b) with Lemma 20 to show the if direction of Proposition 18(b).

We still need to show the only-if directions. Suppose that the simple hypothetical frame rule is sound for a fixed invariant r. We will show that for all predicates p_1, q_1 and all sets X_1 of variables, $\mathsf{great}(p_1, q_1, X_1)[\mathsf{sim}(R_r)]\mathsf{great}(p_1 * r, q_1 * r, X_1)$ holds; this implies that r is precise by the if direction of Proposition 23(b). Since the simple hypothetical frame rule is sound for r, the following holds for all predicates p_1, q_1, p, q, sets X_1 of variables, and commands C such that $\mathit{Modifies}(C)(\{\})$ is disjoint from r:

$$\frac{\{p_1\}k_1\{q_1\}[X_1] \vdash \{p\}C\{q\}}{\{p_1 * r\}k_1\{q_1 * r\}[X_1] \vdash \{p * r\}C\{q * r\}}$$

Here we take the empty context for Γ in the hypothetical rule. Thus, the side condition for the only-if direction of Proposition 14 holds for $\mathcal{P}_r$. Now, Proposition 14 gives the required

$$\forall p_1, q_1, X_1.\ \mathsf{great}(p_1, q_1, X_1)[\mathsf{sim}(R_r)]\mathsf{great}(p_1 * r, q_1 * r, X_1).$$

This establishes the only-if direction of Proposition 18(b).

The other case for fixed preconditions is proved similarly. Suppose that the simple hypothetical frame rule is sound for the fixed preconditions $p_1, \ldots, p_n$. Then, the following instance of the rule holds for all predicates $p, q, r, q_1, \ldots, q_n$, commands C and sets $X_1, \ldots, X_n$ of variables if $\mathit{Modifies}(C)(\{\})$ is disjoint from r:

$$\frac{\{p_1\}k_1\{q_1\}[X_1] \ldots \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}}{\{p_1 * r\}k_1\{q_1 * r\}[X_1] \ldots \{p_n * r\}k_n\{q_n * r\}[X_n] \vdash \{p * r\}C\{q * r\}}$$

Since $\mathcal{P}_r$ satisfies the side condition of Proposition 14 for the only-if direction, the proposition says that each precondition p_i satisfies

$$\forall q, r, X.\ \mathsf{great}(p_i, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p_i * r, q, X).$$

This property of each p_i implies that p_i is precise by Proposition 23(a). This establishes the only-if direction of Proposition 18(a). □

All that remains is to show the propositions whose proofs we postponed.

PROPOSITION 21. *For all predicates p, q and sets X of variables, if X is disjoint from r, we have*

$$\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p, q, X).$$

Proof: This proposition follows from Lemma 19 because the relation $\mathsf{great}(p, q, X)$ satisfies $\mathsf{modifies}(\mathsf{great}(p, q, X), X)$. □

We prove the remaining propositions using the following observations: generalized safety monotonicity for $\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p * r, q * r, X)$ always holds,

irrespective of whether p or r is precise, and both $\mathsf{great}(p, q, X)$ and $\mathsf{great}(p * r, q * r, X)$ modify only those variables in X.

The following lemma will be used in the proof.

LEMMA 22. *We have* $\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p*r, q*r, X)$ *if and only if for all states* (s, h) *and* (s'_1, h'_1)*, if*

1. $\mathsf{great}(p, q, X)$ *is safe at* (s, h)*, and*
2. *there is a heap* h_r *such that* $(s, h_r) \in r$ *and* $(s, h*h_r)[\mathsf{great}(p*r, q*r, X)](s'_1, h'_1)$*,*

then heap h'_1 *can be split into two heaps* h' *and* h'_r *(i.e.,* $h'_1 = h' * h'_r$*) such that*

3. (s'_1, h'_r) *is in* r*, and*
4. *whenever* h *is split into* m *and* m_p *satisfying* $(s, m_p) \in p$*, heap* h' *can also be split into the heap* m *and some* m'_q *satisfying* $(s'_1, m'_q) \in q$*.*

Proof: We focus on the if direction because the other direction follows straightforwardly from the definitions of $\mathsf{sim}(R_r)$ and $\mathsf{great}(p, q, X)$. Suppose that

(a) $\mathsf{great}(p, q, X)$ is safe at a state (s, h), and
(b) the state (s, h) is related to (s_1, h_1) by R_r.

Then

(c) s and s_1 are the same, and h_1 has a subheap h_r such that $h * h_r = h_1$ and $(s, h_r) \in r$.

For generalized safety monotonicity, it suffices to show that $\mathsf{great}(p * r, q * r, X)$ is safe at $(s, h * h_r)$; equivalently, $(s, h * h_r)$ is in $p * r * \mathtt{true}$. This holds because $\mathsf{great}(p, q, X)$ is safe at (s, h) and so, the state (s, h) is in $p * \mathtt{true}$.

For the generalized frame property, consider a state (s'_1, h'_1) such that

(d) $(s_1, h_1)[\mathsf{great}(p * r, q * r, X)](s'_1, h'_1)$.

