Separation and Information Hiding

Peter W. O'Hearn
Queen Mary
University of London

Hongseok Yang
Seoul National
University

John C. Reynolds
Carnegie Mellon
University

Abstract

We investigate proof rules for information hiding, using the recent 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

Keywords: 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 desirable, 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 recent formalism for reasoning about mutable data structures [36].

Our treatment draws on work of Hoare on proof rules for data abstraction and for shared-variable concurrency [16, 17, 18]. 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

POPL'04, January 14–16, 2004, Venice, Italy.

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. This revolves around a notion of "precise" predicates, which are ones that unambiguously identify a portion of state. In essence, we ask the Asserter to be unambiguous when specifying which resource is owned; when this is the case, we find that our proof rules are sound.

Familiarity with the basics of separation logic, as presented in [36], 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 [26] 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 [16, 17, 28, 29]. 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., [30]), 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. We will say more on some of the closely related work at the end of the paper.

2 The Storage Model

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

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

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

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

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 [36] and the hierarchical memory model of [1]. 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).

A natural model of separation that is not an instance of the partial functions model construction above is the "trees with dangling pointers" model of [6]; it would be interesting to axiomatize the essentials of these separation models, by identifying a subclass of the partial monoid models of [32].

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:

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

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

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

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:

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

As a function on predicates we have a total map $*$ from $\mathsf{Predicates} \times \mathsf{Predicates}$ to Predicates which, to the right of $\stackrel{\text{def}}{=}$, 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.

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

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 $\mathtt{let}\ k_1 = C_1, \ldots, k_n = C_n\ \mathtt{in}\ C$ to indicate this.

The command $x := \mathtt{cons}(E_1,\ldots,E_n)$ 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 $\mathtt{dispose}(E)$ deallocates the address E. In $x := [E]$, $[E] := F$ and $\mathtt{dispose}(E)$, 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:

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

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

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

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

3.1 Proof Rules for Information Hiding

We begin with a special-case, programmer-friendly, proof rule, that is a consequence of a more fundamental, logician-friendly, rule to be described later.

Modular Non-Recursive Procedure Declaration Rule

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

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* $k_1, \ldots, k_n$;
- Y is disjoint from p, q, C and the context "$\Gamma, \{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n]$";
- C_i only modifies variables in X_i, Y.

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. A rigorous formulation of what these conditions mean has been placed in an appendix at the end of the paper. We will continue to state the necessary side conditions as we present our proof rules, but there will be little harm if the reader skates over them, or understands them in an intuitive way, while reading the paper. We only stress that the modifies clauses refer exclusively to the stack, where the new part of the paper involves the heap, and $*$.

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

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

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.

However, the choice of invariant r is not specified by programming language syntax $\mathtt{let}\ k_1 = C_1, \ldots, k_n = C_n\ \mathtt{in}\ C$ 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.

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
$C_1, \ldots, C_n$

Table 1. Module Specification Format

We will use the format for module specifications in Table 1. This instructs us to apply the modular procedure rule in a particular way, to prove

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

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.

The modular procedure rule can be derived from a standard rule for parameterless procedure declarations, and the following more basic rule.

Hypothetical Frame Rule

$$\frac{\Gamma, \{p_i\}k_i\{q_i\}[X_i]_{(\text{for } i \leq n)} \vdash \{p\}C\{q\}}{\Gamma, \{p_i * r\}k_i\{q_i * r\}[X_i, Y]_{(\text{for } i \leq n)} \vdash \{p * r\}C\{q * r\}}$$

where

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

The notation $\{p_i\}k_i\{q_i\}[X_1]_{(\text{for } i \leq n)}$ in the rule is a shorthand for $\{p_1\}k_1\{q_1\}[X_1], \ldots, \{p_n\}k_n\{q_n\}[X_n]$, and similarly for $\{p_i * r\}k_i\{q_i * r\}[X_1, Y]_{(\text{for } i \leq n)}$. In examples we will use the modular procedure rule, but will phrase our theoretical results in terms of the hypothetical frame rule.

The hypothetical frame rule is so named because of its relation to the ordinary frame rule from [20, 26]. 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 $n = 0$.)

3.2 Other Proof Rules

We have standard Hoare logic rules for various constructs, along with the rule of consequence.

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

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

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

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

The rule for possibly recursive procedure declarations uses the procedure specifications in proofs of the bodies:

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

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 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.

