# Local Reasoning about Programs that Alter Data Structures

Peter O'Hearn[1], John Reynolds[2], and Hongseok Yang[3]

[1] Queen Mary, University of London
[2] Carnegie Mellon University
[3] University of Birmingham and University of Illinois at Urbana-Champaign

**Abstract.** We describe an extension of Hoare's logic for reasoning about programs that alter data structures. We consider a low-level storage model based on a heap with associated lookup, update, allocation and deallocation operations, and unrestricted address arithmetic. The assertion language is based on a possible worlds model of the logic of bunched implications, and includes spatial conjunction and implication connectives alongside those of classical logic. Heap operations are axiomatized using what we call the "small axioms", each of which mentions only those cells accessed by a particular command. Through these and a number of examples we show that the formalism supports local reasoning: A specification and proof can concentrate on only those cells in memory that a program accesses.

This paper builds on earlier work by Burstall, Reynolds, Ishtiaq and O'Hearn on reasoning about data structures.

## 1 Introduction

Pointers have been a persistent trouble area in program proving. The main difficulty is not one of finding an in-principle adequate axiomatization of pointer operations; rather there is a mismatch between simple intuitions about the way that pointer operations work and the complexity of their axiomatic treatments. For example, pointer assignment is operationally simple, but when there is aliasing, arising from several pointers to a given cell, then an alteration to that cell may affect the values of many syntactically unrelated expressions. (See [20, 2, 4, 6] for discussion and references to the literature on reasoning about pointers.)

We suggest that the source of this mismatch is the global view of state taken in most formalisms for reasoning about pointers. In contrast, programmers reason informally in a local way. Data structure algorithms typically work by applying local surgeries that rearrange small parts of a data structure, such as rotating a small part of a tree or inserting a node into a list. Informal reasoning usually concentrates on the effects of these surgeries, without picturing the entire memory of a system. We summarize this local reasoning viewpoint as follows.

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.

Local reasoning is intimately tied to the complexity of specifications. Often, a program works with a circumscribed collection of resources, and it stands to reason that a specification should concentrate on just those resources that a program accesses. For example, a program that inserts an element into a linked list need know only about the cells in that list; there is no need (intuitively) to keep track of all other cells in memory when reasoning about the program.

The central idea of the approach studied in this paper is of a "spatial conjunction" $P * Q$, that asserts that $P$ and $Q$ hold for separate parts of a data structure. The conjunction provides a way to compose assertions that refer to different areas of memory, while retaining disjointness information for each of the conjuncts. The locality that this provides can be seen both on the level of atomic heap assignments and the level of compound operations or procedures. When an alteration to a single heap cell affects $P$ in $P * Q$, then we know that it will not affect $Q$; this gives us a way to short-circuit the need to check for potential aliases in $Q$. On a larger scale, a specification $\{P\}C\{Q\}$ of a heap surgery can be extended using a rule that lets us infer $\{P * R\}C\{Q * R\}$, which expresses that additional heap cells remain unaltered. This enables the initial specification $\{P\}C\{Q\}$ to concentrate on only the cells in the program's footprint.

The basic idea of the spatial conjunction is implicit in early work of Burstall [3]. It was explicitly described by Reynolds in lectures in the fall of 1999; then an intuitionistic logic based on this idea was discovered independently by Reynolds [20] and by Ishtiaq and O'Hearn [7] (who also introduced a spatial implication $P \mathrel{-\!\!*} Q$, based on the logic BI of bunched implications [11, 17]). In addition, Ishtiaq and O'Hearn devised a classical version of the logic that is more expressive than the intuitionistic version. In particular, it can express storage deallocation.

Subsequently, Reynolds extended the classical version by adding pointer arithmetic. This extension results in a model that is simpler and more general than our previous models, and opens up the possibility of verifying a wider range of low-level programs, including many whose properties are difficult to capture using type systems. Meanwhile, O'Hearn fleshed out the theme of local reasoning sketched in [7], and he and Yang developed a streamlined presentation of the logic based on what we call the "small axioms".

In this joint paper we present the pointer arithmetic model and assertion language, with the streamlined Hoare logic. We illustrate the formalism using programs that work with a space-saving representation of doubly-linked lists, and a program that copies a tree.

Two points are worth stressing before continuing. First, by *local* we do not merely mean *compositional* reasoning: It is perfectly possible to be compositional and global (in the state) at the same time, as was the case in early denotational models of imperative languages. Second, some aspects of this work bear a strong similarity to semantic models of local state [19, 15, 16, 13, 12]. In particular, the conjunction $*$ is related to interpretations of syntactic control of interference [18, 10, 12], and the Frame Rule described in Section 3 was inspired by the idea of the expansion of a command from [19, 15]. Nevertheless, local reasoning about state is not the same thing as reasoning about local state: We are proposing here that specifications and reasoning themselves be kept confined, and this is an issue whether or not we consider programming facilities for *hiding* state.

## 2 The Model and Assertion Language

The model has two components, the store and the heap. The store is a finite partial function mapping from variables to integers. The heap is indexed by a subset `Locations` of the integers, and is accessed using indirect addressing $[E]$ where $E$ is an arithmetic expression.

$$\begin{array}{ll}
\mathtt{Ints} \overset{\Delta}{=} \{..., -1, 0, 1, ...\} & \mathtt{Variables} \overset{\Delta}{=} \{x, y, ...\} \\
\mathtt{Atoms}, \mathtt{Locations} \subseteq \mathtt{Ints} & \mathtt{Locations} \cap \mathtt{Atoms} = \{\},\ \mathtt{nil} \in \mathtt{Atoms} \\
\mathtt{Stores} \overset{\Delta}{=} \mathtt{Variables} \rightharpoonup_{fin} \mathtt{Ints} & \mathtt{Heaps} \overset{\Delta}{=} \mathtt{Locations} \rightharpoonup_{fin} \mathtt{Ints} \\
\mathtt{States} \overset{\Delta}{=} \mathtt{Stores} \times \mathtt{Heaps} &
\end{array}$$

In order for allocation to always succeed, we place a requirement on the set `Locations`: For any positive integer $n$, there are infinitely many sequences of length $n$ of consecutive integers in `Locations`. This requirement is satisfied if we take `Locations` to be the non-negative integers. (In several example formulae, we will implicitly rely on this choice.) Then we could take `Atoms` to be the negative integers, and `nil` to be $-1$.

Integer and boolean expressions are determined by valuations

$$[\![E]\!]s \in \mathtt{Ints} \qquad [\![B]\!]s \in \{true, false\}$$

where the domain of $s \in$ `Stores` includes the free variables of $E$ or $B$. The grammars for expressions are as follows.

$$\begin{aligned}
E, F, G &::= x, y, ... \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F \\
B &::= \mathtt{false} \mid B \Rightarrow B \mid E = F \mid E < F \mid \mathtt{isatom?}(E) \mid \mathtt{isloc?}(E)
\end{aligned}$$

The expressions $\mathtt{isatom?}(E)$ and $\mathtt{isloc?}(E)$ test whether $E$ is an atom or location.