We need to find a state (s', h') that satisfies

$$(s, h)[\mathsf{great}(p, q, X)](s', h') \text{ and } (s', h')[R_r](s'_1, h'_1).$$

The condition 1 in the if clause of the lemma holds for (s, h) by the assumption (a) above. The condition 2 holds for (s, h) and (s'_1, h'_1) by the property (c) and assumption (d). Thus, the if clause of the lemma gives a splitting $h' * h'_r = h'_1$ of h'_1 satisfying the two properties 3 and 4 in the statement of the lemma. We show that (s'_1, h') is the required state. Since (s'_1, h'_r) is in r, state (s'_1, h') is related to (s'_1, h'_1) by R_r. For $(s, h)[\mathsf{great}(p, q, X)](s'_1, h')$, the requirement (2) from the definition of great holds because it is precisely the property 4 of the splitting $h' * h'_r$ from the statement of the lemma; and the remaining requirement (1) from the definition of great holds since s and s'_1 can differ only for variables in X, because we know $(s, h * h_r)[\mathsf{great}(p * r, q * r, X)](s'_1, h'_1)$ from (d) and (c). □

PROPOSITION 23.

(a) A predicate p *is precise if and only if* $\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p*r, q*r, X)$ *holds for for all predicates* r *and* q*, and sets* X *of variables.*

(b) A predicate r is precise if and only if $\mathsf{great}(p,q,X)[\mathsf{sim}(R_r)]\mathsf{great}(p*r, q*r, X)$ *holds for all predicates p, q and sets X of variables.*

Proof: We first prove the only-if directions of (a) and (b) using Lemma 22; assuming p or r precise, we show that $\mathsf{great}(p,q,X)[\mathsf{sim}(R_r)]\mathsf{great}(p*r, q*r, X)$ holds

Suppose that properties 1 and 2 from the statement of Lemma 22 hold. We must find a splitting of h'_1 into h' and h'_r making properties 3 and 4 hold.

Since $\mathsf{great}(p,q,X)$ is safe at (s,h), state (s,h) is in $p * \mathtt{true}$. Thus, heap h can be partitioned into h_p, h_0 (i.e., $h = h_p * h_0$) such that $(s,h_p) \in p$. Since $(s, h*h_r)$ is related to (s'_1, h'_1) by $\mathsf{great}(p*r, q*r, X)$, heap h'_1 can be split into the heap h_0 and heaps h'_q, h'_r (i.e., $h'_1 = h_0 * h'_q * h'_r$) such that state (s'_1, h'_q) is in q and state (s'_1, h'_r) is in r. The heaps $h'_q * h_0$ and h'_r form the sought partitioning of h'_1 (taking h' to be $h'_q * h_0$). Property 3 is immediate. For property 4, we have two subcases.

(a) Assume p precise. We know that $h_p * h_0 = h$ is a splitting of h such that $(s,h_p) \in p$ is true, thus satisfying the antecedent of 4 (taking $m = h_0$, $m_p = h_p$). Furthermore, since p is precise this can be the only splitting satisfying the antecedent, so to satisfy property 4 it suffices to show it's consequent for this particular splitting. The consequent asks for a splitting $m * m'_q = h'$ for some m'_q where $(s'_1, m'_q) \in q$. Our candidate m'_q is the heap h'_q which we obtained above, and m has been chosen as h_0. We know that state (s'_1, h'_q) is in q, and we defined h' to be $h_0 * h'_q$, so we obtain the desired consequent just requested. This establishes property 4, and we finished the proof of the only if part of (a).

(b) Assume r precise. Consider a splitting $m_p * m = h$ of h such that $(s, m_p) \in p$. For property 4 we need to split $h'_q * h_0$ into the heap m and some m'_q that satisfies $(s'_1, m'_q) \in q$. Note that $m_p * h_r$ and m form a splitting of $h * h_r$, and that $(s, m_p * h_r)$ is in $p * r$. Therefore, we can use $(s, h*h_r)[\mathsf{great}(p*r, q*r, X)](s'_1, h'_1)$ to split h'_1 into the heap m and two heaps m'_q, m'_r (i.e., $h'_1 = m'_q * m'_r * m$) such that (s'_1, m'_q) is in q and (s'_1, m'_r) is in r. Since r is precise, and h'_r and m'_r are both subheaps of h'_1, and we know that $(s'_1, h'_1) \in r$, it follows that m'_r must be equal to h'_r; this and the identities $h'_1 = m'_q * m'_r * m$ and $h'_1 = h_0 * h'_q * h'_r$ then imply that $m'_q * m = h'_q * h_0$. These heaps m'_q and m thus give us the splitting of $h'_q * h_0$ that we were after. This completes the proof of the only if part of (b).