Finally, we have axioms for the basic commands, where x, m, n are assumed to be distinct variables.

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

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

$$\Gamma \vdash \left\{\begin{array}{l} x = m \\ \wedge\, \mathtt{emp}\end{array}\right\} x := \mathtt{cons}(E_1, \ldots, E_k)\{x \mapsto E_1[m/x], \ldots, E_k[m/x]\}$$

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

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

These axioms describe the effect of each command on only one, or sometimes no, heap cells. Typically, their 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.

Interface Specifications	
$\{\mathtt{emp}\}\mathtt{alloc}\{x \mapsto -,-\}\,[x]$	
$\{x \mapsto -,-\}\mathtt{free}\{\mathtt{emp}\}\,[]$	
Resource Invariant: $list(f)$	
Private Variables: f	
Internal Implementations	
$\mathtt{if}\ f = \mathtt{nil}\ \mathtt{then}\ x := \mathtt{cons}(-,-)$ $\mathtt{else}\ x := f; f := x.2;$	(code for `alloc`)
$x.2 := f; f := x;$	(code for `free`)

Table 2. Memory Manager Module

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 2.

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 $list(f)$ is the representation invariant, where

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

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 [22]. 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 2 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`.

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

The most important step is the middle application of the rule of consequence. At that point we still have the original resource invariant $list(f)$ and the knowledge that x points to something, separately. But since the second field of what x points to holds f, we can obtain $list(x)$ 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

$\{(y \mapsto a,z) * (z \mapsto c,w) * list(f)\}$
`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.

$\{(y \mapsto a,x) * (x \mapsto b,z) * (z \mapsto c,w)\}$
`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. (A more positive example of aliasing, which incidentally would not be amenable to unique-reference disciplines, would be a program to dispose nodes in a graph.)

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

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

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 3 we give a queue module. In the specification we use a predicate $listseg(x, \alpha, y)$ which says that there is an acyclic linked list from x to y 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 "abstract" variables is standard in module specifications, as is the inclusion of assignment statements involving abstract variables

Interface Specifications

$\{Q = \alpha \wedge z = n \wedge P(z)\}$ enq $\{Q = \alpha\cdot\langle n\rangle \wedge \mathtt{emp}\}\ [Q]$

$\{Q = \langle m\rangle\cdot\alpha \wedge \mathtt{emp}\}$ deq $\{Q = \alpha \wedge z = m \wedge P(z)\}\ [Q,z]$

$\{\mathtt{emp}\}$ isempty? $\{(w = (Q = \varepsilon)) \wedge \mathtt{emp}\}\ [w]$

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

Private Variables: x, y, t

listseg Predicate Definition

$$\begin{array}{l} \mathit{listseg}(x,\alpha,y) \ \stackrel{\text{def}}{\iff} \\ \mathtt{if}\ x = y\ \mathtt{then}\ (\alpha = [\,] \wedge \mathtt{emp}) \\ \mathtt{else}\ \big(\exists v,z,\alpha'.\ (\alpha = \langle v\rangle\cdot\alpha' \wedge x \mapsto v,z) * P(v) \\ \qquad\qquad\qquad\qquad\qquad * \mathit{listseg}(z,\alpha',y)\big) \end{array}$$

Internal Implementations

$Q := Q\cdot\langle z\rangle$; (code for enq)
$t :=$ cons$(-,-)$; $y.1 := z$; $y.2 := t$; $y := t$

$Q :=$ cdr(Q); (code for deq)
$z := x.1$; $t := x$; $x := x.2$; dispose(t)

$w := (x = y)$ (code for isempty?)

Table 3. Queue Module, Parametric in $P(v)$

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.

$\{Q = \langle n\rangle\cdot\alpha \wedge \mathtt{emp}\}$
  deq
$\{Q = \alpha \wedge z = n \wedge P(z)\}$
  $z.1 := 42$
$\{???\}$

In case $P(v)$ is either emp or $\mathit{list}(v)$ 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 sequent in the premise:

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

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, and 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 $\mathit{list}(f)$ 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.

We pursue the first two options in the remainder of the paper, by defining a model of the programming language and investigating a notion of precise predicate.

7 A Denotational 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. 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 program, but the ability to refer to it leads to significant simplifications in proofs about the semantics.