The assertions include all of the boolean expressions, the points-to relation $E \mapsto F$, all of classical logic, and the spatial connectives `emp`, $*$ and $-\!\!*$.

$$\begin{array}{lll}
P, Q, R ::= & B \mid E \mapsto F & \text{Atomic Formulae} \\
\mid & \mathtt{false} \mid P \Rightarrow Q \mid \forall x.P & \text{Classical Logic} \\
\mid & \mathtt{emp} \mid P * Q \mid P -\!\!* Q & \text{Spatial Connectives}
\end{array}$$

Various other connectives are defined as usual: $\neg P = P \Rightarrow \mathtt{false}$; $\mathtt{true} = \neg(\mathtt{false})$; $P \vee Q = (\neg P) \Rightarrow Q$; $P \wedge Q = \neg(\neg P \vee \neg Q)$; $\exists x.\ P = \neg \forall x. \neg P$.

We use the following notations in the semantics of assertions.

1. $dom(h)$ denotes the domain of definition of a heap $h \in$ `Heaps`, and $dom(s)$ is the domain of $s \in$ `Stores`;

2. $h \# h'$ indicates that the domains of $h$ and $h'$ are disjoint;
3. $h * h'$ denotes the union of disjoint heaps (i.e., the union of functions with disjoint domains);
4. $(f \mid i \mapsto j)$ is the partial function like $f$ except that $i$ goes to $j$. This notation is used both when $i$ is and is not in the domain of $f$.

We define a satisfaction judgement $s, h \models P$ which says that an assertion holds for a given store and heap. (This assumes that Free$(P) \subseteq dom(s)$, where Free$(P)$ is the set of variables occurring freely in $P$.)

$$
\begin{array}{lll}
s, h \models B & \text{iff} & [\![B]\!]s = true \\
s, h \models E \mapsto F & \text{iff} & \{[\![E]\!]s\} = dom(h) \text{ and } h([\![E]\!]s) = [\![F]\!]s \\
s, h \models \mathtt{false} & & \text{never} \\
s, h \models P \Rightarrow Q & \text{iff} & \text{if } s, h \models P \text{ then } s, h \models Q \\
s, h \models \forall x.P & \text{iff} & \forall v \in \mathtt{Ints}.\, [s \mid x \mapsto v], h \models P \\
s, h \models \mathtt{emp} & \text{iff} & h = [\,] \text{ is the empty heap} \\
s, h \models P * Q & \text{iff} & \exists h_0, h_1.\, h_0 \# h_1,\ h_0 * h_1 = h,\ s, h_0 \models P \text{ and } s, h_1 \models Q \\
s, h \models P \mathrel{-\!\!*} Q & \text{iff} & \forall h'.\, \text{if } h' \# h \text{ and } s, h' \models P \text{ then } s, h * h' \models Q
\end{array}
$$

Notice that the semantics of $E \mapsto F$ is "exact", where it is required that $E$ is the only active address in the current heap. Using $*$ we can build up descriptions of larger heaps. For example, $(10 \mapsto 3) * (11 \mapsto 10)$ describes two adjacent cells whose contents are 3 and 10.

On the other hand, $E = F$ is completely heap independent (like all boolean and integer expressions). As a consequence, a conjunction $(E = F) * P$ is true just when $E = F$ holds in the current store and when $P$ holds for the same store and some heap contained in the current one.

It will be convenient to have syntactic sugar for describing adjacent cells, and for an exact form of equality. We also have sugar for when $E$ is an active address.

$$
\begin{array}{lll}
E \mapsto F_0, ..., F_n & \overset{\Delta}{=} & (E \mapsto F_0) * \cdots * (E + n \mapsto F_n) \\
E \doteq F & \overset{\Delta}{=} & (E = F) \wedge \mathtt{emp} \\
E \mapsto - & \overset{\Delta}{=} & \exists y. E \mapsto y \qquad\qquad (y \notin \text{Free}(E))
\end{array}
$$

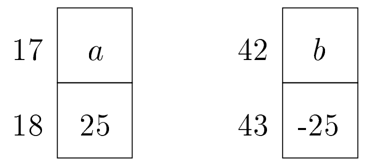
A characteristic property of $\doteq$ is the way it interacts with $*$:

$$(E \doteq F) * P \quad \Leftrightarrow \quad (E = F) \wedge P.$$

As an example of adjacency, consider an "offset list", where the next node in a linked list is obtained by adding an offset to the position of the current node. Then the formula

$$(x \mapsto a, o) * (x + o \mapsto b, -o)$$

describes a two-element, circular, offset list that contains $a$ and $b$ in its head fields and offsets in its link fields. For example, in a store where $x = 17$ and $o = 25$, the formula is true of a heap



The semantics in this section is a model of (the Boolean version of) the logic of bunched implications [11, 17]. This means that the model validates all the laws of classical logic, commutative monoid laws for `emp` and $*$, and the "parallel rule" for $*$ and "adjunction rules" for $-\!\!*$.

$$\frac{P \Rightarrow Q \quad R \Rightarrow S}{P * R \Rightarrow Q * S}$$

$$\frac{P * R \Rightarrow S}{P \Rightarrow R -\!\!* S} \qquad \frac{P \Rightarrow R -\!\!* S \quad Q \Rightarrow R}{P * Q \Rightarrow S}$$

Other facts, true in the specific model, include

$$\big((E \mapsto F) * (E' \mapsto F') * \texttt{true}\big) \Rightarrow E \neq E' \qquad \texttt{emp} \Leftrightarrow \forall x.\, \neg(x \mapsto - * \texttt{true})$$

See [21] for a fuller list.

## 3 The Core System

In this section we present the core system, which consists of axioms for commands that alter the state as well as a number of inference rules. We will describe the meanings for the various commands informally, as each axiom is discussed.

There is one axiom for each of four atomic commands. We emphasize that the right-hand side of := is *not* an expression occurring in the forms $x := [E]$ and $x := \texttt{cons}(E_1, ..., E_k)$; $[\cdot]$ and `cons` do not appear within expressions. Only $x := E$ is a traditional assignment, and it is the only atomic command that can be described by Hoare's assignment axiom. In the axioms $x, m, n$ are assumed to be distinct variables.