We show the if part of (a) by contradiction. Suppose that p is not precise. Then, there are states (s,h_1) and (s,h_2) in p whose heaps are different but consistent: for all l in $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2)$, the r-values $h_1(l)$ and $h_2(l)$ are the same. Let s, h_1, h_2 be such stack and heaps such that the cardinality of $\mathsf{dom}(h_1 \cup h_2)$ is the least. Let h be $h_1 \cap h_2$, and let h'_i be $h_i - h$ for $i = 1, 2$. Let q be the predicate $\{(s, [])\}$, and let r be the predicate defined by

$$r = \{(s, h'_1 * h * h'_2), (s, [])\} \cup \{(s,m) \mid \exists m_0 .\, m * m_0 = h'_1 * h * h'_2 \wedge (s,m) \in p\}.$$

We show that

(1) $(s, h'_1 * h * h'_2)\, [\mathsf{great}(p*r, q*r, X)]\, (s, h'_1 * h * h'_2)$ holds; but

(2) heap $h'_1 * h * h'_2$ can not be partitioned into two heaps m' and m'_r such that (s, m'_r) is in r and $(s, h'_1 * h * h'_2)[\mathsf{great}(p,q,X)](s, m')$ holds.

Then, since $(s, [])$ is in r and $\mathsf{great}(p, q, X)$ is safe at $(s, h'_1 * h * h'_2)$, these two facts contradict the assumption that $\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p * r, q * r, X)$. For the first fact, consider a splitting of $h'_1 * h * h'_2$ into heaps m_p, m_r, m such that state (s, m_p) is in p and state (s, m_r) is in r.

—When m_r is the empty heap $[]$, we split $h'_1 * h * h'_2$ to the empty heap $[]$, the heap m_p and the heap m. This $([] * m_p) * m$ is the required splitting for $(s, h'_1 * h * h'_2)[\mathsf{great}(p * r, q * r, X)](s, h'_1 * h * h'_2)$. It is because $(s, [])$ is in q and (s, m_p) is in r by the definition of r.

—When m_r is $h'_1 * h * h'_2$, both m_p and m are empty. The required partitioning of $h'_1 * h * h'_2$ is $([] * (h'_1 * h * h'_2)) * []$, because $(s, [])$ is in q and $(s, h'_1 * h * h'_2)$ is in r.

—When m_r is a subheap of $h'_1 * h * h'_2$ satisfying p (that is, $(s, m_r) \in p$), either heap m_p is $(h'_1 * h * h'_2) - m_r$, or both m_p and m_r are empty. The reason is that if m_p is a strict subheap of $(h'_1 * h * h'_2) - m_r$, then the cardinality of $m_p \cup m_r$ is strictly smaller than that of $h_1 \cup h_2$; because both (s, m_p) and (s, m_r) are in p, the heaps m_p and m_r must be equal, which, under $m_r \# m_p$, implies $m_p = m_r = []$. When m_p is $(h'_1 * h * h'_2) - m_r$, the required splitting of $h'_1 * h * h'_2$ is $([] * (h'_1 * h * h'_2)) * []$, because $(s, [])$ is in q and $(s, h'_1 * h * h'_2)$ is in r. In the other case, where both m_p and m_r are empty, the splitting $([] * []) * (h'_1 * h * h'_2)$ of $h'_1 * h * h'_2$ becomes the required one, because $(s, []) \in q$ and $(s, []) \in r$.

For the second fact, note that $\mathsf{great}(p, q, X)$ can not relate $(s, h'_1 * h * h'_2)$ to any states, because two different sub-states (s, h_1) and (s, h_2) of $(s, h'_1 * h * h'_2)$ are in p, but q is the singleton set. This completes the proof of the if part of (a).

We prove the if direction of (b) by contradiction. Suppose that r is not precise. Then, r has states (s, h_1) and (s, h_2) such that h_1 and h_2 are different but consistent: for all l in $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2)$, we have $h_1(l) = h_2(l)$. We pick such s, h_1, h_2 in such a way that the cardinality of $\mathsf{dom}(h_1 \cup h_2)$ is the least. Let h be the restriction of h_1 to $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2)$, and let h'_1 and h'_2 be $h_1 - h$ and $h_2 - h$, respectively. Since h_1 is different from h_2, at least one of h'_1 and h'_2 must be nonempty. Without loss of generality, we assume that h'_1 is not empty. Define predicate p to be the set $\{(s, h'_1), (s, [])\}$, and predicate q the set $\{(s, h'_2), (s, [])\}$. We claim that $\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p * r, q * r, X)$ does not hold. To prove this claim, we show that

(1) $(s, h'_1 * h * h'_2)\ [\mathsf{great}(p * r, q * r, X)]\ (s, h'_1 * h * h'_2)$ holds; but

(2) there are no splittings of $h'_1 * h * h'_2$ into heaps m, m_r such that (s, m_r) is in r and $(s, h'_1)[\mathsf{great}(p, q, X)](s, m)$ holds.

Note that the claim follows from these two, because $(s, h * h'_2)$ is in r and $\mathsf{great}(p, q, X)$ is safe at (s, h'_1). That is, if we establish 1 and 2, we will have contradicted the generalized frame property.

For 1, from the definition of great it suffices to show that

3. for every partitioning of $h'_1 * h * h'_2$ into three parts m_p, m_r, and m, if (s, m_p) is in p and (s, m_r) is in r, heap $h'_1 * h * h'_2$ can be split into the heap m and some heaps m'_q and m'_r such that (s, m'_q) is in q and (s, m'_r) is in r.