Recall that a state consists of a pair, of a stack and a heap. A command is interpreted as a binary relation

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

that relates an input state to possible output states, or a special output, `fault`, which indicates an attempted access of an address not in the domain of the heap. In fact, because we use a fault-avoiding interpretation of Hoare triples, it would be possible to use the domain

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

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 according to the semantics of triples. 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, so that our semantics will not distinguish a command C from one that nondeterministically chooses C or divergence.

We will not consider all relations, but rather only those that validate the locality properties of the (ordinary) frame rule. We say that a relation c: $\mathsf{States} \leftrightarrow \mathsf{States} \cup \{\mathtt{fault}\}$ is *safe* at a state (s,h) when $\neg((s,h)\,[c]\,\mathtt{fault})$. We just list the properties here, and refer the reader to [39] for further explanation of them. The locality properties are:

1. $\mathsf{Safety\ Monotonicity}$: 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. $\mathsf{Frame\ Property}$: 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 $h'_0 \leq h'$ such that
$$h'_0\#h_1,\ h'_0*h_1 = h',\text{ and } (s,h)[c](s',h'_0).$$

Commands will be interpreted using the following domain.

> The poset LRel of "local relations" is the set of all relations c satisfying the safety monotonicity and frame properties, ordered by subset inclusion.

LEMMA 1. LRel *is a chain-complete partial order with a 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.*

The meaning of a command is given in the context of an environment η, that maps procedure identifiers to relations in LRel.

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

The semantics of expressions depends only on the stack

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

The valuations are standard and omitted.

Selected valuations for commands are in Table 4. The main point to note is the treatment of `fault`. We have included only the basic commands and sequential composition. The interpretation of conditionals is as usual, a procedure call applies the environment to the corresponding variable, and while loops and `letrec` receive standard least fixed-point interpretations, which are guaranteed to exist by Lemma 1.

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

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

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

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, 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 $y \notin X$ and $(s,h)[c](s',h')$, we have that $s(y) = s'(y)$.
- $\{p\}c\{q\}$ holds if and only if for all states (s,h) in p,
  1. $\neg((s,h)\,[c]\,\mathtt{fault})$; and
  2. if $(s,h)\,[c]\,(s',h')$ then state (s',h') is in q.

Now we can define the semantics: A sequent

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

holds if and only if

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

$$(s,h)[[\![x := E]\!]\eta]a \iff a = (s[x \mapsto [\![E]\!]s], h)$$

$$(s,h)[[\![x := \mathtt{cons}(E_1,\ldots,E_n)]\!]\eta]a \iff \exists m.\ \ (m,\ldots,m+n-1 \not\in \mathsf{dom}(h)) \\ \wedge \big(a = (s[x \mapsto m], h * [m \mapsto [\![E_1]\!]s, \ldots, m+n-1 \mapsto [\![E_n]\!]s])\big)$$

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

$$(s,h)[[\![[E] := F]\!]\eta]a \iff \text{if } [\![E]\!]s \in \mathsf{dom}(h) \text{ then } a = (s, h[[\![E]\!]s \mapsto [\![F]\!]s]) \text{ else } a = \mathtt{fault}$$

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

$$(s,h)[[\![C_1;C_2]\!]\eta]a \iff \Big(\exists (s',h').\,(s,h)\,[[\![C_1]\!]\eta]\,(s',h') \wedge (s',h')\,[[\![C_2]\!]\eta]\,a\Big) \vee \Big((s,h)\,[[\![C_1]\!]\eta]\,\mathtt{fault} \wedge a = \mathtt{fault}\Big)$$

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:

$$(s,h)\,[seq(c_1,c_2)]\,a \iff \Big(\exists (s',h').\,(s,h)\,[c_1]\,(s',h') \wedge (s',h')\,[c_2]\,a\Big) \vee \Big((s,h)\,[c_1]\,\mathtt{fault} \wedge a = \mathtt{fault}\Big)$$

$$(s,h)\,[b \rightsquigarrow c_1;c_2]\,a \iff \text{if } b(s) = \mathit{true} \text{ then } (s,h)[c_1]a \text{ else } (s,h)[c_2]a$$

$$(d_1,\ldots,d_n) = \mathit{fix}(\lambda d_1,\ldots,d_n \in \mathsf{LRel}^n.\,(F_1,...,F_n)) \qquad \text{(where } F_i = [\![C_i]\!]\eta[k_1 \mapsto d_1,\ldots,k_n \mapsto d_n])$$

Table 4. Selected Valuations

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

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.

> 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 of reading out the relevant cells in order to express spatial separation in traditional Hoare logic [4].