The Small Axioms

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

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

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

$$\{x \doteq n\}\, x := E\, \{x \doteq (E[n/x])\}$$

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

The Structural Rules

Frame Rule

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\ \mathrm{Modifies}(C) \cap \mathrm{Free}(R) = \{\}$$

Auxiliary Variable Elimination

$$\frac{\{P\}\,C\,\{Q\}}{\{\exists x.P\}\,C\,\{\exists x.Q\}}\ x \not\in \mathrm{Free}(C)$$

Variable Substitution

$$\frac{\{P\}\,C\,\{Q\}}{(\{P\}\,C\,\{Q\})[E_1/x_1,...,E_k/x_k]}\ \begin{array}{l}\{x_1,...,x_k\} \supseteq \mathrm{Free}(P,C,Q), \text{ and} \\ x_i \in \mathrm{Modifies}(C) \text{ implies} \\ E_i \text{ is a variable not free in any other } E_j\end{array}$$

Rule of Consequence

$$\frac{P' \Rightarrow P \qquad \{P\}\,C\,\{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\,C\,\{Q'\}}$$

The first small axiom just says that if $E$ points to something beforehand (so it is active), then it points to $F$ afterwards, and it says this for a small portion of the state in which $E$ is the only active cell. This corresponds to the operational idea of $[E] := F$ as a command that stores the value of $F$ at address $E$ in the heap. The axiom also implicitly says that the command does not alter any variables; this is covered by our definition of its Modifies set below.

The $\mathtt{dispose}(E)$ instruction deallocates the cell at address $E$. In the post-condition for the `dispose` axiom `emp` is a formula which says that the heap is empty (no addresses are active). So, the axiom states that if $E$ is the sole active address and it is disposed, then in the resulting state there will be no active addresses. Here, the exact points-to relation is necessary, in order to be able to conclude `emp` on termination.

The $x := \mathtt{cons}(E_1, ..., E_k)$ command allocates a contiguous segment of $k$ cells, initialized to the values of $E_1, ..., E_k$, and places in $x$ the address of the first cell from the segment. The precondition of the axiom uses the exact equality, which implies that the heap is empty. The axiom says that if we begin with the empty heap and a store where $x = m$, we will obtain $k$ contiguous cells with appropriate values. The variable $m$ in this axiom is used to record the value of $x$ before the command is executed.

We only get fixed-length allocation from $x := \mathtt{cons}(E_1, ..., E_k)$. It is also possible to formulate an axiom for a command $x := \mathtt{alloc}(E)$ that allocates a segment of length $E$; see [21].

We have also included small axioms for the other two commands, but they are less important. These commands are not traditionally as problematic, because they do not involve heap alteration.

The small axioms are so named because each mentions only the area of heap accessed by the corresponding command. For $[E] := F$ and $x := [E]$ this is one cell, in the axioms for `dispose` or `cons` precisely those cells allocated or deallocated are mentioned, and in $x := E$ no heap cells are accessed.

The notion of free variable referred to in the structural rules is the standard one. Modifies$(C)$ is the set of variables that are assigned to within $C$. The Modifies set of each of $x := \mathtt{cons}(E_1, ..., E_k)$, $x := E$ and $x := [E]$ is $\{x\}$, while for $\mathtt{dispose}(E)$ and $[E] := F$ it is empty. Note that the Modifies set only tracks potential alterations to the store, and says nothing about the heap cells that might be modified.

In this paper we treat the Rule of Consequence semantically. That is, when the premisses $P' \Rightarrow P$ and $Q \Rightarrow Q'$ are true in the model for arbitrary store/heap pairs, we will use the rule without formally proving the premisses.

The Frame Rule codifies a notion of local behaviour. The idea is that the precondition in $\{P\}C\{Q\}$ specifies an area of storage, as well as a logical property, that is sufficient for $C$ to run and (if it terminates) establish postcondition $Q$. If we start execution with a state that has additional heap cells, beyond those described by $P$, then the values of the additional cells will remain unaltered. We use $*$ to separate out these additional cells. The invariant assertion $R$ is what McCarthy and Hayes called a "frame axiom" [9]. It describes cells that are not accessed, and hence not changed, by $C$.

As a warming-up example, using the Frame Rule we can prove that assigning to the first component of a binary cons cell does not affect the second component.

$$\dfrac{\dfrac{\{x \mapsto a\}\,[x] := b\,\{x \mapsto b\}}{\{(x \mapsto a) * (x+1 \mapsto c)\}\,[x] := b\,\{(x \mapsto b) * (x+1 \mapsto c)\}}\ \text{Frame}}{\{x \mapsto a, c\}\,[x] := b\,\{x \mapsto b, c\}}\ \text{Syntactic Sugar}$$

The overlap of free variables between $x + 1 \mapsto c$ and $[x] := b$ is allowed here because Modifies$([x] := b) = \{\}$.

## 4 Derived Laws

The small axioms are simple but not practical. Rather, they represent a kind of thought experiment, an extreme take on the idea that a specification can concentrate on just those cells that a program accesses.

In this section we show how the structural rules can be used to obtain a number of more convenient derived laws (most of which were taken as primitive in [20, 7]). Although we will not explicitly state a completeness result, along the way we will observe that weakest preconditions or strongest postconditions are derivable for each of the individual commands. This shows a sense in which nothing is missing in the core system, and justifies the claim that each small axiom gives enough information to understand how its command works.

We begin with $[E] := F$. If we consider an arbitrary invariant $R$ then we obtain the following derived axiom using the Frame Rule with the small axiom as its premise.

$$\{(E \mapsto -) * R\}\,[E] := F\,\{(E \mapsto F) * R\}$$

This axiom expresses a kind of locality: Assignment to $[E]$ affects the heap cell at position $E$ only, and so cannot affect the assertion $R$. In particular, there is no need to generate alias checks within $R$. With several more steps of Auxiliary Variable Elimination we can obtain an axiom that is essentially the one from [20]:

$$\{\exists x_1, \cdots, x_n.\,(E \mapsto -) * R\}\,[E] := F\,\{\exists x_1, \cdots, x_n.\,(E \mapsto F) * R\}$$

where $x_1, ..., x_n \notin \mathrm{Free}(E, F)$.

For allocation, suppose $x \notin \mathrm{Free}(E_1, ..., E_k)$. Then a simpler version of the small axiom is

$$\{\mathtt{emp}\}x := \mathtt{cons}(E_1, ..., E_k)\{x \mapsto E_1, ..., E_k\}$$

This can be derived using rules for auxiliary variables and Consequence. If, further, $R$ is an assertion where $x \notin \mathrm{Free}(R)$ then

$$\dfrac{\dfrac{\{\mathtt{emp}\}\,x := \mathtt{cons}(E_1, ..., E_k)\,\{x \mapsto E_1, ..., E_k\}}{\{\mathtt{emp} * R\}\,x := \mathtt{cons}(E_1, ..., E_k)\,\{(x \mapsto E_1, ..., E_k) * R\}}\ \text{Frame}}{\{R\}\,x := \mathtt{cons}(E_1, ..., E_k)\,\{(x \mapsto E_1, ..., E_k) * R\}}\ \text{Consequence}$$

The conclusion is the strongest postcondition, and a variant involving auxiliary variables handles the case when $x \in \mathrm{Free}(R, E_1, ..., E_k)$.