Let $m_p * m_r * m$ be a partitioning of $h'_1 * h * h'_2$ such that (s, m_p) is in p and (s, m_r) is in r. Because of the definition of p, heap m_p is either h'_1 or $[]$. We consider these two cases separately:

—When m_p is h'_1, heap m_r must be equal to $h * h'_2$ because r can not contain a state whose stack is s and whose heap is a strictly smaller subheap of h'_2: since h'_1 is not empty, if r did have such a state, the s, h_1, h_2 could not be the least in terms of the cardinality of $\mathsf{dom}(h_1 \cup h_2)$. So, m is the empty heap $[]$. We split $h'_1 * h * h'_2$ three ways, into h'_2, $h'_1 * h$, and the empty heap $[]$. Since $h'_1 * h$ is h_1, state $(s, h'_1 * h)$ is in r so that $(s, h'_2 * (h'_1 * h))$ is in $q * r$. Therefore, the splitting $(h'_2 * h'_1 * h) * [] = h'_1 * h * h'_2$ is the required one for property 3 above.

—When m_p is the empty heap, we partition $h'_1 * h * h'_2$ to $[]$, m_r and m. The splitting $([] * m_r) * m$ of $h'_1 * h * h'_2$ is the required one for property 3 because $(s, [])$ is in q and (s, m_r) is in r.

For 2 suppose, toward contradiction, that there is a splitting $m * m_r = h'_1 * h * h'_2$ of $h'_1 * h * h'_2$ making both $(s, m_r) \in r$ and $(s, h'_1)[\mathsf{great}(p, q, X)](s, m)$ true. Since $(s, [])$ is in p and $(s, h'_1)[\mathsf{great}(p, q, X)](s, m)$ holds, heap h'_1 must be a subheap of m. Moreover, since (s, h'_1) is in p and $(s, h'_1)[\mathsf{great}(p, q, X)](s, m)$ holds, state (s, m) must be in q. Therefore, q must have a state whose heap has h'_1 as a subheap. But, q does not have such a state: h'_1 is not a subheap of $[]$ or h'_2. Thus, we have the required contradiction. □

12. SUPPORTED AND INTUITIONISTIC PREDICATES

We consider two further special classes of predicates, which can be used to furnish further sufficient conditions for the soundness of the hypothetical frame rule. A *supported* predicate is one where, for any given heap, if the collection of subheaps satisfying the predicate is not empty, it has a least element. An intuitionistic predicate is one whose truth is invariant under heap enlargement. These kinds of predicate are of interest in situations, such as in a garbage collected language, where one would like to write loose specifications which state that a certain data structure is present in memory, without worrying about whether there are additional cells.

DEFINITION 24. [Supported and Intuitionistic Predicates]
A predicate p is supported *if and only if for all states (s, h), when h has a subheap h' satisfying $(s, h') \in p$, there is a least subheap h_p of h with $(s, h_p) \in p$: for all subheaps h' of h, if (s, h') satisfies p, we have $h_p \leq h'$. A predicate p is* intuitionistic *if and only if for every state (s, h), if $(s, h) \in p$ and $h \leq h'$ then also $(s, h') \in p$.*

Supported predicates do not ensure unique heap splittings in the way that precise ones do. However, if p is precise and q intuitionistic then we can choose a canonical splitting: make the heap for p be the least possible one, and enlarge the heap for q as much as necessary.

The reader will have noticed intuitive similarity between the concepts of precise and supported predicates. For each precise predicate p, the predicate $p * \mathtt{true}$ is intuitionistic and supported. In fact, there is a converse map from intuitionistic supported predicates to precise predicates.

LEMMA 25. *If a predicate p is precise, the predicate $p * \mathtt{true}$ is intuitionistic and supported, and if a predicate q is intuitionistic and supported, the predicate*

$q \wedge \neg(q * \neg\mathtt{emp})$ *is precise. Moreover, we have*

$$\big((p * \mathtt{true}) \wedge \neg((p * \mathtt{true}) * \neg\mathtt{emp})\big) = p$$

for all precise predicates p*, and*

$$(q \wedge \neg(q * \neg\mathtt{emp})) * \mathtt{true} = q$$

for all intuitionistic supported predicates q*.*

Proof: For each predicate p, we have the following equivalences:

$$\begin{array}{c}(s,h) \in p * \mathtt{true} \iff \exists h' \leq h.\ (s,h') \in p \\ (s,h) \in \big(p \wedge \neg(p * \neg\mathtt{emp})\big) \iff (s,h) \in p \quad \text{and} \quad \forall h' < h.\ (s,h') \notin p\end{array}$$

With these equivalences, it is straightforward to show that if p is precise, the predicate $p * \mathtt{true}$ is intuitionistic and supported. Next, we will show that for every intuitionistic supported predicate p, the predicate $p \wedge \neg(p * \neg\mathtt{emp})$ is precise. Consider a state (s,h) such that h has a subheap h' such that $(s,h') \in p \wedge \neg(p * \neg\mathtt{emp})$. Because of the above equivalence for $p \wedge \neg(p * \neg\mathtt{emp})$, state (s,h') is in p, but for all strict subheaps h'' of h', we have $(s,h'') \notin p$. Since p is supported, h' is the unique subheap of h such that $(s,h') \in p$ and h' is the least such. This shows that $p \wedge \neg(p * \neg\mathtt{emp})$ is precise.