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

$$\mathit{listseg}(x,y) \quad \stackrel{\text{def}}{\iff} \quad (x = y \wedge \mathtt{emp}) \vee \\ \big(x \neq y \wedge \exists z.\ (x \mapsto z) * \mathit{listseg}(z,y)\big)$$

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. For this reason, the list segment predicate in [36] is not precise, if we wrap it in an existential quantifier over the sequence parameter.

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 3. *A predicate p is precise if and only if $p * -$ distributes over $\wedge$:*

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

We also have closure properties of precise predicates.

LEMMA 4. *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.*

10 Soundness

All of the proof rules from Section 3.2 are sound in the denotational model. The main result of the paper concerns the hypothetical frame rule and, by implication, the modular procedure rule.

THEOREM 5.

(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.*

Theorem 5(a) addresses point 1 from Section 6: it rules out the precondition $0 \vee 1$ in the conundrum, which is not precise. Theorem 5(b) addresses point 2: it rules out the invariant `true`, which is not precise. And this result covers the queue and memory manager examples, where the preconditions and invariants are all precise.

There are two main concepts used in the proof of the theorem.

- **The greatest relation**. We identify the greatest relation for a specification $\{p\}k\{q\}[X]$, which is the largest local relation satisfying it. 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.

In the next two subsections we define these concepts and state some of their properties, and then we sketch their relevance in the proof of the theorem.

10.1 The Greatest Relation

For each specification $\{p\} - \{q\}[X]$, define $\mathsf{great}(p,q,X)$

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

The first equivalence says that $\mathsf{great}(p,q,X)$ is safe at (s,h) just when p holds in (s,h_p) for some subheap h_p of h. The second equivalence is about state changes. The condition (1) means that $\mathsf{great}(p,q,X)$ can modify only those variables in X. Condition (2) says that $\mathsf{great}(p,q,X)$ 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 6. *The relation* $\mathsf{great}(p,q,X)$ *is in* LRel, *and satisfies* $\{p\} - \{q\}$ *and* $\mathsf{modifies}(-,X)$. *Moreover, it is the greatest such: for all local relations* c *in* LRel, *we have that* $\{p\}c\{q\} \wedge \mathsf{modifies}(c,X) \implies c \subseteq \mathsf{great}(p,q,X)$.

The *greatest* environment for a context Γ is the largest environment, in the pointwise order, satisfying all the procedure specifications in Γ. It maps k to $\mathsf{great}(p,q,X)$ when $\{p\}k\{q\}[X] \in \Gamma$; otherwise, it maps k to the top relation $\mathsf{States} \times (\mathsf{States} \cup \{\mathtt{fault}\})$. Greatest environments give us a simpler way to interpret sequents and proof rules. A sequent $\Gamma \vdash \{p\}C\{q\}$ holds just if the triple $\{p\}(\llbracket C \rrbracket \eta)\{q\}$ holds for the greatest environment η satisfying Γ, leading to

PROPOSITION 7. *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 $\{p\}\llbracket C \rrbracket \eta \{q\} \implies \{p'\}\llbracket C \rrbracket \eta' \{q'\}$ *for the greatest environments η and η' that, respectively, satisfy Γ and Γ'.*

10.2 Simulation

Let $R : \mathsf{States} \leftrightarrow \mathsf{States}$ be a binary relation between states. For c, c_1 in LRel, we say that c_1 *simulates* c upto R, denoted $c[\mathsf{sim}(R)]c_1$, just if the following properties hold:

- Generalized Safety Monotonicity: if $(s,h)[R](s_1,h_1)$, and c is safe at (s,h), then c_1 is safe at (s_1,h_1).
- Generalized Frame Property: if $(s,h)[R](s_1,h_1)$, c is safe at (s,h), and $(s_1,h_1)[c_1](s'_1,h'_1)$, then there is a state (s',h') such that $(s,h)[c](s',h')$ and $(s',h')[R](s'_1,h'_1)$.

$c[\mathsf{sim}(R)]c_1$ says that for R-related initial states (s,h) and (s_1,h_1), when we have enough resources at (s,h) to run c safely, we also have enough resources at (s_1,h_1) to run c_1 safely; and in that case, every state transition from (s_1,h_1) in c_1 can be tracked by a transition from (s,h) in c.