As an example of the use of these laws, recall the assertion $(x \mapsto a, o) * (x+o \mapsto b, -o)$ that describes a circular offset-list. Here is a proof outline for a sequence of commands that creates such a structure.

$$\begin{array}{l}
\{\mathtt{emp}\}\\
\ \ x := \mathtt{cons}(a, a)\\
\{x \mapsto a, a\}\\
\ \ t := \mathtt{cons}(b, b)\\
\{(x \mapsto a, a) * (t \mapsto b, b)\}\\
\ \ [x + 1] := t - x\\
\{(x \mapsto a, t - x) * (t \mapsto b, b)\}\\
\ \ [t + 1] := x - t\\
\{(x \mapsto a, t - x) * (t \mapsto b, x - t)\}\\
\{\exists o.\,(x \mapsto a, o) * (x + o \mapsto b, -o)\}
\end{array}$$

The last step, which is an instance of the Rule of Consequence, uses $t - x$ as the witness for $o$. Notice how the alterations in the last two commands are done locally. For example, because of the placement of $*$ we know that $x + 1$ must be different from $t$ and $t + 1$, so the assignment $[x + 1] := t - x$ cannot affect the $t \mapsto b, b$ conjunct.

If we wish to reason backwards, then $-\!*$ can be used to express weakest preconditions. Given an arbitrary postcondition $Q$, choosing $(E \mapsto F) -\!* Q$ as the invariant gives a valid precondition for $[E] := F$

$$\cfrac{\cfrac{\{E \mapsto -\}\, [E] := F\, \{E \mapsto F\}}{\{(E \mapsto -) * ((E \mapsto F) -\!* Q)\}\, [E] := F\, \{(E \mapsto F) * ((E \mapsto F) -\!* Q)\}}\ \text{Frame}}{\{(E \mapsto -) * ((E \mapsto F) -\!* Q)\}\, [E] := F\, \{Q\}}\ \text{Consequence}$$

The Consequence step uses an adjunction rule for $*$ and $-\!*$. The precondition obtained is in fact the weakest: it expresses the "update as deletion followed by extension" idea explained in [7]. The weakest precondition for allocation can also be expressed with $-\!*$.

The weakest precondition for `dispose` can be computed directly, because the Modifies set of $\texttt{dispose}(E)$ is empty.

$$\cfrac{\cfrac{\{E \mapsto -\}\, \texttt{dispose}(E)\, \{\texttt{emp}\}}{\{(E \mapsto -) * R\}\, \texttt{dispose}(E)\, \{\texttt{emp} * R\}}\ \text{Frame}}{\{(E \mapsto -) * R\}\, \texttt{dispose}(E)\, \{R\}}\ \text{Consequence}$$

The conclusion is (a unary version of) the axiom for `dispose` from [7].

The weakest precondition axiom for $x := E$ is the usual one of Hoare. For $x := [E]$ is it similar, using $\exists$ to form a "let binder" (where $n \notin \text{Free}(E, P, x)$.

$$\{P[E/x]\}x := E\{P\}$$
$$\{\exists n.\, (\texttt{true} * E \mapsto n) \wedge P[n/x]\}x := [E]\{P\}$$

The formal derivations of these laws from the small axioms make heavy use of Variable Substitution and Auxiliary Variable Elimination; the details are contained in Yang's thesis [24].

Another useful derived law for $x := [E]$ is for the case when $x \notin \text{Free}(E, R)$, $y \notin \text{Free}(E)$, and when the precondition is of the form $(E \mapsto y) * R$. Then,

$$\{(E \mapsto y) * R\}\, x := [E]\, \{(E \mapsto x) * R[x/y]\}.$$

## 5 Beyond the Core

In the next few sections we give some examples of the formalism at work. In these examples we use sequencing, `if-then-else`, and a construct `newvar` for declaring a local variable. We can extend the core system with their usual Hoare logic rules.

$$\frac{\{P \wedge B\}\, C\, \{Q\} \quad \{P \wedge \neg B\}\, C'\, \{Q\}}{\{P\}\, \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C' \{Q\}} \qquad \frac{\{P\}\, C_1\, \{Q\} \quad \{Q\}\, C_2\, \{R\}}{\{P\}\, C_1; C_2\, \{R\}}$$

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\, \texttt{newvar}\, x.\, C\, \{Q\}}\ x \notin \text{Free}(P, Q)$$

We will also use simple first-order procedures. The procedure definitions we need will have the form

```
procedure $p(x_1, ..., x_n; y)$
$B$
```

where $x_1, ..., x_n$ are variables not changed in the body $B$ and $y$ is a variable that is assigned to. Procedure headers will always contain all of the variables occurring freely in a procedure body. Accordingly, we define

$$\begin{aligned} \text{Modifies}(p(x_1, ..., x_n; y)) &= \{y\} \\ \text{Free}(p(x_1, ..., x_n; y)) &= \{x_1, ..., x_n, y\}. \end{aligned}$$

We will need these clauses when applying the structural rules. In the examples the calling mechanism can be taken to be either by-name for all the parameters, or by-value on the $x_i$'s and by-reference on $y$.

Procedures are used in Section 7 mainly to help structure the presentation, but in Section 6 we also use recursive calls. There we appeal to the standard partial correctness rule which allows us to use the specification we are trying to prove as an assumption when reasoning about the body [5].

Our treatment in what follows will not be completely formal. We will continue to use the Rule of Consequence in a semantic way, and we will make inductive definitions without formally defining their semantics. Also, as is common, we will present program specifications annotated with intermediate assertions, rather than give step-by-step proofs.

## 6 Tree Copy

In this section we consider a procedure for copying a tree. The purpose of the example is to show the Frame Rule in action.

For our purposes a tree will either be an atom $a$ or a pair $(\tau_1, \tau_2)$ of trees. Here is an inductive definition of a predicate $\texttt{tree}\,\tau\, i$ which says when a number $i$ represents a tree $\tau$.

$$\begin{aligned} \texttt{tree}\, a\, i &\overset{\Delta}{\Longleftrightarrow} i = a \land \texttt{isatom?}(a) \land \texttt{emp} \\ \texttt{tree}\,(\tau_1, \tau_2)\, i &\overset{\Delta}{\Longleftrightarrow} \exists x, y. (i \mapsto x, y) * (\texttt{tree}\,\tau_1\, x * \texttt{tree}\,\tau_2\, y) \end{aligned}$$

These two cases are exclusive. For the first to be true $i$ must be an atom, where in the second it must be a location.