It remains to show that

—if p is intuitionistic and supported, we have $(p \wedge \neg(p * \neg\mathtt{emp})) * \mathtt{true} = p$; and
—if p is precise, we have $((p * \mathtt{true}) \wedge \neg((p * \mathtt{true}) * \neg\mathtt{emp})) = p$.

We show the first implication as follows:

$$\begin{array}{ll}& (s,h) \in \big((p \wedge \neg(p * \neg\mathtt{emp})) * \mathtt{true}\big) \\ \iff & \exists h' \leq h.\ (s,h') \in \big(p \wedge \neg(p * \neg\mathtt{emp})\big) \\ \iff & \exists h' \leq h.\ \big((s,h') \in p \text{ and } \big(\forall h'' < h'.\ (s,h'') \notin p\big)\big) \\ \iff & (s,h) \in p \qquad (\because p \text{ is intuitionistic and supported}).\end{array}$$

The second implication holds because of the following:

$$\begin{array}{ll}& (s,h) \in \big(p * \mathtt{true} \wedge \neg((p * \mathtt{true}) * \neg\mathtt{emp})\big) \\ \iff & \big((s,h) \in p * \mathtt{true} \quad \text{and} \quad \forall h' < h.\ (s,h') \notin p * \mathtt{true}\big) \\ \iff & (s,h) \in p \qquad\qquad\qquad\qquad (\because p \text{ is precise}).\end{array}$$

□

THEOREM 26. *The hypothetical frame rule is sound in the following cases:*

(a) the preconditions $p_1, \ldots, p_n$ *are supported, and the postconditions* $q_1, \ldots, q_n$ *are intuitionistic; or*

(b) the resource invariant r *is supported, and the postconditions* $q_1, \ldots, q_n$ *are intuitionistic.*

Notice that the first point does not contradict the only if part of Theorem 8(a), because it mentions postconditions in addition to preconditions. Likewise, the

second point does not contradict Theorem 8(b), because it mentions postconditions as well as the resource invariant.

The proof of this theorem makes use of the decomposition into modifies weakening and the simple hypothetical frame rule, and goes as in Section 11.5, with the following key proposition.

PROPOSITION 27.

(a) If a predicate p is supported and a predicate q is intuitionistic, then

$$\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p * r, q * r, X)$$

holds for all predicates r and sets X of variables.

(b) If a predicate r is supported and a predicate q is intuitionistic, then

$$\mathsf{great}(p, q, X)[\mathsf{sim}(R_r)]\mathsf{great}(p * r, q * r, X)$$

holds for for all predicates p and sets X of variables.

Proof: We prove (a) using Lemma 22. Consider states $(s, h), (s'_1, h'_1)$ and a heap h_r such that $\mathsf{great}(p, q, X)$ is safe at (s, h), state (s, h_r) is in r, and $(s, h * h_r)$ and (s'_1, h'_1) are related by $\mathsf{great}(p * r, q * r, X)$. Since $\mathsf{great}(p, q, X)$ is safe at (s, h) and p is supported, heap h can be split into h_p and h_0 such that h_p is the least subheap of h that makes $(s, h_p) \in p$ hold. Since $(s, h * h_r)[\mathsf{great}(p * r, q * r, X)](s'_1, h'_1)$ and $(h_p * h_r) * h_0 = h * h_r$, we can partition h'_1 into the heap h_0 and some heaps h'_q and h'_r such that $(s', h'_q) \in q$ and $(s', h'_r) \in r$. We claim that $h'_q * h_0$ and h'_r form the required partitioning of h'_1 by Lemma 22. Since (s', h'_r) is in r, it suffices to show that for all splittings $m_p * m = h$ of h, if the state (s, m_p) is in p, heap $(h'_q * h_0)$ can be split into the heap m and some heap m'_q (i.e., $m * m'_q = h'_q * h_0$) such that (s', m'_q) is in q. Consider a splitting $m_p * m = h$ of h such that the state (s, m_p) is in p. Heap m_p must have h_p as a subheap, because h_p is the least heap such that $(s, h_p) \in p$. So, m must be a subheap of h_0. We split the heap $h'_q * h_0$ into $h'_q * (h_0 - m)$ and m. The state $(s', h'_q * (h_0 - m))$ is in q because q is intuitionistic and (s', h'_q) is already in q. Thus, this partitioning $(h'_q * (h_0 - m)) * m$ is the required one, and we have finished the proof of (a).