Suppose r is a predicate. The following relation R_r plays a central role in the analysis of the hypothetical frame rule.

$$(s,h)[R_r](s_1,h_1) \stackrel{\text{def}}{\iff} s = s_1 \wedge \exists h_r . h_1 = h * h_r \wedge (s,h_r) \in r$$

The next result gives us a way to connect hypotheses in the premise and conclusion of the hypothetical rule, and additionally provides the characterization of precise predicates that is at the core of Theorem 5.

PROPOSITION 8.

(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.

The detailed proof of Theorem 5 relies on developing machinery that allows us to apply this key proposition; this development, and the proof of the proposition itself, is nontrivial, and will be left to the full paper. However, the relevance of the proposition can be seen by considering a special case of the hypothetical frame rule, for the key case of procedure call, and where the modified variables are held fixed:

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

We give a proof of the following proposition about this special case; it is the central step for showing Theorem 5.

PROPOSITION 9.

(a) *The above special case of the hypothetical frame rule is sound for a fixed precondition p_1 if and only if p_1 is precise.*

(b) *The above special case is sound for a fixed invariant r if and only if r is precise.*

Note that the if direction of the proposition is implied by the same direction of Theorem 5, and that for the only-if direction, the proposition implies the theorem. The proof of this proposition uses a lemma that characterizes $\mathsf{sim}(R_r)$ using Hoare triples.

LEMMA 10. *Local relations c and c_1 are related by $\mathsf{sim}(R_r)$ if and only if for all predicates p, q, we have*

$$\{p\}c\{q\} \implies \{p * r\}c_1\{q * r\}.$$

Proof: [of Proposition 9]

Using Proposition 7 and Lemma 10, we can simplify the above special case of the hypothetical frame rule as follows:

for all predicates p, q,
if $\{p_1\}k_1\{q_1\}[X_1] \vdash \{p\}k\{q\}$ holds,
then $\{p_1 * r\}k_1\{q_1 * r\}[X_1] \vdash \{p * r\}k\{q * r\}$ holds

$\iff (\because$ Proposition 7)

for all predicates p, q,
if $\{p\}\mathsf{great}(p_1, q_1, X_1)\{q\}$ holds,
then $\{p * r\}\mathsf{great}(p_1 * r, q_1 * r, X_1)\{q * r\}$ holds

$\iff (\because$ Lemma 10)

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

Now, Proposition 8(a) gives (a) of this proposition, and Proposition 8(b) gives (b) of this proposition. ∎

10.3 Supported and Intuitionistic Predicates

There is a relaxation of the notion of precise predicate that can be used to provide further sufficient conditions for soundness. A predicate is *supported* if, for any stack and heap, the collection of subheaps making it true (while holding the stack constant) is empty or has a least element. A predicate is *intuitionistic* if it is closed under heap extension.

THEOREM 11. *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 5(a), because it mentions postconditions in addition to preconditions. Likewise, the second point does not contradict Theorem 5(b), because it mentions postconditions as well as the resource invariant. This result about supported predicates would give us a version of the hypothetical frame rule appropriate when we are not interested in nailing down definite portions of memory using assertions, as might be the case in a garbage-collected language.

11 Related and Future Work

As we have emphasized, reliance on fixed resource partitioning has been an obstacle to the development of modular methods of program specification that are applicable to widely used programming languages. Because the separating conjunction $*$ is a logical connective, which depends on the state, it allows us to describe situations where the partition between a module and its clients changes over time. For example, with a resource manager module the resources transfer back and forth between the module and a client, as allocation and deallocation operations are performed, but correct operating relies on separation being maintained at all times.

A different reaction to the limitations of fixed partitioning has been the development of the assume-guarantee method of reasoning about program components [24, 21]. While this has proven successful, we are unsure whether it could be profitably applied to mutable data structures with embedded pointers. In any case, when partitioning can be ensured, be it fixed or dynamic, an invariant-based methodology leads to pleasantly modular specifications and proofs.

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 [23] (also, [12]). They use abstract variables (like our use of the variable Q in Table 3) 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 at all 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 (e.g., [19, 10, 15]). 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.

The advantage of confinement schemes is their use of static typing, or static analysis, to provide algorithmic guarantees of information hiding properties. Conversely, separation logic is more flexible, not just because it is based on logic rather than types, but also because it allows any number of pointers from the outside, requiring only that these pointers not be dereferenced without permission. Current confinement schemes have difficulty with ownership transfer that involves aliasing (because they tend to rely on "unique" pointers), such as a program that disposes all of the elements in a graph, or with examples where resource partitioning depends on arithmetic properties.