The $\texttt{tree}\,\tau\, i$ predicate is "exact", in the sense that when it is true the current heap must have all and only those heap cells used to represent the tree. If $\tau$ has $n$ pairs in it and $s, h \models \texttt{tree}\,\tau\, i$ then the domain of $h$ has size $2n$.

The specification of the $\texttt{CopyTree}$ procedure is

$$\{\texttt{tree}\,\tau\, p\}\ \texttt{CopyTree}(p; q)\ \{(\texttt{tree}\,\tau\, p) * (\texttt{tree}\,\tau\, q)\}.$$

and here is the code.

```
procedure CopyTree($p; q$)
newvar $i, j, i', j'$.
$\{\mathtt{tree}\,\tau\, p\}$
if isatom?($p$) then
   $\{\tau = p \wedge \mathtt{isatom?}(p) \wedge \mathtt{emp}\}$
   $\{(\mathtt{tree}\,\tau\, p) * (\mathtt{tree}\,\tau\, p)\}$
    $q := p$
   $\{(\mathtt{tree}\,\tau\, p) * (\mathtt{tree}\,\tau\, q)\}$
else
   $\{\exists \tau_1, \tau_2, x, y.\, \tau \doteq (\tau_1, \tau_2) * (p \mapsto x, y) * (\mathtt{tree}\,\tau_1\, x) * (\mathtt{tree}\,\tau_2\, y)\}$
    $i := [p]$; $j := [p+1]$;
   $\{\exists \tau_1, \tau_2.\, \tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j)\}$
    CopyTree($i; i'$);
   $\{\exists \tau_1, \tau_2.\, \tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j) * (\mathtt{tree}\,\tau_1\, i')\}$
    CopyTree($j; j'$);
   $\{\exists \tau_1, \tau_2.\, \tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j) * (\mathtt{tree}\,\tau_1\, i')$
    $*(\mathtt{tree}\,\tau_2\, j')\}$
    $q := \mathtt{cons}(i', j')$
   $\{\exists \tau_1, \tau_2.\, \tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j) * (\mathtt{tree}\,\tau_1\, i')$
    $*(\mathtt{tree}\,\tau_2\, j') * (q \mapsto i', j')\}$
   $\{(\mathtt{tree}\,\tau\, p) * (\mathtt{tree}\,\tau\, q)\}$
```

Most of the steps are straightforward, but the two recursive calls deserve special comment. In proving the body of the procedure we get to use the specification of CopyTree as an assumption. But at first sight the specification does not appear to be strong enough, since we need to be sure that CopyTree($i; i'$) does not affect the assertions $p \mapsto i, j$ and $\mathtt{tree}\,\tau_2\, j$. Similarly, we need that CopyTree($j; j'$) does not affect $\mathtt{tree}\,\tau_1\, i'$.

These "does not affect" properties are obtained from two instances of the Frame Rule:

$$\frac{\{\mathtt{tree}\,\tau_1\, i\}\ \mathtt{CopyTree}(i; i')\ \{(\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_1\, i')\}}{\begin{array}{l}\{\tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j)\} \\ \mathtt{CopyTree}(i; i') \\ \{\tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j) * (\mathtt{tree}\,\tau_1\, i')\}\end{array}}$$

and

$$\frac{\{\mathtt{tree}\,\tau_2\, j\}\ \mathtt{CopyTree}(j; j')\ \{(\mathtt{tree}\,\tau_2\, j) * (\mathtt{tree}\,\tau_2\, j')\}}{\begin{array}{l}\{\tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j) * (\mathtt{tree}\,\tau_1\, i')\} \\ \mathtt{CopyTree}(j; j') \\ \{\tau \doteq (\tau_1, \tau_2) * (p \mapsto i, j) * (\mathtt{tree}\,\tau_1\, i) * (\mathtt{tree}\,\tau_2\, j) * (\mathtt{tree}\,\tau_1\, i') * (\mathtt{tree}\,\tau_2\, j')\}.\end{array}}$$

Then, the required triples for the calls are obtained using Auxiliary Variable Elimination to introduce $\exists \tau_1, \tau_2$. (It would also have been possible to strip the existential at the beginning of the proof of the else part, and then reintroduce it after finishing instead of carrying it through the proof.)

This section illustrates two main points. First, if one does not have some way of representing or inferring frame axioms, then the proofs of even simple programs with procedure calls will not go through. In particular, for recursive programs attention to framing is essential if one is to obtain strong enough induction hypotheses. The `CopyTree` procedure could not be verified without the Frame Rule, unless we were to complicate the initial specification by including some explicit representation of frame axioms.

Second, the specification of `CopyTree` illustrates the idea of a specification that concentrates only on those cells that a program accesses. And of course these two points are linked; we need some way to infer frame axioms, or else such a specification would be too weak.

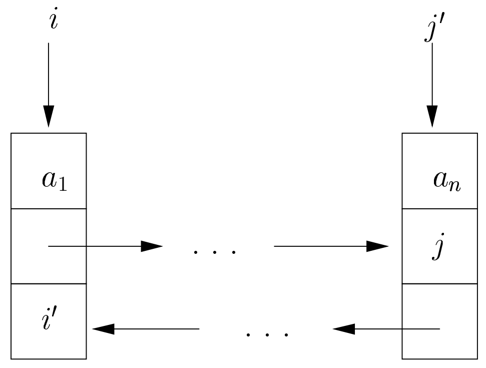
## 7 Difference-linked Lists

The purpose of this section is to illustrate the treatment of address arithmetic, and also disposal. We do this by considering a space-saving representation of doubly-linked lists.

Conventionally, a node in a doubly-linked list contains a data field, together with a field storing a pointer $n$ to the next node and another storing a pointer $p$ to the previous node. In the difference representation we store $n - p$ in a single field rather than have separate fields for $n$ and $p$. In a conventional doubly-linked list it is possible to move either forwards or backwards from a given node. In a difference-linked list given the current node $c$ we can lookup the difference $d = n - p$ between next and previous pointers. This difference does not, by itself, give us enough information to determine either $n$ or $p$. However, if we also know $p$ we can calculate $n$ as $d + p$, and similarly given $n$ we can obtain $p$ as $n - d$. So, using the difference representation, it is possible to traverse the list in either direction as long as we keep track of the previous or next node as we go along.

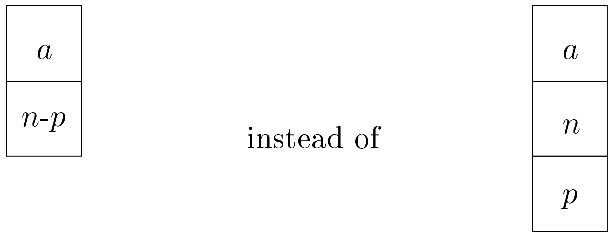
A similar, more time-efficient, representation is sometimes given using the xor of pointers rather than their difference.