We also prove (b) using Lemma 22. Consider states $(s, h), (s'_1, h'_1)$ and a heap h_r such that $\mathsf{great}(p, q, X)$ is safe at (s, h), state (s, h_r) is in r, and $(s, h * h_r)$ and (s'_1, h'_1) are related by $\mathsf{great}(p * r, q * r, X)$. Since $\mathsf{great}(p, q, X)$ is safe at (s, h), heap h can be split into h_p, h_0 (i.e., $h_0 * h_p = h$) such that $(s, h_p) \in p$. Since $(s, h * h_r)[\mathsf{great}(p * r, q * r, X)](s'_1, h'_1)$ and $h_p * h_r * h_0 = h * h_r$, we can partition the heap h'_1 into the heap h_0 and some heaps h'_q, h'_r (i.e., $h'_q * h'_r * h'_0 = h'_1$) such that (s', h'_q) is in q and (s', h'_r) is in r. In particular, we can partition h'_1 in such a way that h'_r is the least subheap of h'_1 such that $(s', h'_r) \in r$, because the invariant r is supported and the postcondition q is intuitionistic. The splitting $(h'_q * h_0) * h'_r$ of h'_1 is the required one by Lemma 22. To see the reason, suppose that h is split into m_p and m such that $(s, m_p) \in p$. Since $(s, h * h_r)[\mathsf{great}(p * r, q * r, X)](s'_1, h'_1)$, we can split h'_1 into the heap m and some heaps m'_q, m'_r such that (s', m'_q) is in q and (s', m'_r) is in r. Heap m'_r has h'_r as a subheap because h'_r is the least subheap of h'_1 satisfying $(s', h'_r) \in r$. Now, we can partition heap $(h'_q * h_0)$ into $m'_q * (m'_r - h'_r)$ and m, where $m'_r - h'_r$ is the restriction of m'_r to those cells in $\mathsf{dom}(m'_r) - \mathsf{dom}(h'_r)$.

Since q is intuitionistic and (s', m'_q) is in q, state $(s', m'_q * (m'_r - h'_r))$ is also in q. This gives the conclusion. □

13. CONCLUSION

This paper has two main contributions. First, we described a new inference rule, the hypothetical frame rule, which gives a powerful way of hiding resources when reasoning about heap-manipulating programs. Second, we provided a theoretical analysis of when the new rule is sound.

The theoretical work was nontrivial, and to ease it we focussed attention on a simplified programming language with parameterless procedures only. The extension to first-order procedures is probably not difficult: we could follow the ideas of [Hoare 1971; Cook 1978] in the treatment of procedures in Hoare logic, which utilizes certain anti-aliasing conditions. The extension to higher-order procedures is another matter. Proof rules for higher-order procedures are generally difficult, quite apart from issues of information hiding.

Perhaps the most significant previous work that addresses information hiding in program logics, and that confronts mutable data structures, is that of [Leino and Nelson 2002]. They use auxiliary variables (like our use of the variable Q in Table IV) to specify modules, and they develop a subtle notion of "modular soundness" that identifies situations when clients cannot access the internal representation of a module. This much is similar in spirit to what we are attempting, but on the technical level we are not sure if there is any relationship between the separating conjunction and their notion of modular soundness.

The information-hiding problems caused by pointers have been a concern for a number of years in the object-oriented types community, beginning with Hogg's colorful declaration "that objects provide encapsulation is the big lie of object-oriented programming" [Hogg 1991]. A focal point of that work has been a concept of "confinement", which disallows or controls pointers into data representations. Some confinement systems use techniques similar to regions, with control over the number and direction of pointers across region boundaries [Clarke et al. 2001; Grothoff et al. 2001].

In this paper an emphasis was placed on ownership transfer, which allows the dynamic reconfiguration of resource partitions between program components. This phenomenon is not uncommon in systems and object-oriented programs; good examples are given by `malloc()` and `free()`, and by thread pool managers used in (e.g.) web servers. The type systems for confinement have difficulty dealing with such idioms.

At the time of publication of the preliminary version of this paper in the POPL'04 proceedings there was relatively little related work on program logic and information hiding (and abstraction) in the presence of the heap. In addition to [Leino and Nelson 2002] we mention also [Müller and Poetzsch-Heffter 2000]. But since then there have been many further developments. We briefly mention some of them to conclude the paper.

[Birkedal et al. 2005] have described an extension of the work here to call-by-name higher-order procedures, for a language that mixes procedures and state in the manner of idealized Algol. They have described higher-order frame rules, where the

hypothetical rule here is second-order. The higher-order frame rules are formulated for call-by-name procedures, which combine state and functions in the way that Idealized Algol and Haskell do, rather than the way that ML does.

Incidentally, the work in [Birkedal et al. 2005] uses a form of semantics that validates the hypothetical frame rule without a restriction on precision, while denying the conjunction rule. This represents a different reaction to the conundrum of Section 6 than the one taken in this paper, and the consistency of which further underlines the subtleties surrounding the rules. The semantics is a kind of possible-worlds model, where the worlds are resource invariants. While it is perhaps not easy to justify the invalidity of the conjunction law of intuitive grounds, this possible-worlds construction has proven to be very useful in situations where the existence of a model is difficult to come by; see [Birkedal and Yang 2007; Nanevski et al. 2006; Benton 2006; Peterson et al. 2008] for further applications of that semantics.