Recent work on the semantics of confinement uses heap partitioning in an essential way [3, 33], thus suggesting the prospect of a deeper connection between type systems for confinement and logics of separation. There is also the possibility of promoting the heap partitioning operation $*$ used in the semantic models to a type operator in the source language; an immediate step could even be attempted to obtain a form of alias types with information hiding [37]. Further unification along these lines could be valuable.

It is striking that many proof systems for object-oriented languages work by exposing class invariants or other descriptions of internal states at method call sites (e.g., [13, 34]). Of course, the developers of such systems have rightly been careful, as unsoundness can very easily result if one incorrectly hides invariants. It seems plausible, however, that taking explicit account of separation or confinement could lead to an improved logic of objects.

Further afield in aims, but closer in technique, are logics of mobile and concurrent processes that have been developed by Cardelli, Caires and Gordon [9, 5]; related ideas have also been used to study semi-structured data [8, 7]. Cardelli et. al. use subsets of a commutative monoid, as in the general models of bunched logic [31, 32], but the interaction between logic and program dynamics is very different to that here. The models of [9, 5] do not satisfy the properties (such as the frame property) that drive our approach to information hiding. Furthermore, although the "pointers from outside" phenomenon certainly occurs in their setting, based as it is on the π-calculus, they do not use the conjunction $*$ (or $|$ in their notation) to control these pointers/names; rather, they employ a form of new name quantifier, following Gabbay and Pitts [14]. Despite the surface similarity of logical structure, we do not feel that we fully understand the relationship between the two approaches.

An intriguing question is if there is a link between the hypothetical frame rule and the data abstraction provided by polymorphic types. Polymorphic typing can be used to hide the type in a data abstraction [35, 25], but this is not the same thing as hiding dynamic resources. For example, if we hide the type of a reference, polymorphic typing does not guarantee that the reference is not aliased, and accessible from outside the data abstraction. Still, the hypothetical frame rule ensures that a proof of client code can be used with any (precise) representation invariant, so there is an obvious intuitive analogy with polymorphic functions; the client should be parametrically polymorphic in the resource invariant. If this analogy can be made precise it may provide the basis for an approach to data refinement, and encapsulated components as first-class values, that accounts for mutable structures and dynamic ownership transfer.

We have stayed in a sequential setup in this paper, but the ideas seem relevant to concurrent programming. Indeed, the treatment by Hoare in [17] revolves around the concept of spatial separation, and the work here grew out of an attempt to directly adapt that approach, and its extension by Owicki and Gries [27], to separation logic. 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.

Work is underway on the semantics of the concurrent logic, and we are hopeful that a thorough account of the concurrency proof rules will be forthcoming.

Finally, in this paper we have concentrated on program proving, but there have been striking successes in software model checking [2, 11], which use abstraction to bridge the gap between infinite state language models and the finite state models expected by model checking algorithms; in essence, abstract interpretations are chosen, and sometimes refined, that allow for the synthesis of loop invariants. We wonder if one might synthesize resource invariants describing heap-sensitive properties as well, and use this to partition the model checking effort. For this to be workable the immediate challenge is to devise expressive heap abstractions [38] that are compatible with an information-hiding rule like the hypothetical frame rule.

Acknowledgements. We have benefitted greatly from discussions with Josh Berdine, Richard Bornat and Cristiano Calcagno. O'Hearn's research was supported by the EPSRC project "Local Reasoning about State". Reynolds's research was partially supported by an EPSRC Visiting Fellowship at Queen Mary, University of London, 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 grant No. R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation. Yang would like to thank EPSRC for supporting his visit to University of London in 2002, during which he first got involved in this work.

12 References

[1] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *18th LICS*, 2003.

[2] T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. *Proceedings of the Workshop on Advances in Verification*, 2000.

[3] A. Banerjee and D. Naumann. Representation independence, confinement and access control. In *29th POPL*, 2002.

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

[5] L. Cardelli and L Caires. A spatial logic for concurrency. In TACS'01, LNCS 2255:1–37, Springer, 2001.

[6] L. Cardelli, P. Gardner, and G. Ghelli. Querying trees with pointers. Unpublished notes, 2003.

[7] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. *Proceedings of ICALP'02*.

[8] L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. *Proceedings of ESOP'01*.

[9] L. Cardelli and A. D. Gordon. Anytime, anywhere. Modal logics for mobile ambients. In *27th POPL*, 2000.