We now give a definition of a predicate `dl`. If we were working with conventional doubly-linked lists then $\mathtt{dl}\, a_1 \cdots a_n\, (i, i', j, j')$ would correspond to



Typically, a doubly-linked list with front $i$ and back $j'$ would satisfy the predicate $\mathtt{dl}\,\alpha\,(i, \mathtt{nil}, \mathtt{nil}, j')$. The reason for the internal nodes $i'$ and $j$ is to allow us to consider partial lists, not terminated by `nil`.

A definition of `dl` for conventional doubly-linked lists was given in [20]. The main alteration we must make is to use

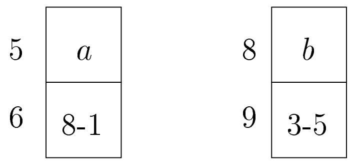


to represent a node.

Here is the definition.

$$\begin{aligned}\mathtt{dl}\,\epsilon\,(i, i', j, j') &\overset{\Delta}{\iff} \mathtt{emp} \wedge i = j \wedge i' = j' \\ \mathtt{dl}\,a\alpha\,(i, i', k, k') &\overset{\Delta}{\iff} \exists j.(i \mapsto a, j - i') * \mathtt{dl}\,\alpha\,(j, i, k, k')\end{aligned}$$

We are using juxtaposition to represent the consing of an element $a$ onto the front of a sequence $\alpha$, and $\epsilon$ to represent the empty sequence. As a small example, $\mathtt{dl}\,ab\,(5, 1, 3, 8)$ is true of



It is instructive to look at how this definition works for a sequence consisting of a single element, $a$. For $\mathtt{dl}\,a\,(i, i', j, j')$ to hold we must have $\exists x.(i \mapsto a, x - i') * \mathtt{dl}\,\epsilon\,(x, i, j, j')$; we can pick $x$ to be $j$, as suggested by the $i = j$ part of the case for $\epsilon$. We are still left, however, with the requirement that $i = j'$, and this in fact leads us to the characterization $i \mapsto a, j - i' \wedge i = j'$ of $\mathtt{dl}\,a\,(i, i', j, j')$.

Thus, a single-lement list exemplifies how the $\epsilon$ case is arranged to be compatible with the operation of consing an element onto the front of a sequence. The roles of the $i = j$ and $i' = j'$ requirements are essentially reversed for the dual operation, of adding a single element onto the end of a sequence. This operation is characterized as follows.

$$\mathtt{dl}\,\alpha a\,(i, i', k, k') \Leftrightarrow \exists j'.\,\mathtt{dl}\,\alpha\,(i, i', k', j') * k' \mapsto a, k - j'$$

In the examples to come we will also use the following properties.

$$\begin{aligned}j' \neq \mathtt{nil} \wedge \mathtt{dl}\,\alpha\,(i, \mathtt{nil}, j, j') &\Rightarrow \exists \beta, a, k.\,\alpha \doteq \beta a * \\ &\qquad \mathtt{dl}\,\beta\,(i, i', j', k) * j' \mapsto a, j - k \\ \mathtt{dl}\,\alpha\,(i, i', j, \mathtt{nil}) &\Rightarrow \mathtt{emp} \wedge \alpha = \epsilon \wedge i' = \mathtt{nil} \wedge i = j \\ \mathtt{dl}\,\alpha\,(\mathtt{nil}, i', j, j') &\Rightarrow \mathtt{emp} \wedge \alpha = \epsilon \wedge j = \mathtt{nil} \wedge i' = j'\end{aligned}$$

Doubly-linked lists are often used to implement queues, because they make it easy to work at either end. We axiomatize an enqueue operation.

Rather than give the code all at once, it will be helpful to use a procedure to encapsulate the operation of *setting a right pointer*. Suppose we are in the position of having a pointer $j'$, whose difference field represents pointing on the right to, say, $j$. We want to swing the right pointer so that it points to $k$ instead. The specification of the procedure is

$$\{\mathtt{dl}\,\alpha\,(i,\mathtt{nil},j,j')\}\;\mathtt{setrptr}(j,j',k;i)\{\mathtt{dl}\,\alpha\,(i,\mathtt{nil},k,j')\}.$$

Notice that this specification handles the $\alpha=\epsilon$ case, when $j'$ does not point to an active cell.

Postponing the definition and proof of `setrptr` for a moment, we can use it to verify a code fragment for putting an value $a$ on the end of a queue.

```
$\{\mathtt{dl}\,\alpha\,(\mathit{front},\mathtt{nil},\mathtt{nil},\mathit{back})\}$
 $t := \mathit{back};$
$\{\mathtt{dl}\,\alpha\,(\mathit{front},\mathtt{nil},\mathtt{nil},t)\}$
 $\mathit{back} := \mathtt{cons}(a,\mathtt{nil}-t);$
$\{\mathtt{dl}\,\alpha\,(\mathit{front},\mathtt{nil},\mathtt{nil},t)\;*\;\mathit{back}\mapsto a,\mathtt{nil}-t\}$
 $\mathtt{setrptr}(\mathtt{nil},t,\mathit{back};\mathit{front})$
$\{\mathtt{dl}\,\alpha\,(\mathit{front},\mathtt{nil},\mathit{back},t)\;*\;\mathit{back}\mapsto a,\mathtt{nil}-t\}$
$\{\mathtt{dl}\,\alpha a\,(\mathit{front},\mathtt{nil},\mathtt{nil},\mathit{back})\}$
```

The code creates a new node containing the value $a$ and the difference $\mathtt{nil}-t$. Then, the procedure call $\mathtt{setrptr}(\mathtt{nil},t,\mathit{back};\mathit{front})$ swings the right pointer associated with $t$ so that the next node becomes *back*. In the assertions, the effect of $\mathit{back} := \mathtt{cons}(a,\mathtt{nil}-t)$ is axiomatized by tacking $*(\mathit{back}\mapsto a,\mathtt{nil}-t)$ onto its precondition. This sets us up for the call to `setrptr`; because of the placement of $*$ we know that the call will not affect $(\mathit{back}\mapsto a,\mathtt{nil}-t)$. More precisely, the triple for the call is obtained using Variable Substitution to instantiate the specification, and the Frame Rule with $(\mathit{back}\mapsto a,\mathtt{nil}-t)$ as the invariant.