An approach to reasoning about objects using separation logic has been developed in [Parkinson and Bierman 2005; Parkinson 2005]. Instead of hiding resource invariants, as here, there the approach is to let the client use abstract predicates (i.e., predicate variables) when reasoning about the use of an object, without knowing the definition of the predicate. In a sense, this transports the fundamental ideas on polymorphism and data abstraciton [Reynolds 1983; Mitchell and Plotkin 1988] from types for functional languages to logic for imperative programs.

Parkinson suggests that the hypothetical frame rule and abstract predicates are complementary: the former addresses information hiding, and the latter addresses abstraction. For some programs (e.g., with malloc/free) the specifications using the hypothetical frame rule are more succinct than with abstract predicates alone. On the other hand, the abstract predicates are very powerful. They support multi-instance classes easily, examples where the class invariant is not the natural concept to use in a specification, and cases where we want to distinguish different internal states of objects without telling the entire invariant. The many examples in [Parkinson and Bierman 2005; Parkinson 2005; 2007] confirm the power and naturality of abstract predicates. See also the examples and theory developed in [Biering et al. 2007; Krishnaswami et al. 2007].

A significant recent line of work on reasoning about objects, the Boogie methodology, takes some inspiration from type systems for objects, particularly ownership typing schemes, but uses assertions rather than types to describe an ownership hierarchy [Barnett et al. 2004]. As a consequence it is considerably more flexible than the typing systems; for example, it deals with ownership transfer by, like here, allowing for dynamically changing partitions [Banerjee and Naumann 2005]. While more flexible than the type systems, it is not always the case that a data abstraction fits naturally into a hierarchy. An example is a queue, where neither the left nor the right end of the queue conceptually dominates the other in a hierarchy. As a result, a number of extensions or alterations of Boogie have been developed which aim to deal with the inflexibility of the hierarchy structures (e.g., NaumannBarnett04b, NaumannBarnett04,LeinoMuller04). We refer to the survery article [Naumann 2007] for further information and references concerning work on Boogie and other work on specifying object-oriented programs.

A distinguishing feature of the work on Boogie is that they deal with the possi-

bility of re-entrant (recursive) modules where an invariant is temporarily broken. Here, if we combine the hypothetical frame rule with the standard rule for recursive procedures, we obtain a proof rule for modules where the resource invariant must be true before each call of a module procedure. It has been claimed that there are natural programming patterns where one wants the invariants to be broken, and ingenious solutions have been devised in the work on Boogie, where auxiliary variables are used to tell a client when an invariant need and need not hold, without telling the client what the invariant is. This issue is too subtle for a full discussion here, and we refer to the survey paper of Naumann referenced above for a detailed account. We also refer to [Parkinson 2007] for an interesting turnabout, where it is (convincingly) argued that the problem lies with the class invariant concept itself, that technical complexities to do with invariants and re-entrance are a symptom of a mistaken assumption (that we should start with class invariants), rather than a fundamental problem to be solved.

We emphasize that the hypothetical frame rule does not take any stance on this "object invariants" problem. It could conceivably be used in concert with Parkinson's solution, and there is nothing in the rule which says that an invariant has to refer to a single object.

Finally, we have stayed in a sequential setup in this paper, but the ideas are relevant to concurrent programming. Indeed, the paper arose originally as a result of a problem in concurrency. In unpublished notes from August 2001, O'Hearn described proof rules for concurrency using $*$ to express heap separation, and showed program proofs where storage moved from one process to another. The proof rules were not published, because O'Hearn was unable to establish their soundness. Then, in August 2002, Reynolds showed that the rules were unsound if used without restriction, and this lead to our focus on precise assertions. Both the promise and subtlety of the proof rules had as much to do with information hiding as concurrency, and it seemed unwise to attempt to tackle both at the same time. At the time of Reynolds's discovery we had already begun work on the hypothetical frame rule, and the counterexample appears here as the conundrum in Section 6.

The work reported in this paper provided the first resolution of Reynolds's conundrum, with the semantic analysis revolving around the concept of precise predicate. Precision was then subsequently used as part of the resolution of the original issues in the concurrent setting [Brookes 2007], which finally allowed the publication of the proof rules that had been circulated in 2001 [O'Hearn 2007].

Acknowledgements. We have benefitted greatly from discussions with Josh Berdine, Richard Bornat and Cristiano Calcagno. O'Hearn' was supported by the EPSRC and by a Royal Society Wolfson Research Merit Award. Reynolds was partially supported by an EPSRC Visiting Fellowship at Queen Mary, by National Science Foundation Grant CCR-0204242, and by the Basic Research in Computer Science (http://www.brics.dk/) Centre of the Danish National Research Foundation. Yang was supported by the EPSRC and by the Basic Research Program of the Korea Science & Engineering Foundation.

REFERENCES

Ahmed, A., Jia, L., and Walker, D. 2003. Reasoning about hierarchical storage. In *18th LICS*, pp33-44.

Banerjee, A. and Naumann, D. A. 2005. State based ownership, reentrance, and encapsulation. In 19th ECOOP.

Barnett, M., DeLine, R., Fahndrich, M., Leino, K., and Schulte, W. 2004. Verification of object-oriented programs with invariants. *Journal of Object Technology 3(6)*, 27–56.