[10] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. ECOOP, LNCS 2072, 2001.

[11] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[12] K.R.M. Leino D. Detlefs and G. Nelson. Wrestling with rep exposure. Digital SRC Research Report 156, 1998.

[13] F. de Boer. A WP calculus for OO. In *FOSSACS*, 1999.

[14] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *14th LICS*, 1999.

[15] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. *OOPSLA*, pp241–253, 2001.

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

[17] C.A.R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques*. Academic Press, 1972.

[18] C.A.R. Hoare. Monitors: An operating system structuring concept. *C.ACM*, 17(10):549–557, October 1974.

[19] J. Hogg, D. Lea, R. Holt, A. Wills, and D. de Champeaux. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, April, 1992.

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

[21] C.B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.

[22] B. Kernighan and D. Ritchie. *The C programming language*. Prentice Hall, 1988.

[23] K.R.M. Leino and G. Nelson. Data abstraction and information hiding. ACM TOPLAS 24(5): 491-553, 2002.

[24] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions of Software Engineering*, July, 1981.

[25] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. ACM TOPLAS 10(3):470-502, 1988.

[26] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of Computer Science Logic*, pages 1–19, 2001. LNCS 2142.

[27] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[28] D.L. Parnas. Information distribution aspects of design methodology. *IFIP Congress (1) 1971:* 339-344, 1972.

[29] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *C.ACM*, 15(12):1053–1058, 1972.

[30] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[31] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications.* Kluwer Applied Logic series, 2002.
[32] D.J. Pym, P.W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *TCS*, to appear, 2003.
[33] U. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Proceedings of ESOP*, 2003.
[34] B. Reus, M. Wirsing, and R. Hennicker. A Hoare calculus for verifying Java realizations of OCL-constrained design models. *FASE Proceedings*, LNCS 2029, pp300–316, 2001.
[35] J.C. Reynolds. Types, abstraction and parametric polymorphism. *IFIP Proceedings*, pp513-523, 1983.
[36] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, *17th LICS*, 2002.
[37] D. Walker and J.G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, pp177–206, 2000.
[38] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Compiler Construction*, pp1–17, 2000.
[39] H. Yang, and P. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS'02*, pp402–416, 2002.

13 Appendix: Variable Conditions

13.1 Side Conditions and Modifies Sets

We now clarify the side conditions for the hypothetical frame rule. To begin with, 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 context. 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 context Γ if for all $\{p\}k\{q\}[Y]$ in Γ, X is disjoint from p, q and Y. 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 $\mathit{Modifies}(C)(\Gamma;\Gamma')$ 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.

$$\begin{array}{lcll} \mathit{Modifies}(k)(\Gamma;\Gamma') & = & X, & \text{if } \{p\}k\{q\}[X] \in \Gamma \\ \mathit{Modifies}(k)(\Gamma;\Gamma') & = & \{\}, & \text{if } \{p\}k\{q\}[X] \in \Gamma' \end{array}$$

The upshot is that $\mathit{Modifies}(C)(\Gamma;\Gamma')$ reports those variables modified by C, except that it doesn't count any procedure calls for procedures 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 $\mathit{Modifies}(C)(\Gamma;\Gamma')$ is the union $\cup_i \mathit{Modifies}(C_i)(\Gamma;\Gamma')$. Two of the basic commands are as follows:

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

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

$$\mathit{Modifies}(C)(\Gamma;\, \{p_1\}k_1\{q_1\}[X_1],\ldots,\{p_n\}k_n\{q_n\}[X_n])$$
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' to be empty in $\mathit{Modifies}(C)(\Gamma;\Gamma')$.

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 abstract variables to specify the behaviour of a module, as exemplified by the treatment of the abstract variable Q in the queue module in Table 3.

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 abstract 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. If some other approach could be developed as an alternative to the changing abstract variables, that was itself not more complex, then perhaps we could finally do away with modifies conditions.

In situations where one wants to capture only "structural integrity" properties of data structures, rather than correctness properties, it is often possible to avoid abstract variables. For example, one sometimes wants to ensure, for example, that a data structure has the correct shape and has no dangling pointers, without giving a complete description of the data that is represented. Because abstract variables are not required (or less often required) in such situations we might get some way with a logic simpler than the one here, that does not require modifies clauses.

13.2 On Existentials and Free Variables

In [26, 36] 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.