Finally, here is an implementation of $\mathtt{setrptr}(j,j',k;i)$.

```
$\{\mathtt{dl}\,\alpha\,(i,\mathtt{nil},j,j')\}$
 if $j'=\mathtt{nil}$ then
 $\{\alpha=\epsilon\wedge\mathtt{emp}\wedge j'=\mathtt{nil}\}$
   $i := k$
 $\{\alpha=\epsilon\wedge\mathtt{emp}\wedge j'=\mathtt{nil}\wedge i=k\}$
 else
 $\{\exists\alpha',b,p.\,(\alpha\doteq\alpha' b)*\mathtt{dl}\,\alpha'\,(i,\mathtt{nil},j',p)*(j'\mapsto b,j-p)\}$
   newvar $d.\ d := [j'+1]; [j'+1] := k+d-j$
 $\{\exists\alpha',b,p.\,(\alpha\doteq\alpha' b)*\mathtt{dl}\,\alpha'\,(i,\mathtt{nil},j',p)*(j'\mapsto b,k-p)\}$
$\{\mathtt{dl}\,\alpha\,(i,\mathtt{nil},k,j')\}$
```

The tricky part in the verification is the `else` branch of the conditional, where the code has to update the difference field of $j'$ appropriately so that $k$ becomes the next node of $j'$. It updates the field by adding $k$ and subtracting $j$; since the field initially stores $j - p$, where $p$ is the address of the previous node, such calculation results in the value $k - p$.

The use of the temporary variable $d$ in the `else` branch is a minor irritation. We could more simply write $[j'+1] := k + [j'+1] - j$ if we were to allow nesting of $[\cdot]$. An unresolved question is whether, in our formalism, such nesting could be dealt with in a way simpler than compiling it out using temporary variables.

Now we sketch a similar development for code that implements a dequeue operation. In this case, we use a procedure $\mathtt{setlptr}(i, i', k; j')$, which is similar to `setrptr` except that it swings a pointer to the left instead of to the right.

$$\{\mathtt{dl}\,\alpha\,(i, i', \mathtt{nil}, j')\}\ \mathtt{setlptr}(i, i', k; j')\ \{\mathtt{dl}\,\alpha\,(i, k, \mathtt{nil}, j')\}$$

The dequeue operation removes the first element of a queue and places its data in $x$.

```
$\{\mathtt{dl}\,a\alpha\,(\mathit{front}, \mathtt{nil}, \mathtt{nil}, \mathit{back})\}$
$\{\exists n'.\, \mathit{front} \mapsto a, n' - \mathtt{nil}\, *\, \mathtt{dl}\,\alpha\,(n', \mathit{front}, \mathtt{nil}, \mathit{back})\}$
 $x := [\mathit{front}];\ d := [\mathit{front}+1];\ n := d + \mathtt{nil};$
$\{x \doteq a\, *\, \mathit{front} \mapsto a, n - \mathtt{nil}\, *\, \mathtt{dl}\,\alpha\,(n, \mathit{front}, \mathtt{nil}, \mathit{back})\}$
 $\mathtt{dispose}(\mathit{front}); \mathtt{dispose}(\mathit{front}+1);$
$\{x \doteq a\, *\, \mathtt{dl}\,\alpha\,(n, \mathit{front}, \mathtt{nil}, \mathit{back})\}$
 $\mathtt{setlptr}(n, \mathit{front}, \mathtt{nil}; \mathit{back})$
$\{x \doteq a\, *\, \mathtt{dl}\,\alpha\,(n, \mathtt{nil}, \mathtt{nil}, \mathit{back})\}$
```

This code stores the data of the first node in the variable $x$ and obtains the next pointer $n$ using arithmetic with the difference field. The placement of $*$ sets us up for disposing $\mathit{front}$ and $\mathit{front}+1$: The precondition to these two commands is equivalent to an assertion of the form $(\mathit{front} \mapsto a) * (\mathit{front}+1 \mapsto n' - \mathtt{nil}) * R$, which is compatible with what is given by two applications of the weakest precondition rule for `dispose`. After the disposals have been done, the procedure call $\mathtt{setlptr}(n, \mathit{front}, \mathtt{nil}; \mathit{back})$ resets the difference field of the node $n$ so that its previous node becomes `nil`.

The code for $\mathtt{setlptr}(i, i', k; j')$ is as follows.

```
$\{\mathtt{dl}\,\alpha\,(i, i', \mathtt{nil}, j')\}$
 if $i = \mathtt{nil}$ then
 $\{\alpha = \epsilon \wedge \mathtt{emp} \wedge i = \mathtt{nil}\}$
   $j' := k$
 $\{\alpha = \epsilon \wedge \mathtt{emp} \wedge i = \mathtt{nil} \wedge k = j'\}$
 else
 $\{\exists \alpha', a, n.\, (\alpha \doteq a\alpha') * \mathtt{dl}\,\alpha'\,(n, i, \mathtt{nil}, j') * (i \mapsto a, n - i')\}$
   $[i+1] := [i+1] + i' - k$
 $\{\exists \alpha', a, n.\, (\alpha \doteq a\alpha') * \mathtt{dl}\,\alpha'\,(n, i, \mathtt{nil}, j') * (i \mapsto a, n - k)\}$
$\{\mathtt{dl}\,\alpha\,(i, k, \mathtt{nil}, j')\}$
```

## 8 Memory Faults and Tight Specifications

In this paper we will not include a semantics of commands or precise interpretation of triples, but in this section we give an informal discussion of the semantic properties of triples that the axiom system relies on.

Usually, the specification form $\{P\}C\{Q\}$ is interpreted "loosely", in the sense that $C$ might cause state changes not described by the pre and postcondition. This leads to the need for explicit frame axioms. An old idea is to instead consider a "tight" interpretation of $\{P\}C\{Q\}$, which should guarantee that $C$ only alters those resources mentioned in $P$ and $Q$; unfortunately, a precise definition of the meaning of tight specifications has proven elusive [1]. However, the description of local reasoning from the Introduction, where a specification and proof concentrate on a circumscribed area of memory, requires something like tightness. The need for a tight interpretation is also clear from the small axioms, or the specifications of `setlptr`, `setrptr` and `CopyTree`.

To begin, the model here calls for a notion of memory fault. This can be pictured by imagining that there is an "access bit" associated with each location, which is on iff the location is in the domain of the heap. Any attempt to read or write a location whose access bit is off causes a memory fault, so if $E$ is not an active address then $[E] := E'$ or $x := [E]$ results in a fault. A simple way to interpret $\texttt{dispose}(E)$ is so that it faults if $E$ is not an active address, and otherwise turns the access bit off.

Then, a specification $\{P\}C\{Q\}$ holds iff, whenever $C$ is run in a state satisfying $P$: (i) it does not generate a fault; and (ii) if it terminates then the final state satisfies $Q$. (This is a partial correctness interpretation; the total correctness variant alters (ii) by requiring that there are no infinite reductions.) For example, according to the fault-avoiding interpretation, $\{17 \mapsto -\}\,[17] := 4\,\{17 \mapsto 4\}$ holds but $\{\texttt{true}\}\,[17] := 4\,\{17 \mapsto 4\}$ does not. The latter triple fails because the empty heap satisfies `true` but $[17] := 4$ generates a memory fault when executed in the empty heap.