Benton, N. 2006. Abstracting allocation. In *20th CSL*. 182–196.

Biering, B., Birkedal, L., and Torp-Smith, N. 2007. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Programming Languages and Systems 29,* 5.

Birkedal, L., Torp-Smith, N., and Yang, H. 2005. Semantics of separation-logic typing and higher-order frame rules. In *20th LICS*.

Birkedal, L. and Yang, H. 2007. Relational parametricity and separation logic. In *10th FOSSACS*.

Bornat, R. 2000. Proving pointer programs in Hoare logic. *Mathematics of Program Construction*.

Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. 2005. Permission accounting in separation logic. In *32nd POPL*, pp59–70.

Brinch Hansen, P., Ed. 2002. *The Origin of Concurrent Programming*. Springer-Verlag.

Brookes, S. D. 2007. A semantics of concurrent separation logic. *Theoretical Computer Science 375(1-3)*, 227–270. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp16-34).

Clarke, D., Noble, J., and Potter, J. 2001. Simple ownership types for object containment. *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53-76, Springer LNCS 2072.

Cook, S. A. 1978. Soundness and completeness of an axiomatic system for program verification. *SIAM J. on Computing 7*, 70–90.

Grothoff, C., Palsberg, J., and Vitek, J. 2001. Encapsulating objects with confined types. Proceedings of OOPSLA, pp241–253.

Hoare, C. A. R. 1971. Procedures and parameters: An axiomatic approach. In *Symposium on the Semantics of Algebraic Languages*, E. Engler, Ed. Springer, 102–116. Lecture Notes in Math. 188.

Hoare, C. A. R. 1972a. Proof of correctness of data representations. Acta Informatica 4, 271-281.

Hoare, C. A. R. 1972b. Towards a theory of parallel programming. In *Operating Systems Techniques*, Hoare and Perrot, Eds. Academic Press.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Comm. ACM 17,* 10, 549–557. Reprinted in [Brinch Hansen 2002].

Hogg, J. 1991. Islands: aliasing protection in object-oriented languages. 6th OOPSLA.

Isthiaq, S. and O'Hearn, P. W. 2001. BI as an assertion language for mutable data structures. In *28th POPL*. 36–49.

Kernighan, B. and Ritchie, D. 1988. *The C Programming Language*. Prentice Hall. Second edition.

Krishnaswami, N., Aldrich, J., and Birkedal, L. 2007. Amodular verification of the subject-observer pattern via higher-order separation logic. In *9th Workshop on Formal Techniques for Java-like Programs*.

Leino, K. R. M. and Nelson, G. 2002. Data abstraction and information hiding. *ACM TOPLAS 24(5)*, 491–553.

Mitchell, J. C. and Plotkin, G. D. 1988. Abstract types have existential types. *ACM Trans. Programming Languages and Systems 10,* 3, 470–502.

Morgan, C. 1988. The specification statement. *ACM TOPLAS 10,* 3 (Jul), 403–419.

Müller, P. and Poetzsch-Heffter, A. 2000. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*. Cambridge University Press.

Nanevski, A., Morrisett, G., and Birkedal, L. 2006. Polymorphism and separation in Hoare type theory. In *ICFP 2006*. 62–73.

Naumann, D. 2007. On assertion-based encapsulation for object invariants and simulations. *Formal Asp. Comput. 19(2): 205-224*.

O'Hearn, P., Reynolds, J., and Yang, H. 2001. Local reasoning about programs that alter data structures. In *15th CSL*, pp1-19.

O'Hearn, P. W. 2007. Resources, concurrency and local reasoning. *Theoretical Computer Science 375(1-3)*, 271–307. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp49-67).

O'Hearn, P. W., Yang, H., and Reynolds, J. C. 2004. Separation and information hiding. In *31st POPL*. 268–280.

Parkinson, M. 2005. Local reasoning for java. Ph.D. thesis, Cambridge University, Cambridge, UK.

Parkinson, M. 2007. Class invariants: the end of the road? Position paper presented at *3rd IWACO*.

Parkinson, M. and Bierman, G. 2005. Separation logic and abstraction. In *32nd POPL*, pp59–70.

Parkinson, M., Bornat, R., and Calcagno, C. 2006. Variables as resource in Hoare logics. In *21st LICS*.

Parnas, D. 1972a. Information distribution aspects of design methodology. Proceedings of IFIP Congress (1) 1971: 339-344.

Parnas, D. 1972b. On the criteria to be used in decomposing systems into modules. CACM, 15(12):1053–1058.

Peterson, R., Birkedal, L., Nanevski, A., and Morrisett, G. 2008. A realizability model of impredicative Hoare type theory. In *17th ESOP*.

Pierce, B. and Turner, D. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming 4,* 2, 207–247.

Reynolds, J. C. 1983. Types, abstraction and parametric polymorphism. Proceedings of IFIP.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp55-74.

Schwarz, J. S. 1974. Semantics of partial correctness formalisms. Ph.D. thesis, Syracuse University, Syracuse, New York.

Schwarz, J. S. 1977. Generic commands - a tool for partial correctness formalisms. *The Computer Journal 20,* 2 (May), 151–155.