In the logic, faults are precluded by the assumptions $E \mapsto -$ and $E \mapsto n$ in the preconditions of the small axioms for $[E] := E'$, $x := [E]$ and $\texttt{dispose}(E)$.

The main point of this section is that this fault-avoiding interpretation of $\{P\}C\{Q\}$ gives us a precise formulation of the intuitive notion of tightness. (We emphasize that this requires faults, or a notion of enabled action, and we do not claim that it constitutes a general analysis of the notion of tight specification.)

> The avoidance of memory faults in specifications ensures that a well-specified program can only dereference (or dispose) those heap cells guaranteed to exist by the precondition, or those which are allocated during execution.

Concretely, if one executes a program proved to satisfy $\{P\}C\{Q\}$, starting in a state satisfying $P$, then memory access bits are unnecessary. A consequence is that it is not necessary to explicitly describe all the heap cells that don't change, because those not mentioned automatically stay the same.

Fault avoidance in $\{P\}C\{Q\}$ ensures that if $C$ is run in a state strictly larger than one satisfying $P$, then any additional cells must stay unchanged; an attempt to write any of the additional cells would falsify the specification, because it would generate a fault when applied to a smaller heap satisfying $P$. For example, if $\{17 \mapsto -\}\, C\, \{17 \mapsto 4\}$ holds then $\{(17 \mapsto -) * (19 \mapsto 3)\}\, C\, \{(17 \mapsto 4) * (19 \mapsto 3)\}$ should as well, as mandated by the Frame Rule, because any attempt to dereference address 19 would falsify $\{17 \mapsto -\}\, C\, \{17 \mapsto 4\}$ if we give $C$ a state where the access bit for 19 is turned off. (This last step is delicate, in that one could entertain operations, such as to test whether an access bit is on, which contradict it; what is generally needed for it is a notion which can be detected in the logic but not the programming language.)

## 9 Conclusion

We began the paper by suggesting that the main challenge facing verification formalisms for pointer programs is to capture the informal local reasoning used by programmers, or in textbook-style arguments about data structures. Part of the difficulty is that pointers exacerbate the *frame problem* [9, 1]. (It is only part of the difficulty because the frame problem does not, by itself, say anything about aliasing.) For imperative programs the problem is to find a way, preferably succinct and intuitive, to describe or imply the frame axioms, which say what memory cells are not altered by a program or procedure. Standard methods, such as listing the variables that might be modified, do not work easily for pointer programs, because there are often many cells not directly named by variables in a program or program fragment. These cells might be accessed by a program by following pointer chains in memory, or they might not be accessed even when they are reachable.

The approach taken here is based on two ideas. The first, described in Section 8, to use a fault-avoiding interpretation of triples to ensure that additional cells, active but not described by a precondition, are not altered during execution. The second is to use the $*$ connective to infer invariant properties implied by these tight specifications.

The frame problem for programs is perhaps more approachable than the general frame problem. Programs come with a clear operational semantics, and one can appeal to concrete notions such as a program's footprint. But the methods here also appear to be more generally applicable. It would be interesting to give a precise comparison with ideas from the AI literature [22], as well as with variations on Modifies clauses [1, 8]. We hope to report further on these matters – in particular on the ideas outlined in Section 8 – in the future. (Several relevant developments can be found in Yang's thesis [24].)

There are several immediate directions for further work. First, the interaction between local and global reasoning is in general difficult, and we do not mean to imply that things always go as smoothly as in the example programs we chose. They fit our formalism nicely because their data structures break naturally into disjoint parts, and data structures that use more sharing are more difficult to handle. This includes tree representations that allow sharing of subtrees, and graph structures. Yang has treated a nontrivial example, the Shorr-Waite graph marking algorithm, using the spatial implication $-\!*$ is used to deal with the sharing found there [23]. More experience is needed in this direction. Again, the challenging problem is not to find a system that is adequate in principle, but rather is to find rules or reasoning idioms that cover common cases simply and naturally.

Second, the reasoning done in examples in this paper is only semi-formal, because we have worked semantically when applying the Rule of Consequence. We know of enough axioms to support a number of examples, but a comprehensive study of the proof theory of the assertion language is needed. Pym has worked out a proof theory of the underlying logic BI [17] that we can draw on. But here we use a specific model of BI and thus require an analysis of properties special to that model. Also needed is a thorough treatment of recursive definitions of predicates.

Finally, the examples involving address arithmetic with difference-linked lists are simplistic. It would be interesting to try to verify more substantial programs that rely essentially on address arithmetic, such as memory allocators or garbage collectors.

ACKNOWLEDGEMENTS.

O'Hearn would like to thank Richard Bornat, Cristiano Calcagno and David Pym for discussions about local reasoning and bunched logic. He was supported by the EPSRC, under the "Verified Bytecode" and "Local Reasoning about State" grants. Reynolds was supported by NSF grant CCR-9804014. Yang was supported by the NSF under grant INT-9813854.

## References

[1] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions of Software Engineering*, 21:809–838, 1995.

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

[3] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[4] C. Calcagno, S. Isthiaq, and P. W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. Proceedings of the Second International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 2000.

[5] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 843–993. Elsevier, Amsterdam, and The MIT Press, Cambridge, Mass., 1990.

[6] C. A. R. Hoare and J. He. A trace model for pointers and objects. In Rachid Guerraoui, editor, *ECCOP'99 - Object-Oriented Programming, 13th European Conference*, pages 1–17, 1999. Lecture Notes in Computer Science, Vol. 1628, Springer.

[7] S. Isthiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of the Twenty-Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, London, January 2001.

[8] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Technical Report Reearch Report 160, Compaq Systems Research Center, Palo Alto, CA, November 2000.

[9] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[10] P. W. O'Hearn. Resource interpretations, bunched implications and the $\alpha\lambda$-calculus. In *Typed $\lambda$-calculus and Applications*, J-Y Girard editor, L'Aquila, Italy, April 1999. Lecture Notes in Computer Science 1581.

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

[12] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):267–223, January 2000.

[13] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, May 1995. Also in [14], vol 2, pages 109–164.

[14] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages.* Two volumes, Birkhauser, Boston, 1997.

[15] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages.* Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.

[16] F. J. Oles. Functor categories and store shapes. In O'Hearn and Tennent [14], pages 3–12. Vol. 2.

[17] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications.* Monograph to appear, 2001.

[18] J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, January 1978. ACM, New York. Also in [14], vol 1.

[19] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland, Amsterdam. Also in [14], vol 1, pages 67-88.

[20] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

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

[22] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[23] H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. Manuscript, October 2000.

[24] H. Yang. *Local Reasoning for Stateful Programs.* Ph.D. thesis, University of Illinois, Urbana-Champaign, Illinois, USA, 2001 (expected).